



# Object Oriented Programming

(송태국 ver 1.9.0)

# 목 차

---

- ✓ 객체지향기술.기술흐름과 객체지향
- ✓ 객체지향개념.종합적인 이해
- ✓ 객체지향설계원칙.단일책임원칙(SRP):구구단
- ✓ 객체지향설계원칙.개방폐쇄원칙(OCP):다중 매핑
- ✓ 협업.Radio
- ✓ 협업과 추상화.crosswalk 1,2,3

# 강의 목표와 구성

- ✓ 현대의 프로그래밍에서 Java, C++, Scala 등과 같은 객체 지향 프로그램 언어를 주로 사용합니다.
- ✓ 누구나 객체 지향(Object Oriented) 개념을 배웠고 알고 있지만 프로그래밍에 제대로 적용하지 못합니다.
- ✓ 이 모듈은 객체 지향의 주요 특성 중에 실세계 맵핑(real world mapping)을 중심으로 실세계에서 일어나는 객체들 간의 끝없는 협업(collaboration)을 시스템 세계에 적용하는 훈련을 합니다.

모듈 목표		
제목	유형	강사
객체지향기술.기술흐름과 객체지향	이론/토론	
객체지향개념.종합적인 이해	이론/토론	
객체지향설계원칙.단일책임원칙(SRP):구구단	이론/실습	
객체지향설계원칙.개방폐쇄원칙(OCP):다중 맵핑	이론/실습	
협업.Radio	이론/실습	송태국
협업과 추상화: crosswalk 1,2,3	이론/실습	



## 모듈: 객체 지향 기술

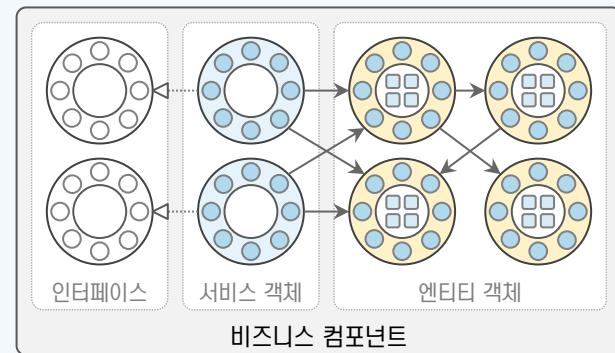
- ❖ 기술의 흐름 속에서 객체 지향이 차지하는 위상을 알아 봅니다.
- ❖ 우리의 현실 속에 객체 지향은 어떻게 다루어지는지 알아 봅니다.
- ❖ 언어의 변환이라는 관점에서 시스템 개발을 이해합니다.

## 기술 흐름과 객체 지향

- ✓ 왜 객체 지향인가?
- ✓ 우리들의 객체 지향
- ✓ 언어 매핑
- ✓ 요약

# 왜 객체 지향인가? (1/4)

- ✓ SW 설계 접근방법 → 절차 지향 설계와 객체 지향 개념을 바탕으로 하는 설계(OOAD, CBD, SOA, MSA)
- ✓ 두 세계는 패러다임이 전혀 다릅니다. 프로그래밍 언어 역시 두 가지로 나눌 수 있습니다.
- ✓ 객체 지향 설계 이후, SW 설계는 비약적으로 성장하였으며, 현재에 이르러서는 모든 산업의 기반이 되었습니다.
- ✓ OOAD는 많은 문제를 안고 있었지만, 그것을 극복하고 컴포넌트 기반 설계로 발전하였습니다.



질서정연한 객체들의 집합으로 구성한 비즈니스 컴포넌트

1995년 ~

OOAD (객체 지향 분석 설계)

절차 지향 분석 설계

2002년 ~

CBD (컴포넌트 기반 개발)

2008년 ~

SOA (서비스 지향 아키텍처)

MSA (마이크로 서비스 아키텍처)

객체지향

온도<100 일 때만  
온도 1도 증가

1도 증가 요청

물의 온도

절차지향

데이터

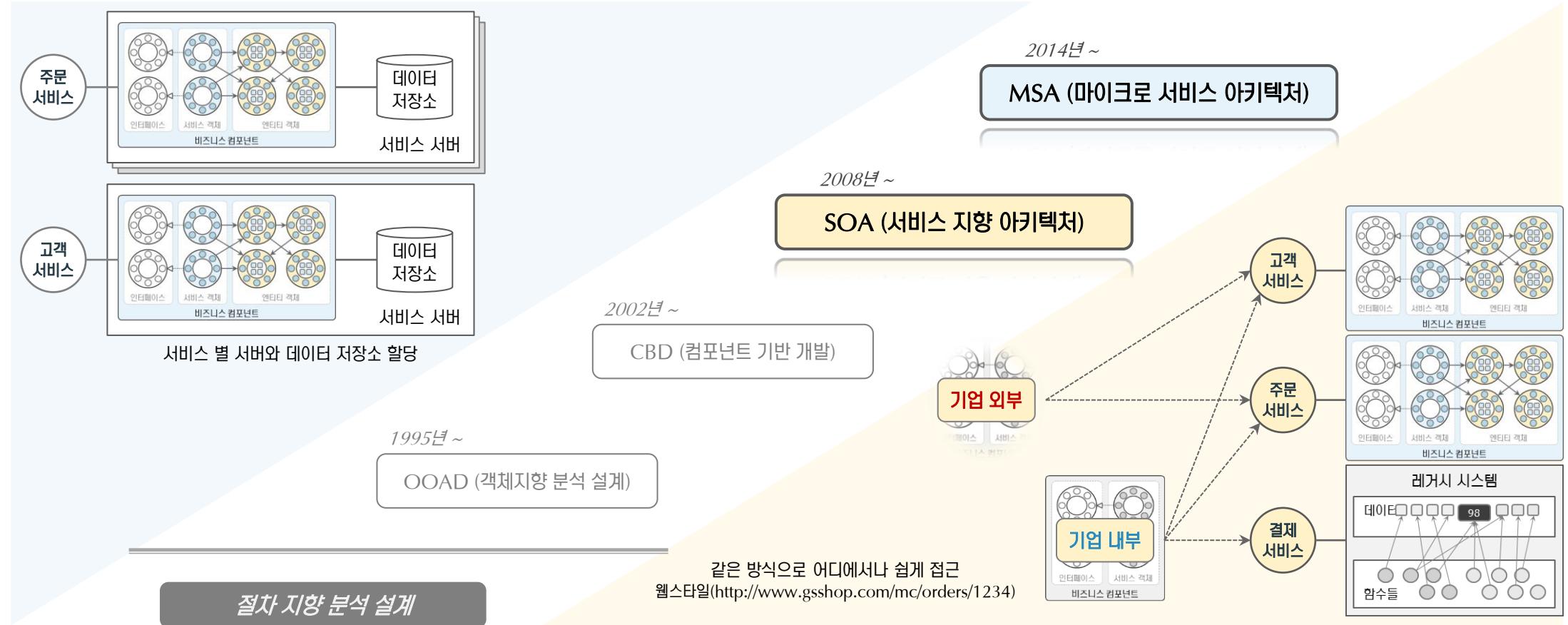
함수들

물의 온도

98

# 왜 객체 지향인가? (2/4)

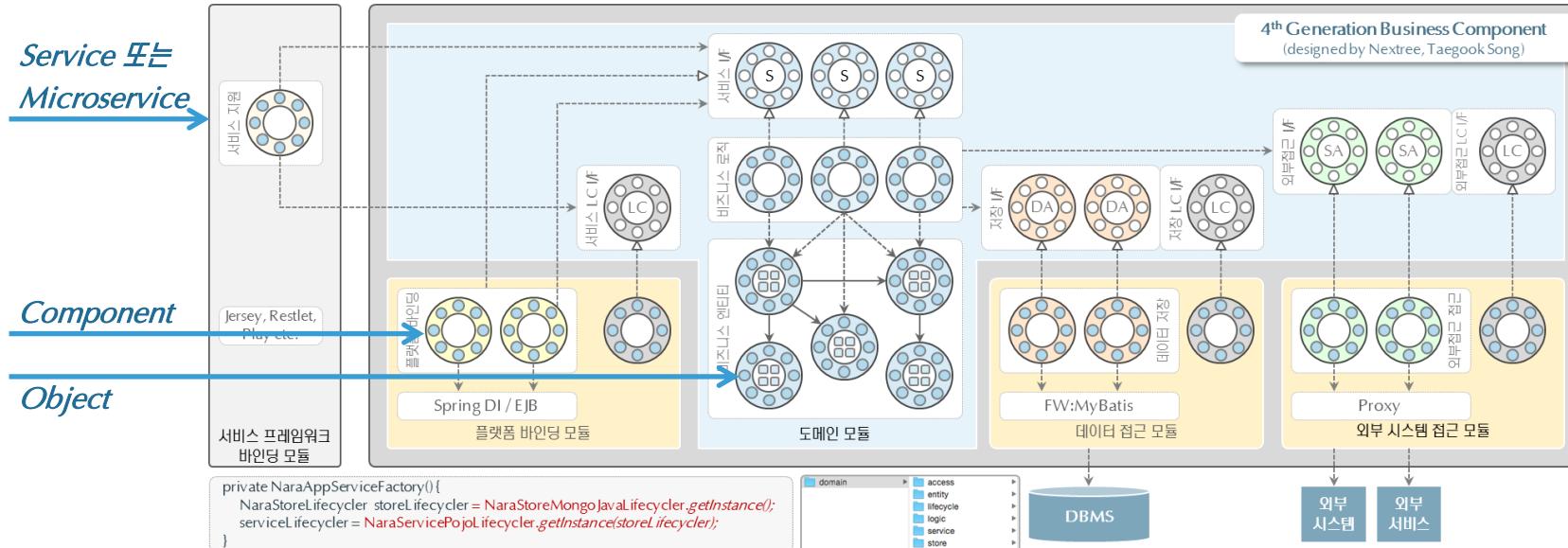
- ✓ 복잡한 컴퓨팅 환경은 기업 내부 분산을 넘어서 기업 간 분산 환경으로 발전하였습니다.
- ✓ SOA는 컴포넌트를 뛰어 넘어, 비즈니스 단위로 써의 서비스로 발전하였으며, 다양한 시련을 겪어왔습니다.
- ✓ 컴포넌트 다음 세대의 서비스는 플랫폼과 환경, 표준 등의 문제를 극복하면서 마이크로 서비스로 발전하였습니다.
- ✓ 이전 기술을 버리고 새로운 기술로 발전한 것이 아니라, 이전 기술의 축적을 바탕으로 발전하여 갔습니다.



# 왜 객체 지향인가? (3/4)

- ✓ 객체 지향 기술과 컴포넌트 기술은 퇴색되지 않고, 서비스 저 안쪽에 탄탄하게 자리잡고 있습니다.
- ✓ 객체 모델링 역량은 컴포넌트와 서비스 내부를 잘 채우는데 도움을 줍니다.
- ✓ 컴포넌트 모델링 역량이 있어야 기술적으로 또는 업무적으로 의미 있는 객체 그룹을 구성할 수 있습니다.
- ✓ 무늬만 컴포넌트로 만드는 이유는 객체 모델링 역량과 컴포넌트 모델링 역량이 부족하기 때문입니다.

[ 4세대 비즈니스 컴포넌트와 서비스 ]

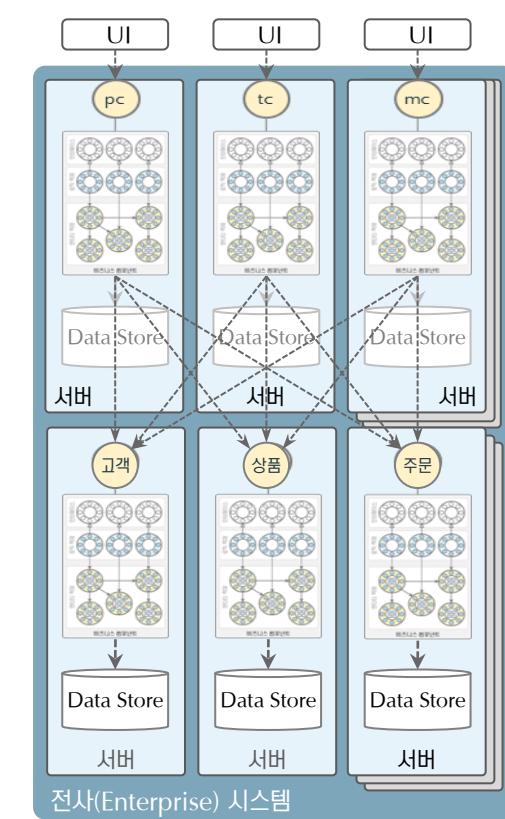
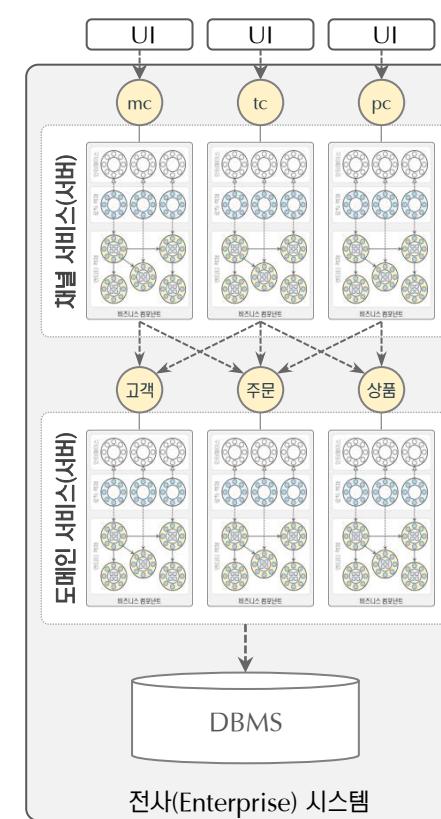
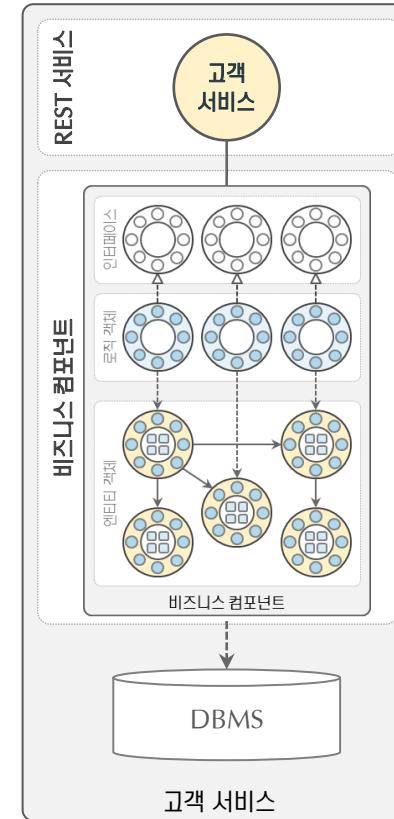
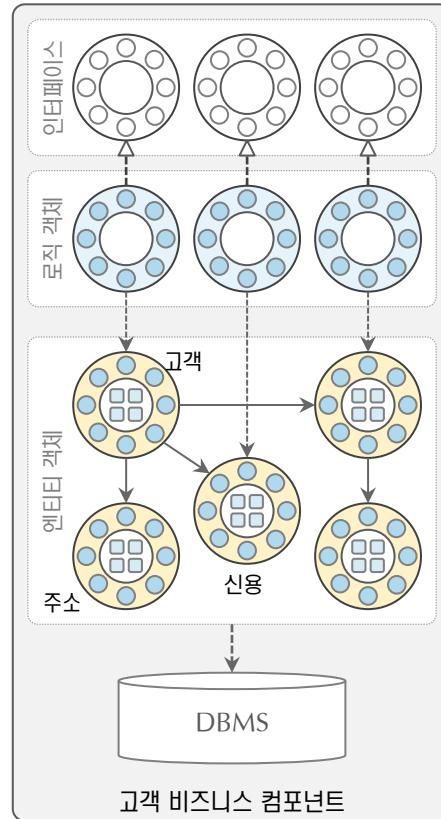


# 왜 객체 지향인가? (4/4)

- ✓ 객체 지향 프로그래밍 언어 출현 이후, 기술의 흐름은 모두 객체를 기반으로 하고 있습니다.
- ✓ 컴포넌트는 객체 들의 질서정연한 구조를 의미하고, 서비스는 컴포넌트를 사용하는 방법을 의미합니다.
- ✓ CBD → SOA → Microservices 로의 흐름 내내 객체는 내부를 채우고 있습니다.

```
public class TdDiffNodeIterator {
    private int nodeSize;
    private int currentIndex;
    private List<TdDiffNode> attrNodeList;
    private TdDiffNodeIterator(TdDiffNode attrNode) {
        this.currentIndex = 0;
        this.attrNodeList = attrNodeList.size();
        this.currentNode = attrNode;
    }
    public int size() {
        return attrNodeList.size();
    }
    public boolean hasNext() {
        if (currentIndex < nodeSize) {
            return true;
        }
        return false;
    }
    public TdDiffNode next() {
        if (hasNext()) {
            return attrNodeList.get(currentIndex++);
        } else {
            return null;
        }
    }
}

public class TdDiffNodeIterator {
    private int nodeSize;
    private int currentIndex;
    private List<TdDiffNode> attrNodeList;
    private TdDiffNodeIterator(TdDiffNode attrNode) {
        this.currentIndex = 0;
        this.attrNodeList = attrNodeList.size();
        this.currentNode = attrNode;
    }
    public int size() {
        return attrNodeList.size();
    }
    public boolean hasNext() {
        if (currentIndex < nodeSize) {
            return true;
        }
        return false;
    }
    public TdDiffNode next() {
        if (hasNext()) {
            return attrNodeList.get(currentIndex++);
        } else {
            return null;
        }
    }
}
```



# 우리들의 객체 지향 (1/5)

- ✓ 개발자라면 누구나 “객체 지향” 개념을 이해하고 있고, “객체 지향” 언어를 사용하고 있습니다.
- ✓ 여러분들이 지금까지 배워온 객체 지향에 대해서 토론하는 시간입니다.
- ✓ 객체 지향 개념을 여러분들의 프로그래밍에 어떻게 적용하고 있습니까?
- ✓ 상속(inheritance) 개념을 알고 있기 때문에 여러분의 프로그램 재사용이 늘어났습니까?

→ 객체 지향의 핵심 개념은 무엇입니까?

→ 객체 지향 프로그래밍 언어라서 좋은 점은 무엇입니까?

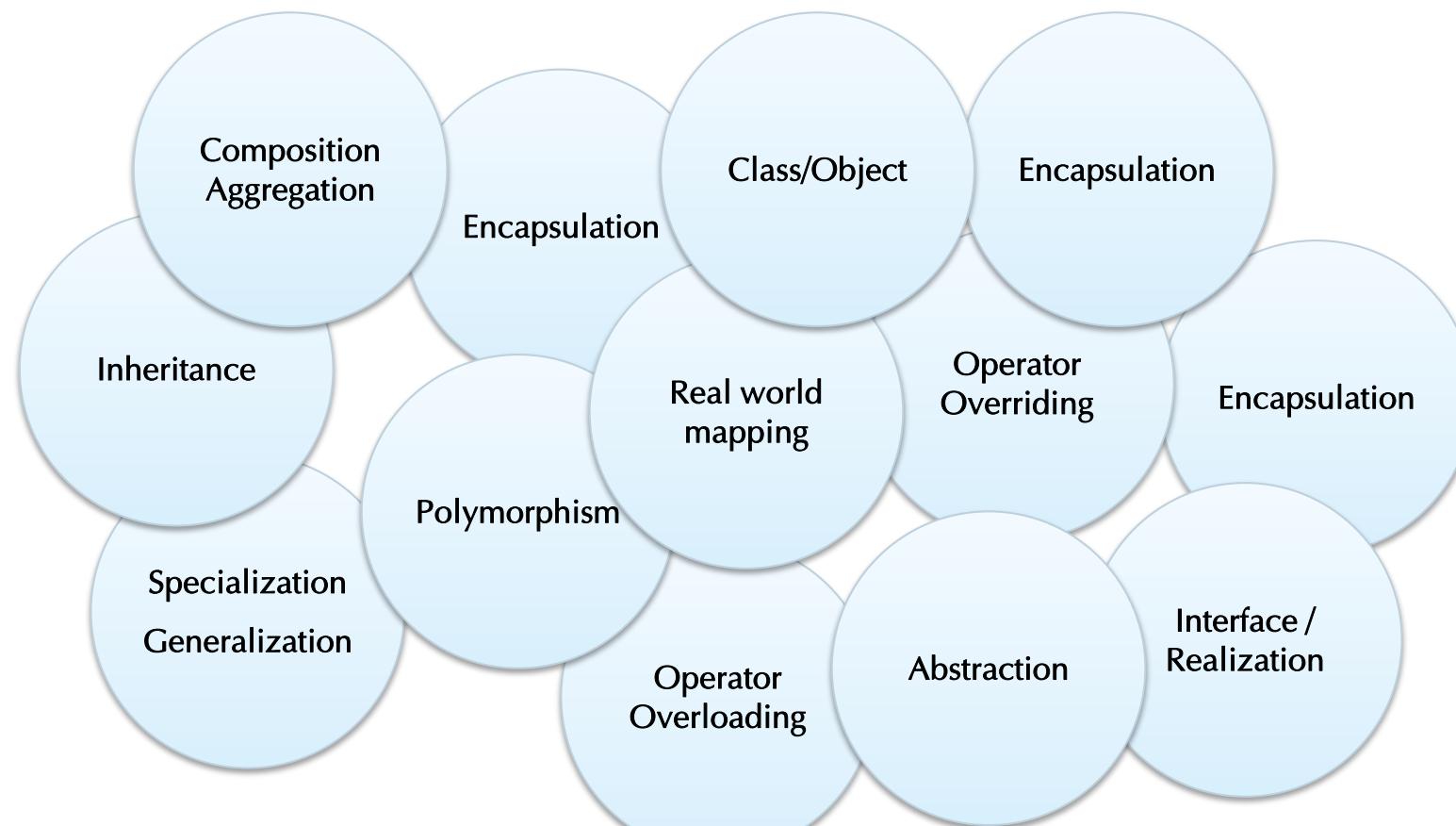
→ 상속은 객체 지향을 대표하는 개념인데 많이 사용합니까?

→ 여러분은 객체 지향 프로그래밍을 하십니까?

→ 여러분의 클래스, 메소드의 라인 수는 얼마입니까?

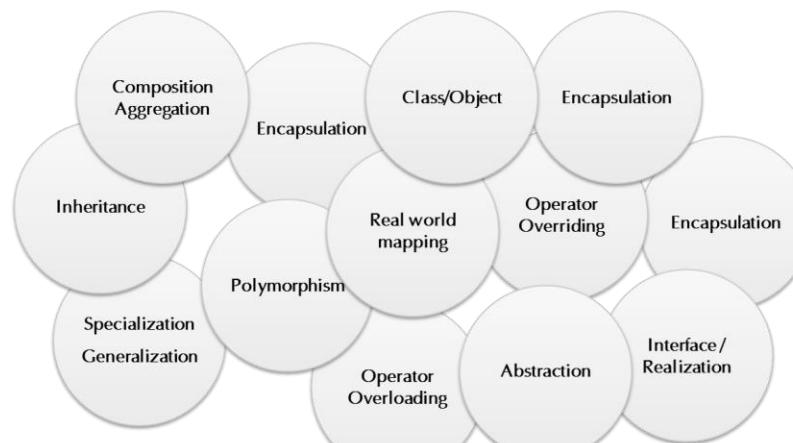
# 우리들의 객체 지향 (2/5)

- ✓ 처음 객체 지향 프로그램을 배울 때를 돌이켜 보면, 생소한 개념 때문에 힘들어 했던 기억이 있을 것입니다.
- ✓ 그런 개념을 이해하고 그 개념들을 프로그래밍 할 때 충분히 활용하였는지 생각해 봅시다.
- ✓ 참 많은 개념들이 있었으며, 그 끝에 남은 것은 클래스, 메소드 등의 개념이 아닌 Syntax는 아니었을까요...
- ✓ 이러한 개념들이 일상의 프로그래밍 작업에 어떤 영향을 주고 있는지 생각해 봅니다.



# 우리들의 객체 지향 (3/5)

- ✓ 처음 배웠던 어렵고 복잡한 객체 지향 개념들은 어디로 갔는지 생각해 봅니다.
- ✓ 여러분의 프로그램 속으로 들어갔다면 여러분의 프로그램은 멋진 프로그램일 수 밖에 없습니다.
- ✓ 어디론가 사라졌다면 여러분은 아직도 “절차 지향” 프로그래밍을 하고 있을 수 있습니다.
- ✓ 객체 지향 개념들이 우리 프로그램에 반영되지 않았다면, 왜 그렇게 되었는지 알아야 합니다.



여러분의 객체 지향 개념

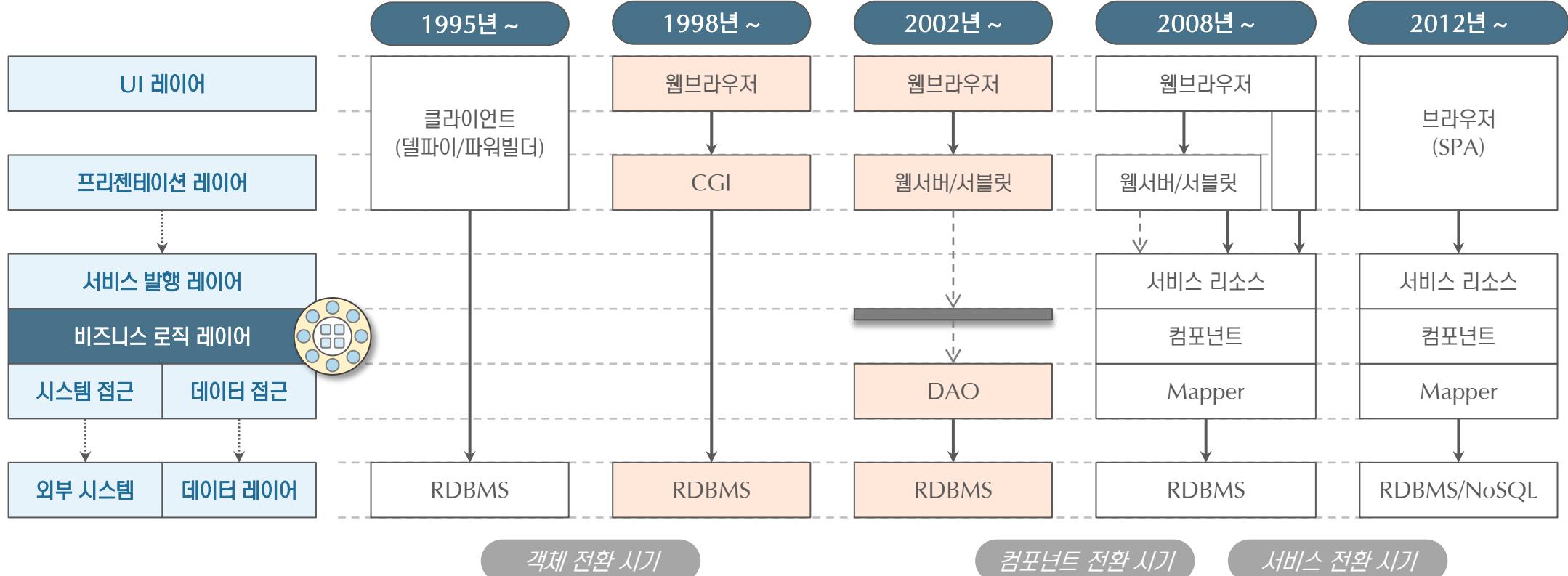
```
import com.nextree.omw.crossroad.Logger;  
  
public class TrafficLight extends TrafficDevice {  
  
    private GreenLight greenLight;  
    private RedLight redLight;  
  
    public TrafficLight(String name, RoadSide side) {  
        super(name);  
        super.setRoadSide(side);  
        this.greenLight = new GreenLight();  
        this.redLight = new RedLight();  
        this.greenLight.off();  
        this.redLight.on();  
    }  
  
    protected void toggleLight() {  
        // 지정시간 토글  
        if (isGreenOn()) {  
            setRedOn();  
        } else {  
            setGreenOn();  
        }  
    }  
  
    protected LightColor getLightColor() {  
        if (isGreenOn()) {  
            return LightColor.Green;  
        } else if (isRedOn()) {  
            return LightColor.Red;  
        }  
        throw new RuntimeException("신호등의 색상이 없습니다...");  
    }  
}
```

여러분의 객체 지향 프로그램



# 우리들의 객체 지향 (4/5) – 기술 유산

- ✓ 기업 컴퓨팅 환경은 2000년 이전 C/S 환경으로부터 시작하여 현재의 MSA까지 발전해 왔습니다.
- ✓ 국내 기업들 대부분 객체로 전환을 놓치고, 그 결과 “무늬만” 컴포넌트 시대를 지나왔습니다.
- ✓ 컴포넌트로 전환하지 못한 기업이 서비스로 전환하는 것은 불가능에 가까운 일이었습니다.
- ✓ 여전히 많은 기업이 1998년 또는 2002년 기술 구조에 머무르고 있으며, 그 원인은 객체 기술 부재입니다.



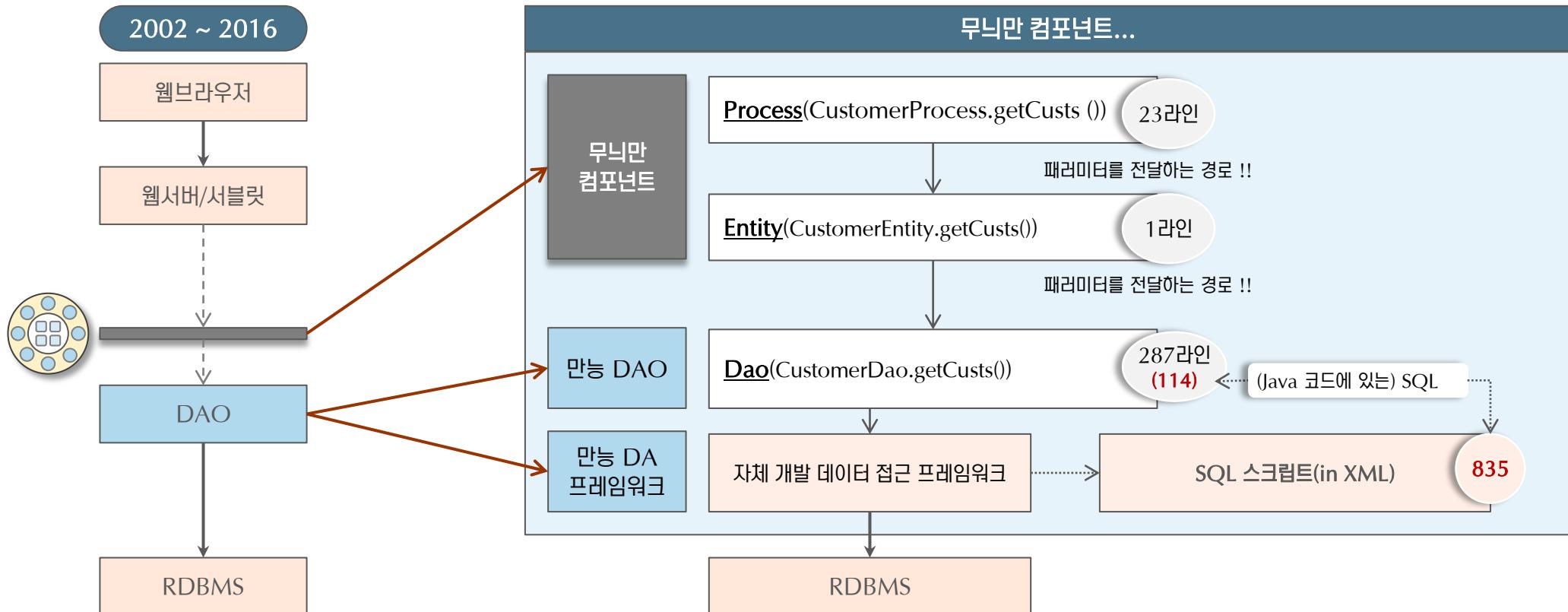
객체 전환 시기

컴포넌트 전환 시기

서비스 전환 시기

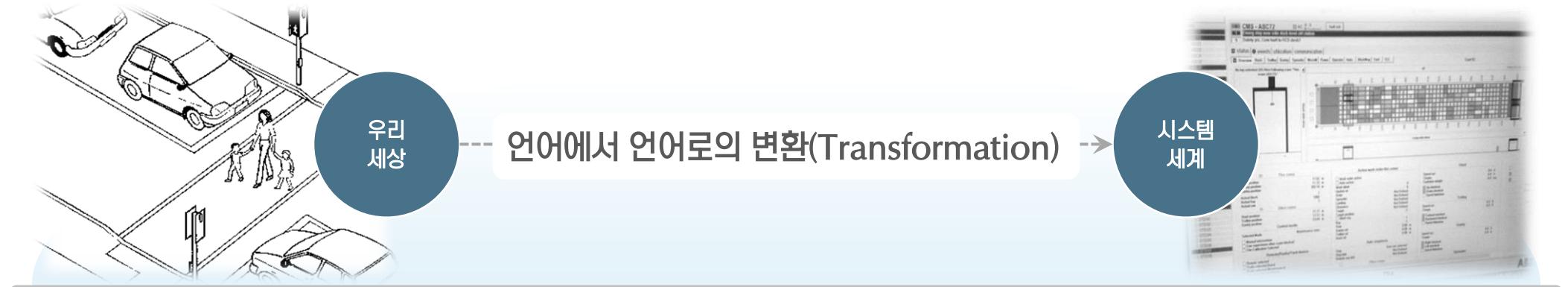
# 우리들의 객체 지향 (5/5) – 기술 유산

- ✓ 설계 기술이 없던 시절 잠시 유행하고 말았어야 할 “무늬만 컴포넌트”가 일상화 되고 말았습니다.
- ✓ 컴포넌트 자체가 없는 구조도 많지만, 있다고 하더라도 컴포넌트를 열어보면 그 속에는 아무것도 없습니다.
- ✓ 도메인 객체, 로직 객체 등으로 가득 차 있어야 할 컴포넌트는 파라미터 전달 경로로 사용할 뿐입니다.
- ✓ 객체가 주로 활동할 공간을 비워 둔 설계 유산 때문에 객체 지향 설계와 프로그래밍이 설 자리를 잃었습니다.



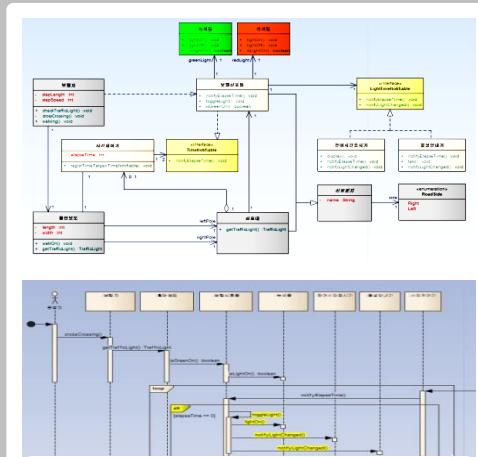
## 언어 매팅(1/3)

- ✓ 우리가 살고 있는 세상에 자주 발생하는 불편함을 해소하기 위해 [SW intensive] 시스템을 개발합니다.
  - ✓ 우리의 불편을 해결할 시스템은 우리가 사용하는 표현으로부터 출발하여 자그마한 시스템 세계를 구성합니다.
  - ✓ 자연어 → 모델링 언어 → 프로그래밍 언어를 거치면서 시스템 세계가 요구하는 특성을 점차 반영합니다.



- 행인이 횡단보도로 다가가며 보행자신호등을 확인한다. 적색등이 켜져있다. 녹색등을 기다리며 멈춰선다.
  - 시간이 흘러 녹색등이 켜지고, 또 시간이 흘러 적색등이 켜진다. 녹색등과 적색등이 교대로 켜진다.
  - 녹색등이 켜지면 음성 안내기는 “녹색등입니다. 건너가십시오.”라는 안내를 하고 “뚜르르, 뚜르르, …”하고 알림음을 낸다. 5초가 남으면 “위험합니다. 건너지 마십시오.”라는 안내를 하고, 0초가 남으면 알림음을 멈춘다.
  - 녹색등이 켜지면 남은시간 표시기에 숫자가 20부터 시작하여 1초 단위로 줄어든다. 적색등이 켜지면 남은 시간 표시기는 깨진다.
  - 행인은 횡단보도를 건너간다. 시간이 부족하면 달린다.

## 요구사항 – 자연언어



## 모델링 – UML

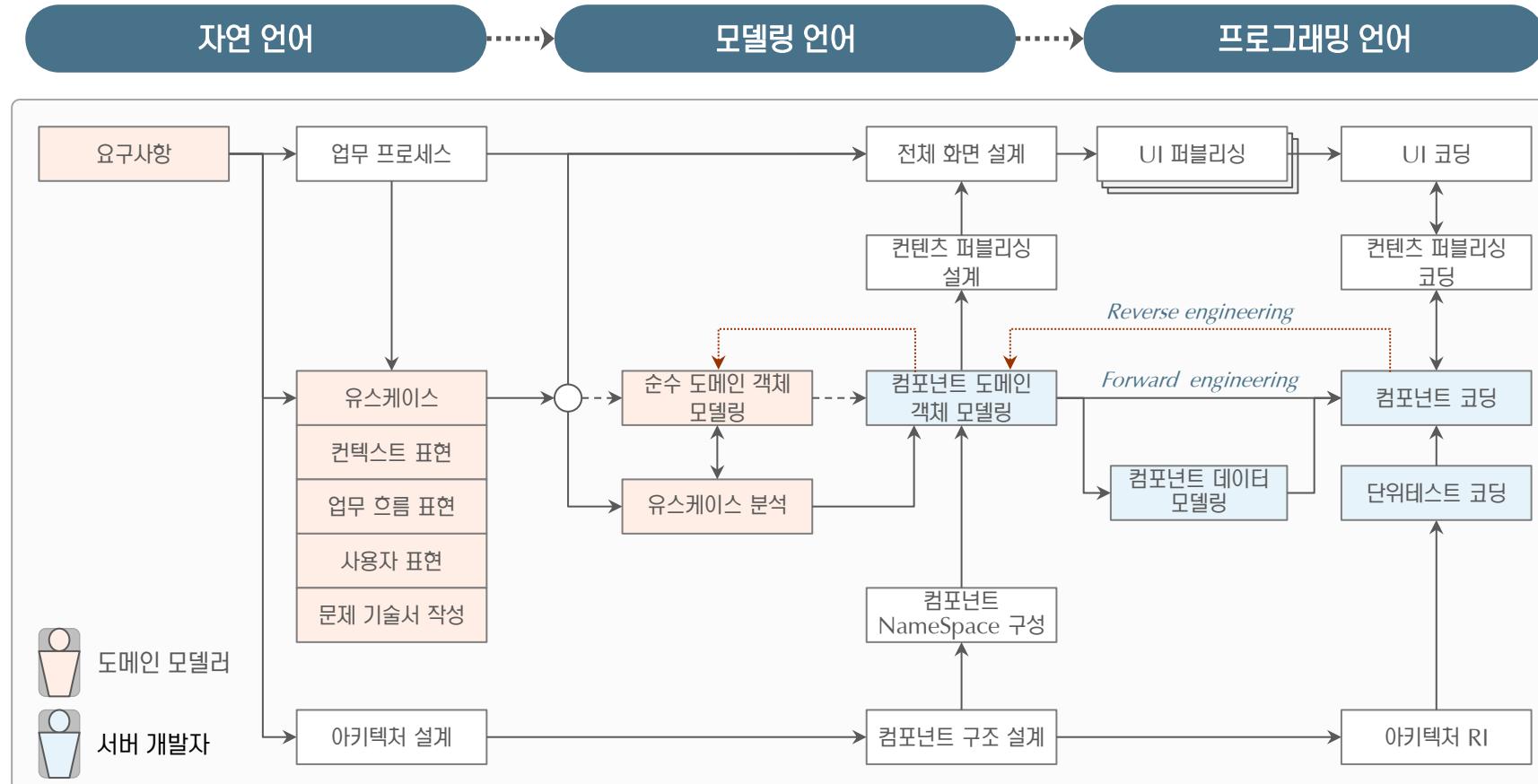
```
import com.nextree.ownm.crossroad.Logger;
public class TrafficLight extends TrafficDevice {
    private GreenLight greenLight;
    private RedLight redLight;
    public TrafficLight(String name, RoadSide side) {
        super(name);
        super.setRoadSide(side);
        this.greenLight = new GreenLight();
        this.redLight = new RedLight();
        this.greenLight.Off();
        this.redLight.on();
    }
    protected void toggleLight() {
        // 0.5초 시간 틈틈이
        if (isGreenOn()) {
            setRedOn();
        } else {
            setGreenOn();
        }
    }
    protected LightColor getLightColor() {
        if (isGreenOn())
            return LightColor.Green;
        else if (isRedOn())
            return LightColor.Red;
        }
        throw new RuntimeException("신호등의 색상이 없습니다...*");
    }
}
```

## 프로그램 – 프로그래밍언어

## 실행코드 – 기계어

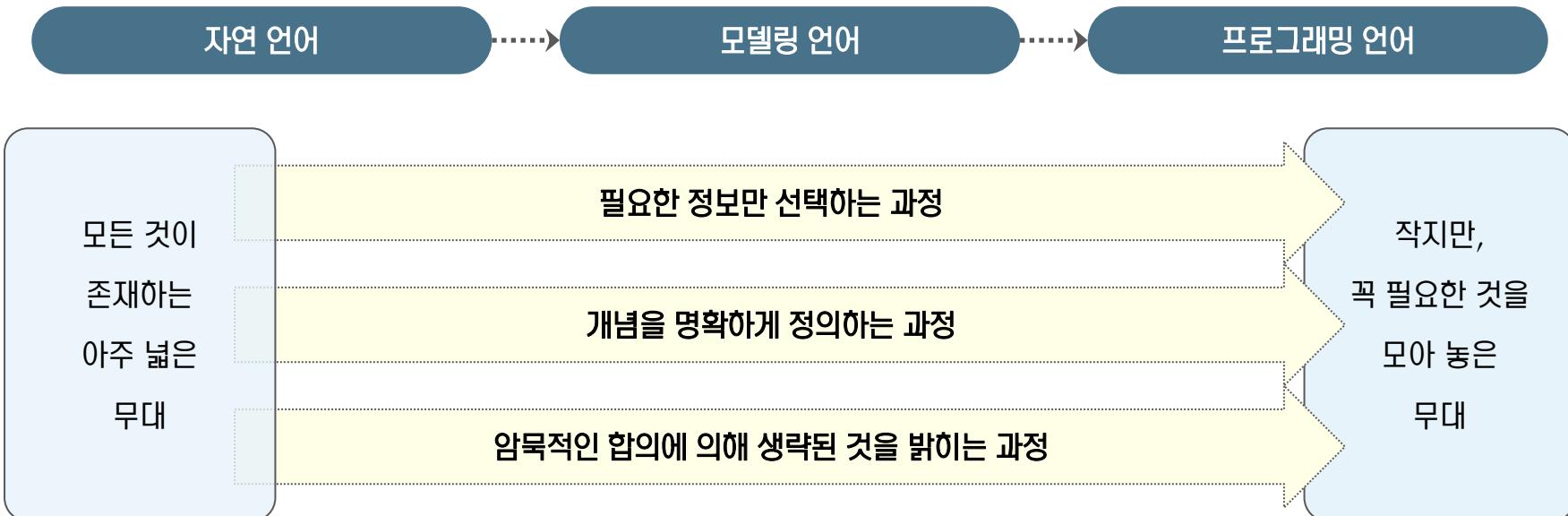
# 언어 매핑(2/3)

- ✓ SW 개발 절차를 살펴보면, 서로 다른 언어로의 변환(transformation)이 이어짐을 알 수 있습니다.
- ✓ 요구사항은 실세계의 자연 언어로 표현하고, 애플리케이션은 시스템 세계의 프로그래밍 언어로 표현합니다.
- ✓ 어떻게 보면, 시스템 세계는 특정 관점을 유지하며 구축한 작은 모형이라고 할 수 있습니다.
- ✓ 실세계는 모델링 세계로 매핑을 거치고, 모델링 세계는 다시 시스템 세계로 매핑이 됩니다.



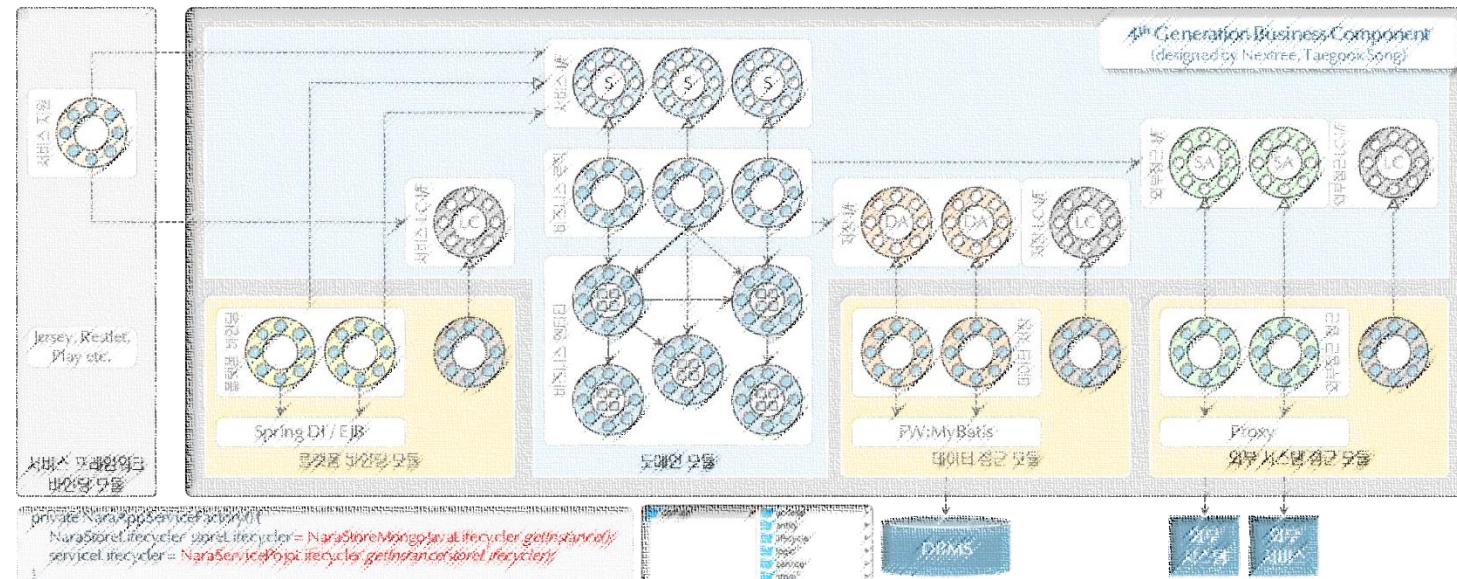
## 언어 매핑(3/3)

- ✓ 시스템 개발 과정에서 뺄 것은 빼고, 정의할 것은 정의하고, 밝힐 것은 밝히는 방식으로 개발을 진행합니다.
- ✓ 프로그래밍의 목표는, 작지만 꼭 필요한 것만 모아 놓은 것 같은 작은 무대를 시스템 안에 만드는 것입니다.
- ✓ 자연어는 풍부한 표현을, 모델링 언어는 다양하지만 간결한 표현을, 프로그래밍 언어는 엄격한 표현을 가능하게 하므로, 우리가 필요한 것을, 구조화하여, 오류 없이 제공할 수 있습니다.



# 요약

- ✓ 현대의 소프트웨어 개발은 객체→컴포넌트→서비스→마이크로서비스로 흘러가고 있습니다.
- ✓ 각 단계 별로 충분히 성숙한 상태에서 다음 기술로 진화 발전할 수 있습니다.
- ✓ 하지만, 우리는 새로운 기술에 관심을 가질 뿐 지난 기술을 과거로 치부해 버리는 경향이 있습니다.
- ✓ 객체 지향 기술은 현대 소프트웨어 기술의 바탕이고 틀이 되는 접근 방법, 사고 방식, 그리고 기술입니다.





## 모듈: 객체 지향 개념

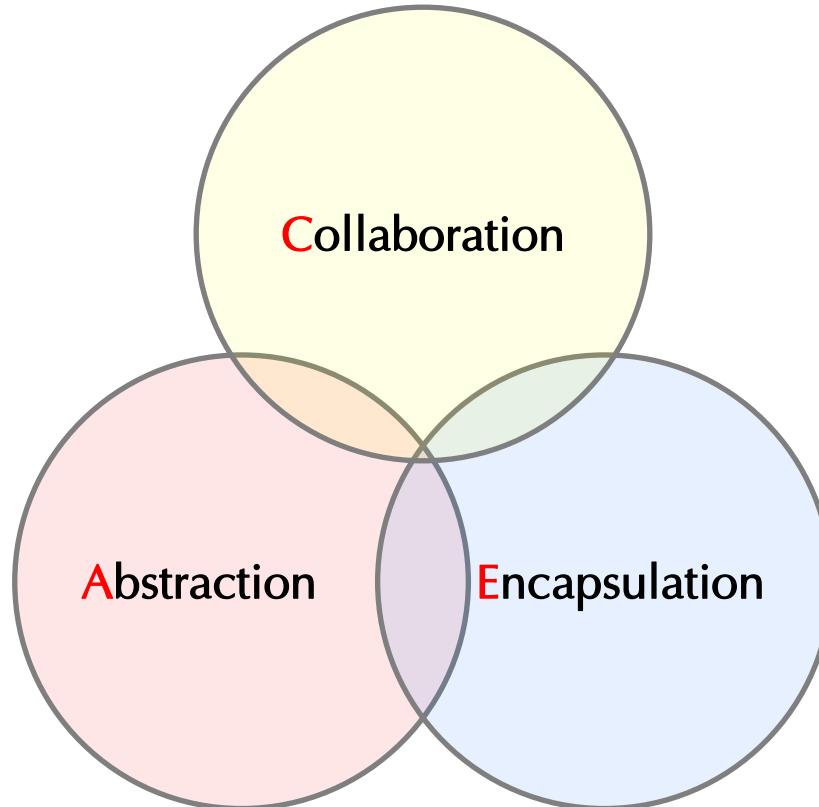
- ❖ 우리가 배워 온 객체지향 개념에 대해 간단하게 요약합니다.
- ❖ 추상화, 캡슐화, 협업 세 가지를 이해합니다.
- ❖ 실세계의 지혜를 얻는 관점에서 실세계 매핑을 이해합니다.

## 종합적인 이해

- ✓ 객체 지향 특성
- ✓ 추상화
- ✓ 캡슐화
- ✓ 협업
- ✓ 요약

# 객체 지향 특성 (1/2) – 프로그래밍

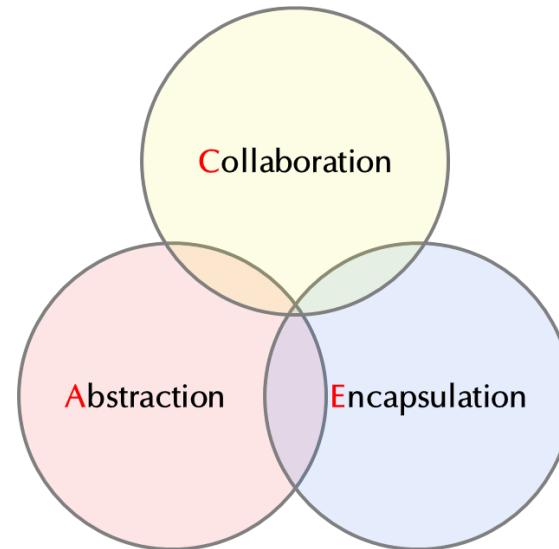
- ✓ 빛의 3원색은 빨강(Red), 녹색(Green), 청색(Blue) 입니다. 줄여서 RGB라고 합니다.
- ✓ 객체 지향 프로그래밍의 3대 요소는 무엇으로 축약할 수 있을까요? 여러 가지 의견들이 많이 있습니다.
- ✓ 추상화(abstraction), 상속(inheritance), 캡슐화(encapsulation), 다형성(polymorphism), ...
- ✓ 우리는 추상화와 캡슐화를 바탕으로 하는 협업(Collaboration) 세 가지를 제시합니다.



## 객체 지향 특성 [2/2] – 모델링

- ✓ 어떤 접근 방법을 사용하든 SW 설계는 낮은 결합도와 높은 응집도를 유지하는 활동이어야 합니다.
- ✓ 원하는 수준의 결합도와 응집도를 유지하기 위해서 관심사를 분리(Separation of Concern)해야 합니다.
- ✓ 객체 지향의 세 가지 특성은 SW 설계 원칙(Loose coupling, high cohesion)을 아주 잘 지원합니다.
- ✓ 뿐만 아니라 강력한 표현력을 제공하여 주기 때문에 복잡한 대상을 단순하게 처리할 수 있습니다.

*Loose coupling, high cohesion !!*



# 추상화(Abstract)

- ✓ 추상화는 공통적인 특징을 찾아 이름과 의미를 부여하는 사고 활동으로 표현력을 극대화할 수 있습니다.
- ✓ 추상화의 힘을 빌리면, 복잡하거나 숫자가 많은 대상을 필요한 만큼 간결하게 표현할 수 있습니다.
- ✓ Abstraction의 범주에는 inheritance, polymorphism, operator overriding 등이 있습니다.
- ✓ 추상화 역량은 인간(human being)을 동물과 구별하는 대표적인 역량입니다.

사육사가 {조류 우리}에 들러서 {새}들의 상태를 확인하다가, 홍학 우리에서 {다리를 다친 홍학}을 발견하고는 수의사에게 치료를 부탁했습니다.

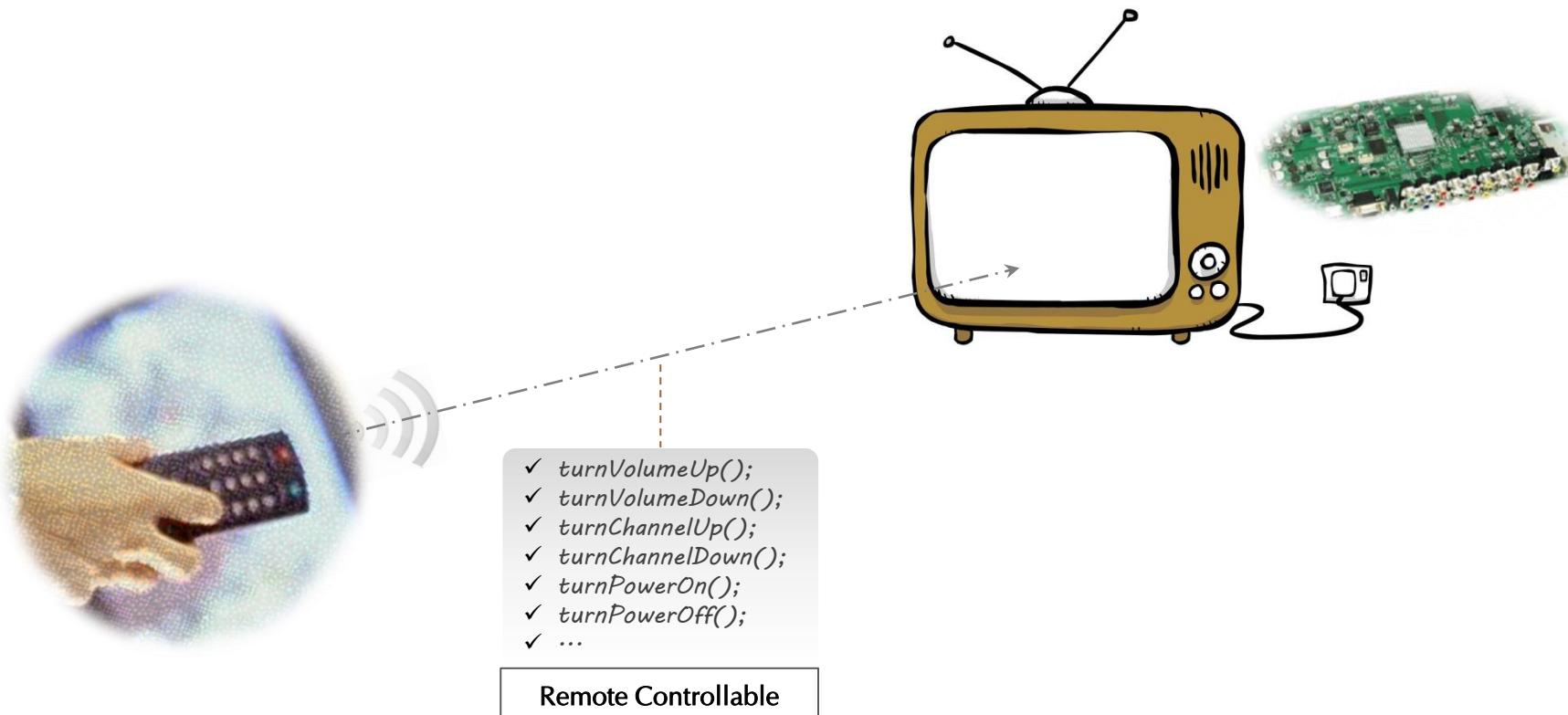


[출처] <http://blog.sangwoodiary.com/entry/20100502-at-the-zoo>

# 캡슐화(Encapsulation)

- ✓ 어떤 기기가 동작하는 방식을 알아야 사용할 수 있다면 TV 조작조차 간단하지 않을 겁니다.
- ✓ 하지만, 우리는 리모컨이라는 작은 기기를 가지고, 소리가 커지는 원리를 몰라도, 쉽게 소리를 키울 수 있습니다.
- ✓ 복잡한 것은 안으로 숨기고, 밖으로는 간단한 인터페이스 만을 내 놓아서, 누구나 쉽게 사용해 주기 때문입니다.
- ✓ 프로그램에서도 복잡한 것은 안쪽에 숨기고(캡슐화를 하고), 밖으로는 간단한 인터페이스를 내 놓아야 합니다.

Abstraction은 효율적인 캡슐화를 하는 방법을 제공합니다. 인터페이스를 실체화(realization)하는 관계는 추상화 계층을 이용하는 방식입니다.



# 협업(Collaboration)

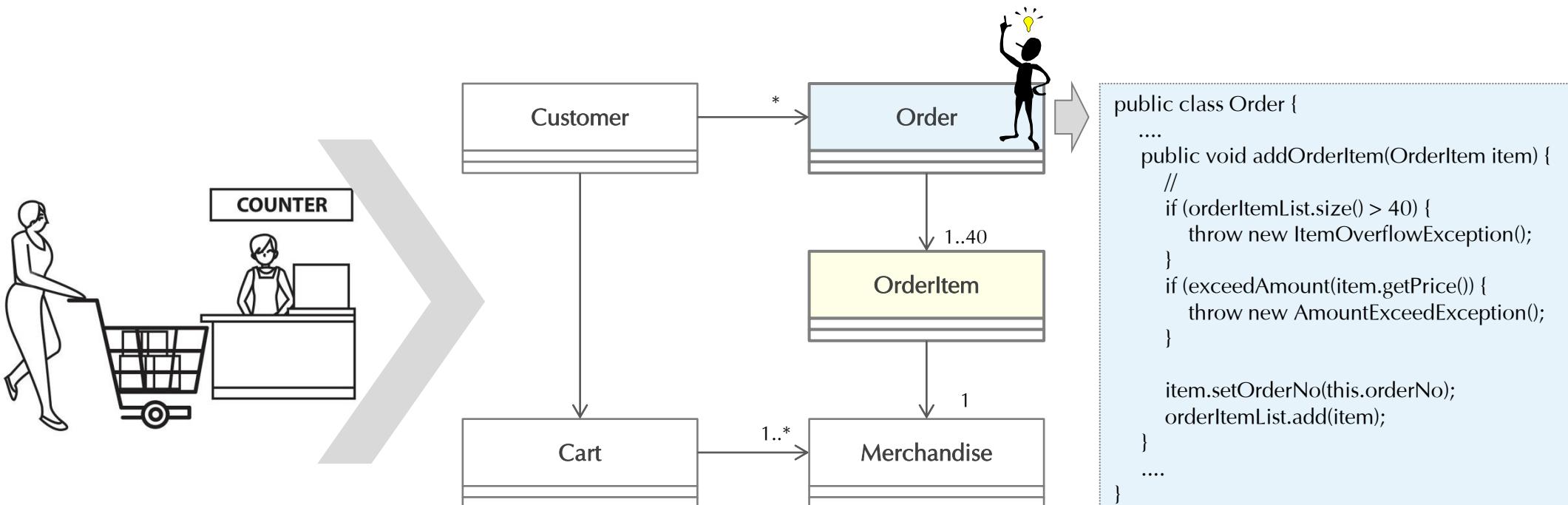
- ✓ 실세계에 존재하는 모든 사람과 동식물의 생활은 모두가 협업의 연속입니다.
- ✓ 오래 세월을 거치면서 협업 방식은 매우 고도화 되었고, 높은 수준의 효율을 보여주고 있습니다.
- ✓ 실세계의 이런 유산을 시스템 세계로 끌고 들어가서 시스템 세계를 건설하는데 사용하여야 합니다.
- ✓ 객체와 개체 간의 협업이라는 관점에서 사고를 하고, 필요하면 실세계에서 지혜를 얻어야 합니다. ← 실세계 매핑



<http://epiccollaboration.com/news-update/mass-collaboration-challenge-integration>

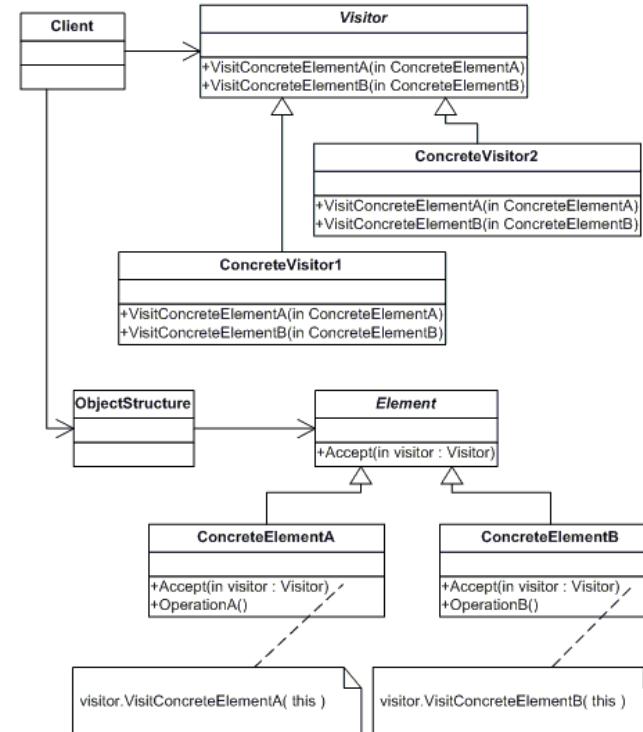
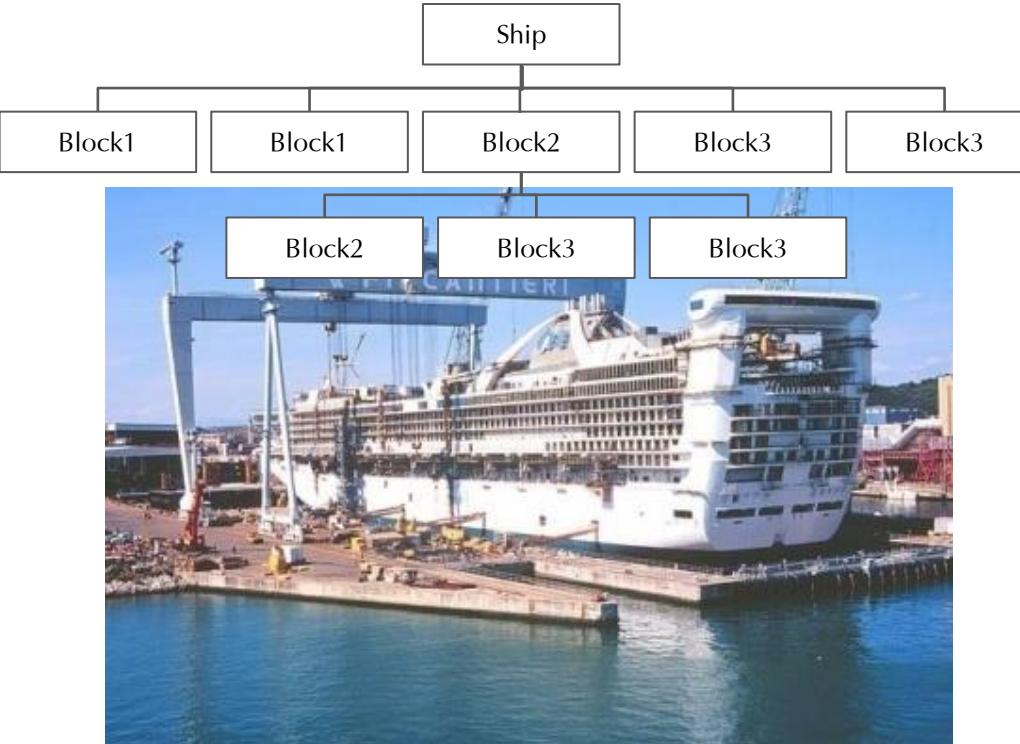
# 협업 – 실세계 매핑 1

- ✓ SW 시스템 세계, 즉 애플리케이션 세계는 실세계의 미니어처입니다.
- ✓ 하지만, 어떤 부분은 실세계의 복잡한 개념을 그대로 옮겨 두어야 합니다. ← 정확한 개념 파악이 필요합니다.
- ✓ 예를 들면, 실제 물품은 여러 개인데 상황에 따라 하나로 또는 여러 개로 다루거나 다른 이름으로 부릅니다.
- ✓ 실세계에 존재하는 개체(entity, object)에 지능을 부여하여 협업하도록 합니다.



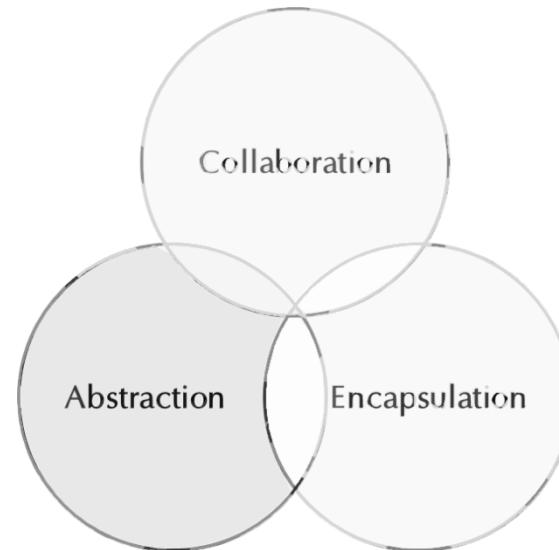
# 협업 – 실세계 매핑 2

- ✓ 때로는 복잡한 문제를 풀어갈 때, 실세계로 역으로 매핑하여 간단하게 풀 수도 있습니다.
- ✓ 시스템 세계의 개념은 대체로 추상적입니다. 객체 지향을 배울 때의 호랑이, 사자는 더 이상 없습니다.
- ✓ 추상화 수준이 높거나 본질적으로 추상적인 대상을 다룰 때, 실세계로의 매핑이 도움이 됩니다.
- ✓ 예를 들어 보겠습니다. Ship Building 도메인에서 BOM(Bill of Materials)을 개발하고 있습니다. 그런데...



# 요약

- ✓ 객체 지향 개념을 종합적으로 이해하고 이를 바탕으로 모델링, 프로그래밍을 하여야 합니다.
- ✓ 추상화는 대상을 단순화하거나, 대상에 대한 초점을 일반적 또는 상세한 곳으로 이동하도록 도와 줍니다.
- ✓ 캡슐화는 복잡한 속사정을 외부로 노출시키지 않음으로써, 외부에서 대상을 단순하게 볼 수 있도록 합니다.
- ✓ 협업은 주어진 시나리오를 수행하는 과정에서 효율적으로 일을 처리하도록 도와 줍니다. 일이 복잡해지더라도 참여하는 객체들은 이전과 같은 수준의 부하를 받도록 설계하여야 합니다. 그러기 위해서는 새로운 객체의 등장과 일 나누기 방식으로 확장을 하여야 합니다.





## 모듈: 객체 지향 설계 원칙

- ❖ Single Responsibility Principle 을 이해합니다.
- ❖ SRP가 프로그램에서 어떻게 사용하는지 실습합니다.
- ❖ 아주 간단한 예제를 통해서 SRP를 이해합니다.

## 단일 책임 원칙(SRP): 구구단

- ✓ 구구단 1
- ✓ 구구단 2 – 책임 분리
- ✓ 구구단 3 – 추가 요구
- ✓ 구구단 4 – 당황스런 요구
- ✓ 구구단 5 – 여러 열로 출력하기(객체지향)
- ✓ 구구단 6 – 정방형으로 출력하기(객체지향)
- ✓ 구구단 7 – 지속적인 확장 실습
- ✓ 요약

# 구구단 1: 얼마나 많은 책임이 있는가?

- ✓ 구구단을 출력하는 프로그램을 작성해 봅니다. 아주 간단하여 하나의 클래스로도 충분합니다.
- ✓ 그런데, `showTable()` 메소드는 여러 가지 일을 하고 있습니다. 즉, 여러 가지 책임을 갖고 있습니다.
- ✓ 이 프로그램은 앞으로 발생 가능한 다양한 변경 요구에 “코드 변경”으로 대응할 수 밖에 없습니다.

Let's print out the multiplication tables.

```
2 times 1 = 2
2 times 2 = 4
2 times 3 = 6
2 times 4 = 8
2 times 5 = 10
2 times 6 = 12
2 times 7 = 14 |
2 times 8 = 16
2 times 9 = 18
```

```
3 times 1 = 3
3 times 2 = 6
3 times 3 = 9
3 times 4 = 12
3 times 5 = 15
3 times 6 = 18
3 times 7 = 21
3 times 8 = 24
3 times 9 = 27
```

```
public void showTable() {
    //
    System.out.println("Let's print out the multiplication tables.");
    System.out.println("-----");

    for (int leftNumber = 2; leftNumber<=9; leftNumber++) {
        for (int rightNumber = 1; rightNumber <=9; rightNumber++) {
            System.out.println(
                String.format(" %2d times %d = %2d ",
                    leftNumber,
                    rightNumber,
                    (leftNumber*rightNumber)));
        }
        System.out.println("-----");
    }
}
```

## Times Table

- + `main(String[])`: void
- + `TimesTable()`
- + `showTable(): void`

## 구구단 2: 책임분리

- ✓ 전체 포맷, 항목 구성, 곱셈, 세 가지 책임으로 분리하여 봅니다.
- ✓ 책임 분리의 첫 번째는 메소드 분리입니다. `showTable()` → `buildTimesItem()`, `multiply()`
- ✓ 변경을 해야 할 때, 세 곳 중에 한 곳만 변경이 되어야 합니다. 동시에 여러 곳을 변경해야 한다면, 책임 분리를 제대로 하지 못한 것입니다.

코딩 실습 !!

```
public void showTable() {
    //
    System.out.println("Let's print out the multiplication tables.");
    System.out.println("-----");

    String itemFormat = " %d times %d = %2d ";

    for (int leftNumber = 2; leftNumber<=9; leftNumber++) {
        for (int rightNumber = 1; rightNumber <=9; rightNumber++) {
            System.out.println(buildTimesItem(itemFormat, leftNumber, rightNumber));
        }
        System.out.println("-----");
    }
}

public String buildTimesItem(String itemFormat, int leftNumber, int rightNumber) {
    //
    return String.format(itemFormat,
        leftNumber,
        rightNumber,
        multiply(leftNumber, rightNumber));
}

public static int multiply(int left, int right) {
    //
    return left * right;
}
```

## 구구단 3:추가 요구 – 여러 열로 출력하기

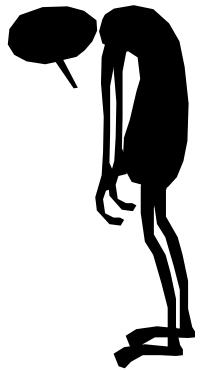
- ✓ 일상에서 흔히 만나는 좌절 상황 !
- ✓ 개발자 입장: 진작 이야기를 해 주었으면 다시 작업하는 일이 없었을 텐데...
- ✓ 고객사 입장: 우린 시장을 읽고 그 변화와 흐름을 비즈니스에 반영할 뿐... (이 정도도 예상 못했나요???)

코딩 실습 !!

```
-----  
2 times 1 = 2  
2 times 2 = 4  
2 times 3 = 6  
2 times 4 = 8  
2 times 5 = 10  
2 times 6 = 12  
2 times 7 = 14 |  
2 times 8 = 16  
2 times 9 = 18  
-----  
3 times 1 = 3  
3 times 2 = 6  
3 times 3 = 9  
3 times 4 = 12  
3 times 5 = 15  
3 times 6 = 18  
3 times 7 = 21  
3 times 8 = 24  
3 times 9 = 27  
-----
```



2 * 1 = 2	3 * 1 = 3	4 * 1 = 4	5 * 1 = 5
2 * 2 = 4	3 * 2 = 6	4 * 2 = 8	5 * 2 = 10
2 * 3 = 6	3 * 3 = 9	4 * 3 = 12	5 * 3 = 15
2 * 4 = 8	3 * 4 = 12	4 * 4 = 16	5 * 4 = 20
2 * 5 = 10	3 * 5 = 15	4 * 5 = 20	5 * 5 = 25
2 * 6 = 12	3 * 6 = 18	4 * 6 = 24	5 * 6 = 30
2 * 7 = 14	3 * 7 = 21	4 * 7 = 28	5 * 7 = 35
2 * 8 = 16	3 * 8 = 24	4 * 8 = 32	5 * 8 = 40
2 * 9 = 18	3 * 9 = 27	4 * 9 = 36	5 * 9 = 45
-----	-----	-----	-----
6 * 1 = 6	7 * 1 = 7	8 * 1 = 8	9 * 1 = 9
6 * 2 = 12	7 * 2 = 14	8 * 2 = 16	9 * 2 = 18
6 * 3 = 18	7 * 3 = 21	8 * 3 = 24	9 * 3 = 27
6 * 4 = 24	7 * 4 = 28	8 * 4 = 32	9 * 4 = 36
6 * 5 = 30	7 * 5 = 35	8 * 5 = 40	9 * 5 = 45
6 * 6 = 36	7 * 6 = 42	8 * 6 = 48	9 * 6 = 54
6 * 7 = 42	7 * 7 = 49	8 * 7 = 56	9 * 7 = 63
6 * 8 = 48	7 * 8 = 56	8 * 8 = 64	9 * 8 = 72
6 * 9 = 54	7 * 9 = 63	8 * 9 = 72	9 * 9 = 81



## 구구단 3: 추가 요구 – 절차 지향적 해결

✓ 잠시 좌절 후, 이런 것쯤이야 쉽게 해결할 수 있겠지요. 그런데 제약 조건이 좀 고약합니다.

```
public void showTable(int columnCount) {
    //
    if (columnCount > MaxTimes) {
        throw new RuntimeException(
            String.format("The column count should be less than %d, but it's %d. --> ", MaxTimes, columnCount));
    }

    int leftNumber = 2;
    while(true) {
        for (int rightNumber = 1; rightNumber <=MaxTimes; rightNumber++) {
            for(int offset=0; offset<columnCount; offset++) {
                if (leftNumber+offset > MaxTimes) {
                    break;
                }
                System.out.print(buildTimesItem(leftNumber+offset, rightNumber));
                System.out.print("  ");
            }
            System.out.println("  ");
        }
        System.out.println("  ");

        leftNumber += columnCount;
        if (leftNumber > MaxTimes) {
            break;
        }
    }
}

public String buildTimesItem(int leftNumber, int rightNumber) {
    //
    String itemFormat = this.getItemFormatStr(leftNumber);

    return String.format(itemFormat,
        leftNumber,
        rightNumber,
        multiply(leftNumber, rightNumber));
}
```



## 구구단 3: 추가 요구 – 절차 지향적 해결과 문제점

- ✓ 문제점1 : Complexity 증가 → 7, 116라인으로 증가
- ✓ 더 큰 문제점 2: 보여 주기에 집중하면서, 컨텍스트 속의 구구단의 개념이 서서히 사라지고 있습니다.
- ✓ 즉, 프로그램은 구구단의 의미를 떠나서 숫자를 여러 컬럼으로 보여 주는 작업에 집중하고 있습니다.
- ✓ 컨텍스트가 프로그램 코드의 흐름과 판단 속으로 매몰되어 가고 있습니다.

```
public void showTable(int columnCount) {  
    //  
    if (columnCount > MaxTimes) {  
        throw new RuntimeException(  
            String.format("The column count should be less than %d, but it's %d. --> ", MaxTimes, columnCount));  
    }  
  
    int leftNumber = 2;  
    while(true) {  
        for (int rightNumber = 1; rightNumber <=MaxTimes; rightNumber++) {  
            for(int offset=0; offset<columnCount; offset++) {  
                if (leftNumber+offset > MaxTimes) {  
                    break;  
                }  
                System.out.print(buildTimesItem(leftNumber+offset, rightNumber));  
                System.out.print(" ");  
            }  
            System.out.println(" ");  
        }  
        System.out.println(" ");  
  
        leftNumber += columnCount;  
        if (leftNumber > MaxTimes) {  
            break;  
        }  
    }  
}  
  
public String buildTimesItem(int leftNumber, int rightNumber) {  
    //  
    String itemFormat = this.getItemFormatStr(leftNumber);  
  
    return String.format(itemFormat,  
        leftNumber,  
        rightNumber,  
        multiply(leftNumber, rightNumber));  
}
```



## 구구단 4:당황스런 요구 – 정방형으로 출력하기

- ✓ 여러 열로 출력하는 구구단을 출력하고 나니 다음과 같은 추가 요구 사항이 기다리고 있습니다.
- ✓ 구구단을 숫자 간의 관계로 파악하기 좋은 형태로 표현하는 것입니다.
- ✓ 믿기 어렵겠지만, 이것도 구구단의 한 형태이므로, 개발자에게 추가되는 요구 사항은 타당합니다.
- ✓ 구구단 개념을 바탕으로 보면 유사한 요구이지만, 표현 관점에서 보면 서로 다른 요구사항입니다.

코딩 실습 !!

9  
8 18 27 36 45 54 63 72 81  
7 16 24 32 40 48 56 64 72  
6 14 21 28 35 42 49 56 63  
5 12 18 24 30 36 42 48 54  
4 10 15 20 25 30 35 40 45  
3 8 15 20 25 30 35 40 45  
2 6 12 16 20 24 28 32 36  
1 4 9 12 15 18 21 24 27  
  6 8 10 12 14 16 18

1 2 3 4 5 6 7 8 9  
2 4 6 8 10 12 14 16 18  
3 6 9 12 15 18 21 24 27  
4 8 12 16 20 24 28 32 36  
5 10 15 20 25 30 35 40 45  
6 12 18 24 30 36 42 48 54  
7 14 21 28 35 42 49 56 63  
8 16 24 32 40 48 56 64 72  
9 18 27 36 45 54 63 72 81



이것도 구구단 ???

## 구구단 4:당황스런 요구- 절차 지향적 해결과 문제점

- ✓ 잠시 고민 후에 별도의 메소드로 구현합니다.
- ✓ 문제점1 : Complexity 증가 → 12,190라인으로 증가
- ✓ 더 큰 문제점 2: `showLineTable()` 메소드와 `showBoxTable()` 메소드는 서로 연관성이 없는 다른 메소드로 써 모두 보여 주기에 집중하고 있습니다.
- ✓ 이 코드에서 구구단 개념은 더 이상 존재하지 않고 화면에 나타난 결과 값으로만 존재하고 있습니다.

```
public void showBoxTable(SortOrder sortOrder) {  
    //  
    int leftNumber = 2;  
  
    if(SortOrder.Descending.equals(sortOrder)) {  
        leftNumber = MaxTimes;  
    }  
  
    System.out.println();  
  
    while(true) {  
        for(int rightNumber = 1; rightNumber <= MaxTimes; rightNumber++) {  
            for(int offset=0; offset<columnCount; offset++) {  
                int newLeftNumber = leftNumber + offset;  
                if(SortOrder.Descending.equals(sortOrder)) {  
                    newLeftNumber = leftNumber - offset;  
                }  
                if (newLeftNumber < 2 || newLeftNumber > MaxTimes) {  
                    break;  
                }  
                System.out.print(buildTimesItem(newLeftNumber, rightNumber));  
                System.out.print(" ");  
            }  
            System.out.println();  
        }  
        System.out.println();  
    }  
}  
  
public void showLineTable(SortOrder sortOrder) {  
    //  
    int leftNumber = 2;  
    int addNumber = 1;  
  
    if(SortOrder.Descending.equals(sortOrder)) {  
        leftNumber = MaxTimes;  
        addNumber = -1;  
    }  
  
    System.out.println();  
  
    while(true) {  
        for(int rightNumber = 1; rightNumber <= MaxTimes; rightNumber++) {  
            System.out.print(getColumn(leftNumber, rightNumber));  
        }  
    }  
}
```

## 구구단 5: 단일 책임의 원칙(SRP)

- ✓ 절차 지향 문제 해결 방법에서 여러분이 개발한 메소드를 자세히 들여다 보세요.
- ✓ 계산, 수식 표현, 정렬 등 여러 가지 일을 동시에 하고 있음을 알 수 있습니다.
- ✓ 이러한 문제 해결 방식은 또 다른 변경 요구가 계산, 수식, 정렬 등 어느 곳에서 오던지 많은 변경이 필요합니다.
- ✓ 한 가지 일을 잘 하는 여러 전문가 객체를 이용하여 문제를 풀어 가야 합니다. → SRP

한 클래스는 하나 그리고 단 하나의 변경 이유를 가져야 한다. 이것은 클래스는 오직 하나의 일만을 가져야 함을 의미한다.

A class should have one and only one reason to change, meaning that a class should have only one job.

# 구구단 5: 여러 열로 출력하기(객체 지향)

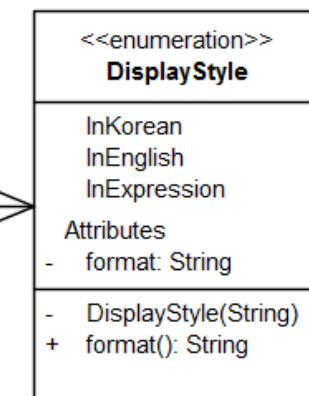
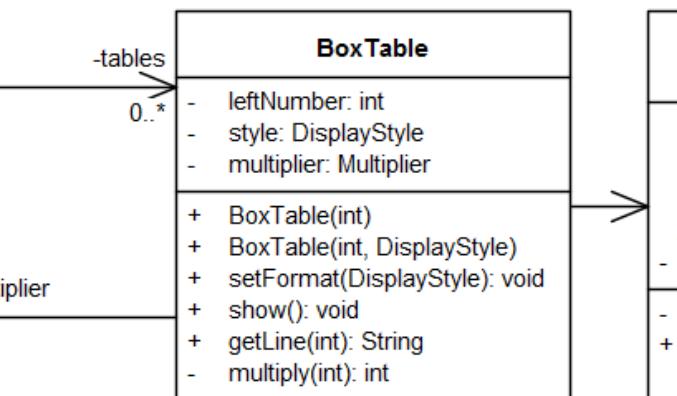
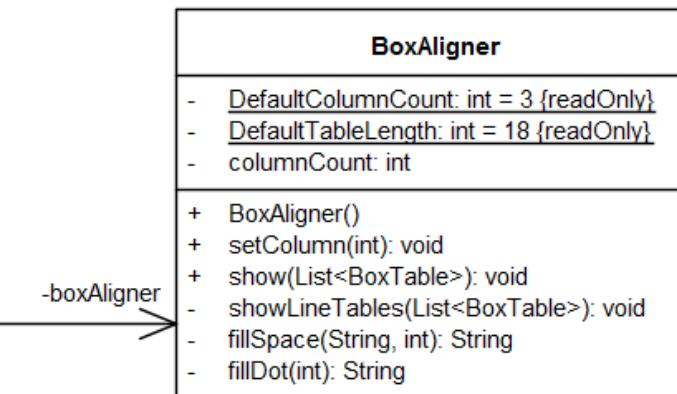
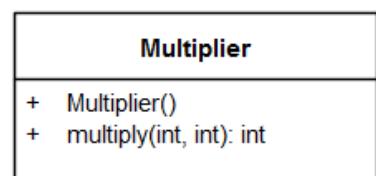
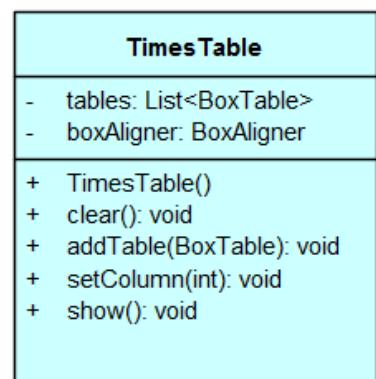
- ✓ 개념을 찾고 책임을 찾아 봅니다. → 각 단 구성, 각 단의 스타일 적용, 계산, 전체 단 구성 조정 등
- ✓ 이제 TimesTable은 각 역할 담당 객체를 거느리고 구구단 출력을 합니다.
- ✓ 여기까지 왔으면, 사용자의 요구 사항이 어디로 뛰어 갈 수 있는지 대략 예측을 할 수 있어야 합니다.

```

2 times 1 = 2      3 * 1 = 3      4 * 1 = 4
2 times 2 = 4      3 * 2 = 6      4 * 2 = 8
2 times 3 = 6      3 * 3 = 9      4 * 3 = 12
2 times 4 = 8      3 * 4 = 12     4 * 4 = 16
2 times 5 = 10     3 * 5 = 15     4 * 5 = 20
2 times 6 = 12     3 * 6 = 18     4 * 6 = 24
2 times 7 = 14     3 * 7 = 21     4 * 7 = 28
2 times 8 = 16     3 * 8 = 24     4 * 8 = 32
2 times 9 = 18     3 * 9 = 27     4 * 9 = 36

3 times 1 = 3      6 * 1 = 6      7 * 1 = 7      8 * 1 = 8
3 times 2 = 6      6 * 2 = 12     7 * 2 = 14     8 * 2 = 16
3 times 3 = 9      6 * 3 = 18     7 * 3 = 21     8 * 3 = 24
3 times 4 = 12     6 * 4 = 24     7 * 4 = 28     8 * 4 = 32
3 times 5 = 15     6 * 5 = 30     7 * 5 = 35     8 * 5 = 40
3 times 6 = 18     6 * 6 = 36     7 * 6 = 42     8 * 6 = 48
3 times 7 = 21     6 * 7 = 42     7 * 7 = 49     8 * 7 = 56
3 times 8 = 24     6 * 8 = 48     7 * 8 = 56     8 * 8 = 64
3 times 9 = 27     6 * 9 = 54     7 * 9 = 63     8 * 9 = 72

```



## 구구단 5: 여러 열로 출력하기(객체 지향)

- ✓ 이제 BoxTable, BoxAligner 등 객체의 책임을 코드로 구현합니다.
- ✓ 코딩을 하는 순간에는 해당 객체의 역할과 책임에만 집중할 수 있으며, 그 과정에서 전체 그림에서 발견하지 못 한 작지만 중요한 기능들을 발견할 수도 있습니다.

```
public BoxTable(int leftNumber, DisplayStyle style) {  
    //  
    this(leftNumber);  
    setFormat(style);  
}  
  
public void setFormat(DisplayStyle style) {  
    //  
    this.style = style;  
}  
  
public void show() {  
    //  
    for (int right=0; right<=9; right++) {  
        System.out.println(getFormattedLine(right));  
    }  
}  
  
public String getFormattedLine(int rightNumber) {  
    //  
    return String.format(style.format(), leftNumber, rightNumber, multiply(rightNumber));  
}  
  
private int multiply(int rightNumber) {  
    //  
    return multiplier.multiply(leftNumber, rightNumber);  
}
```

```
public void show(List<BoxTable> tables) {  
    //  
    List<BoxTable> lineTables = new ArrayList<BoxTable>();  
  
    for (BoxTable table : tables) {  
        lineTables.add(table);  
        if (lineTables.size() == columnCount) {  
            showLineTables(lineTables);  
            lineTables.clear();  
        }  
    }  
  
    if (lineTables.size() > 0) {  
        showLineTables(lineTables);  
    }  
}  
  
private void showLineTables(List<BoxTable> tables) {  
    //  
    int tableSize = tables.size();  
    int lineLength = 0;  
  
    for (int right = 1; right<=9; right++) {  
        StringBuilder builder = new StringBuilder();  
        for (int i=0; i            String line = tables.get(i).getFormattedLine(right);  
            builder.append(fillSpace(line, DefaultTableLength));  
        }  
        lineLength = builder.length();  
        System.out.println(builder.toString());  
    }  
  
    System.out.println(fillDot(lineLength));  
}
```

## 구구단 6:정방형으로 출력하기[객체지향]

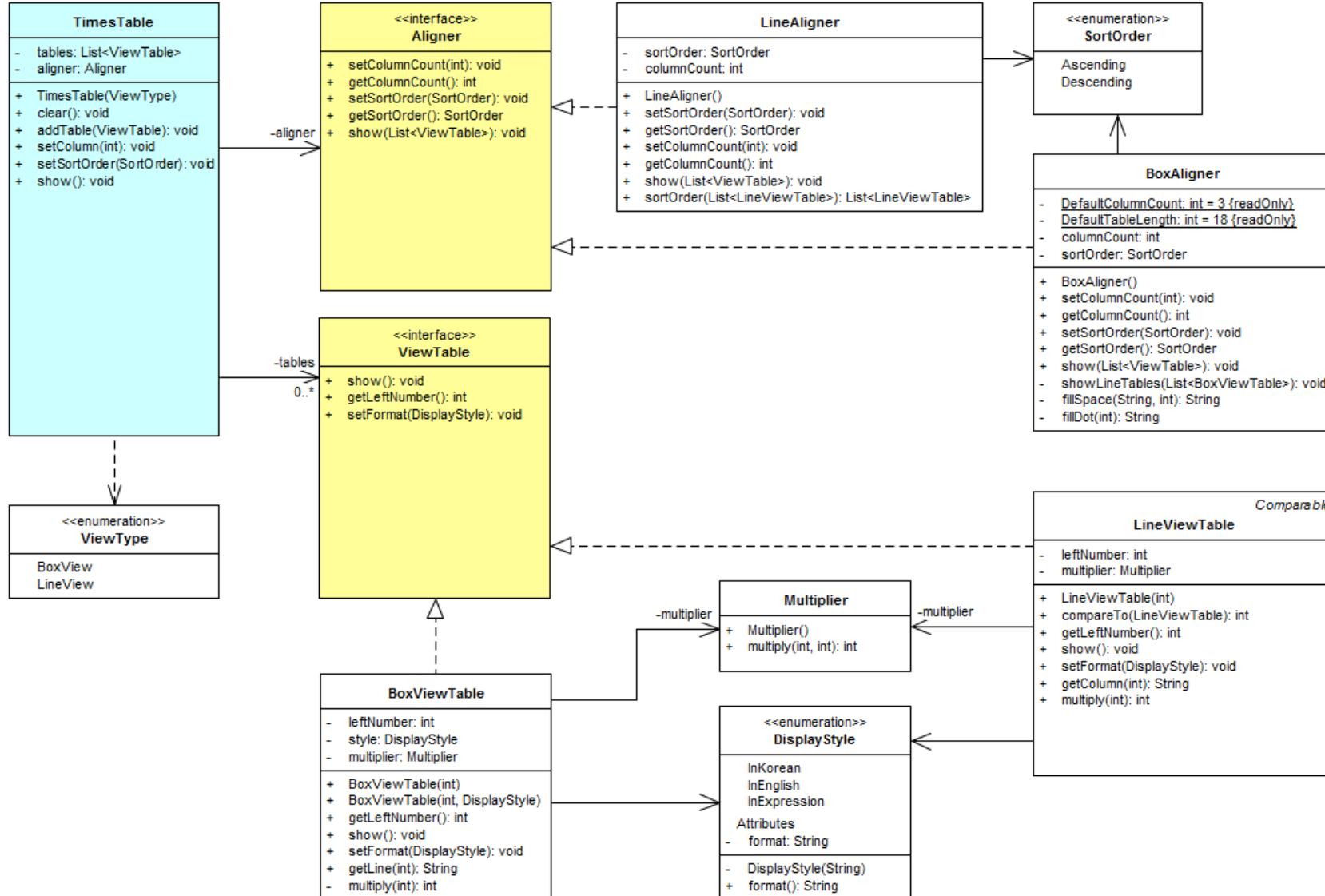
- ✓ 구구단의 본질이 무엇인가를 생각해 보면, 정방형으로 표현하는 것도 충분히 의미가 있어 보입니다.
- ✓ 이러한 표현 방식은 구구단에서 leftNumber와 rightNumber의 관계를 잘 보여줍니다.
- ✓ 예상치 못한 표현 방식이지만, 구구단의 본질이라는 관점에서 충분히 가능한 표현 방법입니다.
- ✓ 앞에서 고민하여 개념의 관계 구조를 만들었는데, 이제 확장을 위한 준비를 할 차례입니다.

코딩 실습 !!



# 구구단 6: 정방형으로 출력하기(객체지향)

✓ 이제 TimesTable은 다양한 Aligner와 ViewTable을 수용할 준비가 되어 있습니다.



## 구구단 6:정방형으로 출력하기[객체지향]

- ✓ 이제 TimesTable 클래스를 이용하여 다양한 표현을 간단하게 할 수 있습니다. 개발자와 TimesTable 간의 협업을 상상할 수 있습니다.

```
public static void main(String[] args) {
    //
    showLineTableAscendingDemo();
    showTimesTableObjectDemo();
    showLineTableDescendingDemo();
    showTimesTableProcedureDemo();
}

private static void showLineTableAscendingDemo() {
    //
    TimesTable timesTable = new TimesTable(ViewType.LineView);
    timesTable.createDefaultViewTables();
    timesTable.setSortOrder(SortOrder.Ascending);
    timesTable.show();
}

private static void showTimesTableObjectDemo() {
    //
    TimesTable timesTable = new TimesTable(ViewType.BoxView);

    // User defined tables
    timesTable.addTable(new BoxViewTable(3, DisplayStyle.InKorean));
    timesTable.addTable(new BoxViewTable(5, DisplayStyle.InEnglish));
    timesTable.addTable(new BoxViewTable(7, DisplayStyle.InExpression));
    timesTable.addTable(new BoxViewTable(9));

    timesTable.setColumn(3);
    timesTable.show();
    timesTable.clear();

    // Default times tables
    timesTable.createDefaultViewTables();
    timesTable.setColumn(5);
    timesTable.show();
}
```

# 구구단 7: 지속적인 확장 실습

- ✓ 이런 구구단은 어떨까요?
- ✓ 구구단을 처음 외우는 학생이나, 외국 인들에게는 유용하겠죠?
- ✓ 앞의 구구단 애플리케이션을 바탕으로 확장하면 됩니다.
- ✓ 새로운 객체가 등장할 수도 있고, 기존의 객체를 확장할 수 있습니다.
- ✓ 이제는 기존 코드 변경은 거의 없이, 확장으로 문제를 풀 수 있습니다.
- ✓ 여러분의 선택은?

$2 \times 1 = 2$	이 일은 이	$3 \times 1 = 3$	삼 일은 삼
$2 \times 2 = 4$	이 이는 사	$3 \times 2 = 6$	삼 이는 육
$2 \times 3 = 6$	이 삼은 육	$3 \times 3 = 9$	삼 삼은 구
$2 \times 4 = 8$	이 사는 팔	$3 \times 4 = 12$	삼 사 십이
$2 \times 5 = 10$	이 오는 십	$3 \times 5 = 15$	삼 오 십오
$2 \times 6 = 12$	이 육 십이	$3 \times 6 = 18$	삼 육 십팔
$2 \times 7 = 14$	이 칠은 십사	$3 \times 7 = 21$	삼 칠 이십일
$2 \times 8 = 16$	이 팔 십육	$3 \times 8 = 24$	삼 팔 이십사
$2 \times 9 = 18$	이 구는 십팔	$3 \times 9 = 27$	삼 구 이십칠

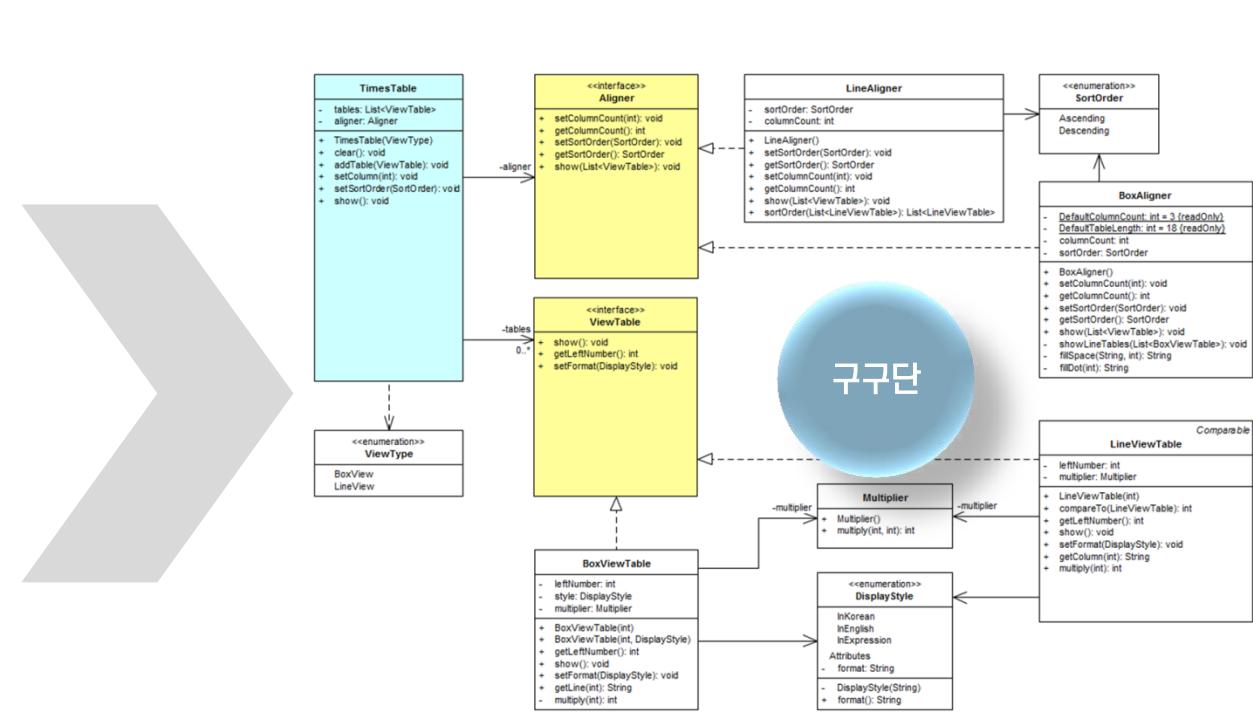
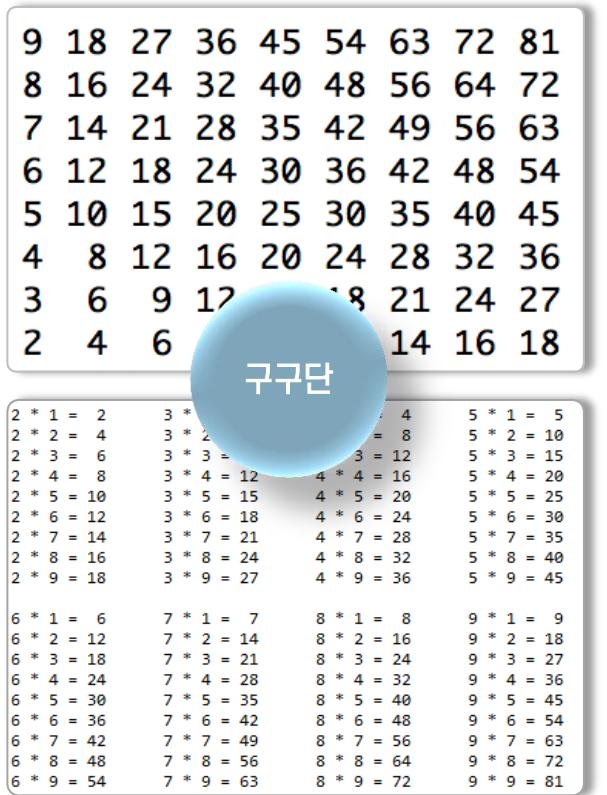
$4 \times 1 = 4$	사 일은 사	$5 \times 1 = 5$	오 일은 오
$4 \times 2 = 8$	사 이는 팔	$5 \times 2 = 10$	오 이는 십
$4 \times 3 = 12$	사 삼 십이	$5 \times 3 = 15$	오 삼 십오
$4 \times 4 = 16$	사 사 십육	$5 \times 4 = 20$	오 사 이십
$4 \times 5 = 20$	사 오 이십	$5 \times 5 = 25$	오 오 이십오
$4 \times 6 = 24$	사 육 이십사	$5 \times 6 = 30$	오 육 삼십
$4 \times 7 = 28$	사 칠 이십팔	$5 \times 7 = 35$	오 칠 삼십오
$4 \times 8 = 32$	사 팔 삼십이	$5 \times 8 = 40$	오 팔 사십
$4 \times 9 = 36$	사 구 삼십육	$5 \times 9 = 45$	오 구 사십오

$6 \times 1 = 6$	육 일은 육	$7 \times 1 = 7$	칠 일은 칠
$6 \times 2 = 12$	육 이 십이	$7 \times 2 = 14$	칠 이 십사
$6 \times 3 = 18$	육 삼 십팔	$7 \times 3 = 21$	칠 삼 이십일
$6 \times 4 = 24$	육 사 이십사	$7 \times 4 = 28$	칠 사 이십팔
$6 \times 5 = 30$	육 오 삼십	$7 \times 5 = 35$	칠 오 삼십오
$6 \times 6 = 36$	육 육 삼십육	$7 \times 6 = 42$	칠 육 사십이
$6 \times 7 = 42$	육 칠 사십이	$7 \times 7 = 49$	칠 칠 사십구
$6 \times 8 = 48$	육 팔 사십팔	$7 \times 8 = 56$	칠 팔 오십육
$6 \times 9 = 54$	육 구 오십사	$7 \times 9 = 63$	칠 구 육십삼

$8 \times 1 = 8$	팔 일은 팔	$9 \times 1 = 9$	구 일은 구
$8 \times 2 = 16$	팔 이 십육	$9 \times 2 = 18$	구 이 십팔
$8 \times 3 = 24$	팔 삼 이십사	$9 \times 3 = 27$	구 삼 이십칠
$8 \times 4 = 32$	팔 사 삼십이	$9 \times 4 = 36$	구 사 삼십육
$8 \times 5 = 40$	팔 오 사십	$9 \times 5 = 45$	구 오 사십오
$8 \times 6 = 48$	팔 육 사십팔	$9 \times 6 = 54$	구 육 오십사
$8 \times 7 = 56$	팔 칠 오십육	$9 \times 7 = 63$	구 칠 육십사
$8 \times 8 = 64$	팔 팔 육십사	$9 \times 8 = 72$	구 팔 칠십이
$8 \times 9 = 72$	팔 구 칠십이	$9 \times 9 = 81$	구 구 팔십일

# 요약

- ✓ 구구단의 의미를 유지하면서 변경 요구에 대응하는 것이 중요합니다. 언어 변환이지 개념 변환이 아닙니다.
- ✓ 추가 요구사항을 “변경”이 아닌 “확장”으로 대응할 수 있다면, 그것은 훌륭한 객체지향 프로그램입니다.
- ✓ 구구단에서 모델은 의미가 매우 약하여 BoxViewTable이나 LineViewTable과 결합한 상태입니다. 이것을 분리하여 개발한다면 어떤 모습일까요?





## 모듈: 객체 지향 설계 원칙

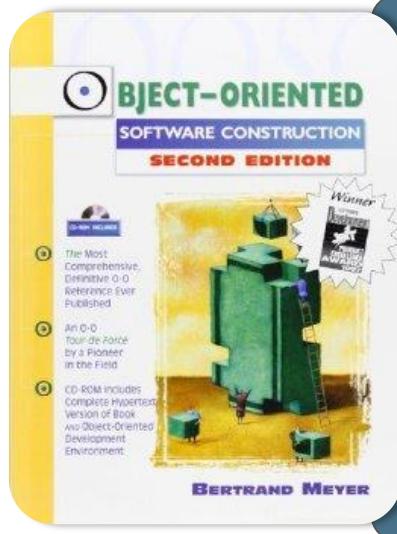
- ❖ Open Closed Principle 을 이해합니다.
- ❖ OCP가 프로그램에서 어떻게 사용하는지 실습합니다.
- ❖ 아주 간단한 예제를 통해서 OCP를 이해합니다.

## 개방 폐쇄 원칙(OCP): 다중 매핑

- ✓ 개념
- ✓ 변경 요구
- ✓ 현재 구조
- ✓ Injection
- ✓ 확장
- ✓ 요약

# OCP: 개념(1/2)

- ✓ 소프트웨어는 태생적으로 성장하는(growing) 존재입니다. 그래서 버전 개념을 내포하고 있습니다.
- ✓ 1988년 Bertrand Meyer는 자신의 책 “Object-Oriented Software Construction”에서 변경에 대응 할 수 있는 설계 가이드를 제시했습니다.
- ✓ “Open for extension, but closed for modification” 개념이 SOLID의 두 번째 원칙 OCP입니다.



소프트웨어 요소(클래스, 모듈, 함수, 등)는 확장에 대해 열려 있어야 하고, 변경에 대해서는 닫혀 있어야 한다.

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

## OCP: 개념(2/2)

- ✓ 변경 할 수 있어야 하는데, 소스 코드나 라이브러리는 변경하지 말라는 의미는 매우 난해할 수 있습니다.
- ✓ Meyer는 이런 원칙을 가능하도록 하려면 “상속” 개념이 필요하다고 했습니다.
- ✓ OCP의 키워드는 바로 “추상화(abstraction)”이며, 인터페이스 기반 설계가 필요합니다.
- ✓ 추상 클래스를 통한 확장 보다는 인터페이스를 통한 확장이 바람직합니다.

### [OPEN] 확장에 대해 열려 있고...

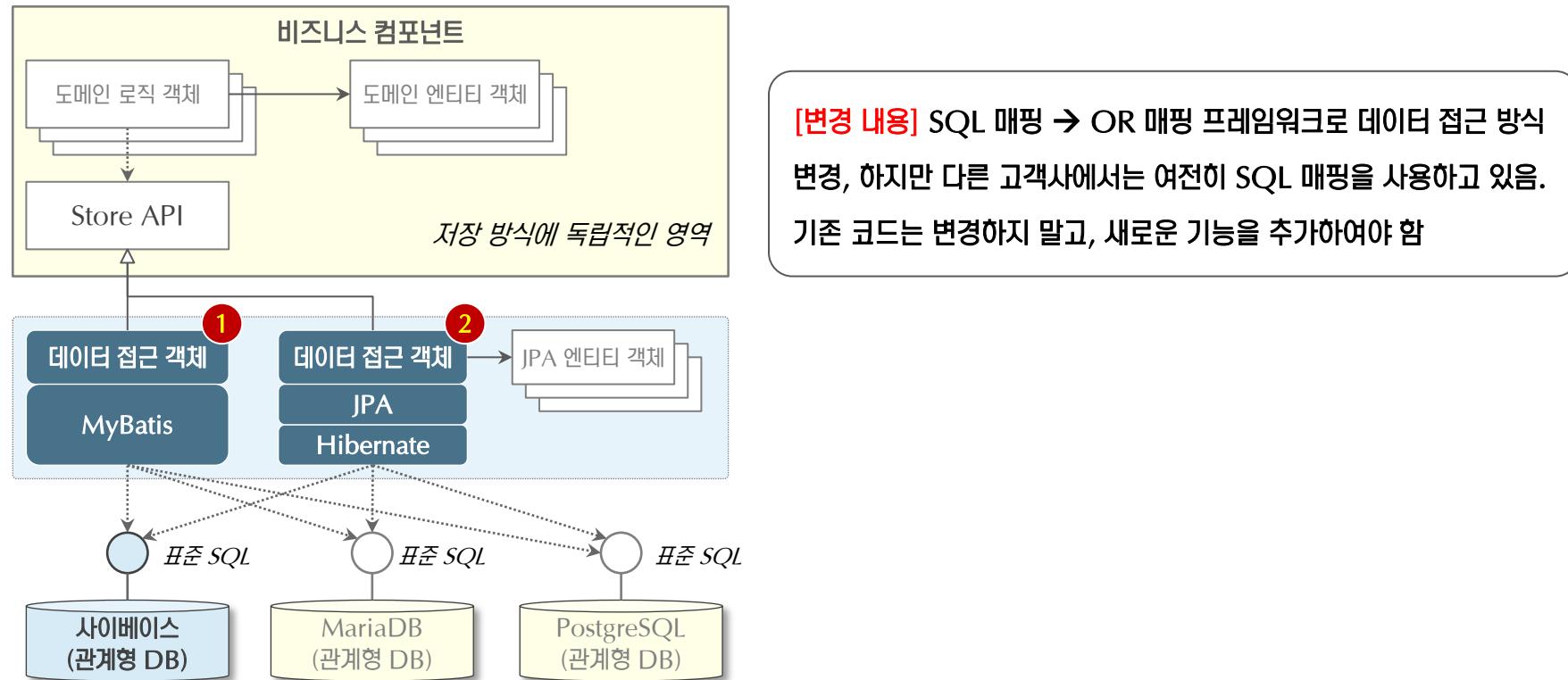
모듈의 기능은 확장 가능해야 한다. 새로운 변경 요구를 충족하기 위해서 모듈에 새로운 기능을 추가할 수 있어야 한다.

### [CLOSE] 변경에 대해 닫혀 있어야...

모듈의 기능을 확장하는 과정에서 기존 기능을 잘 수행하고 있는 모듈의 소스 코드나 라이브러리를 변경해서는 안된다.

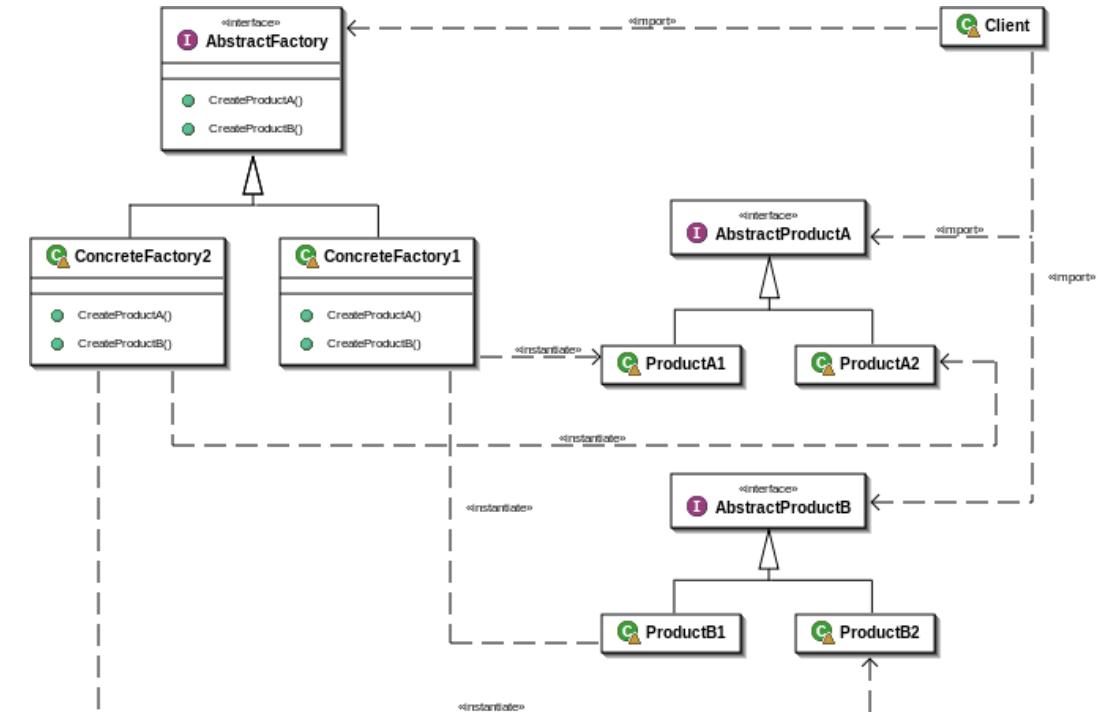
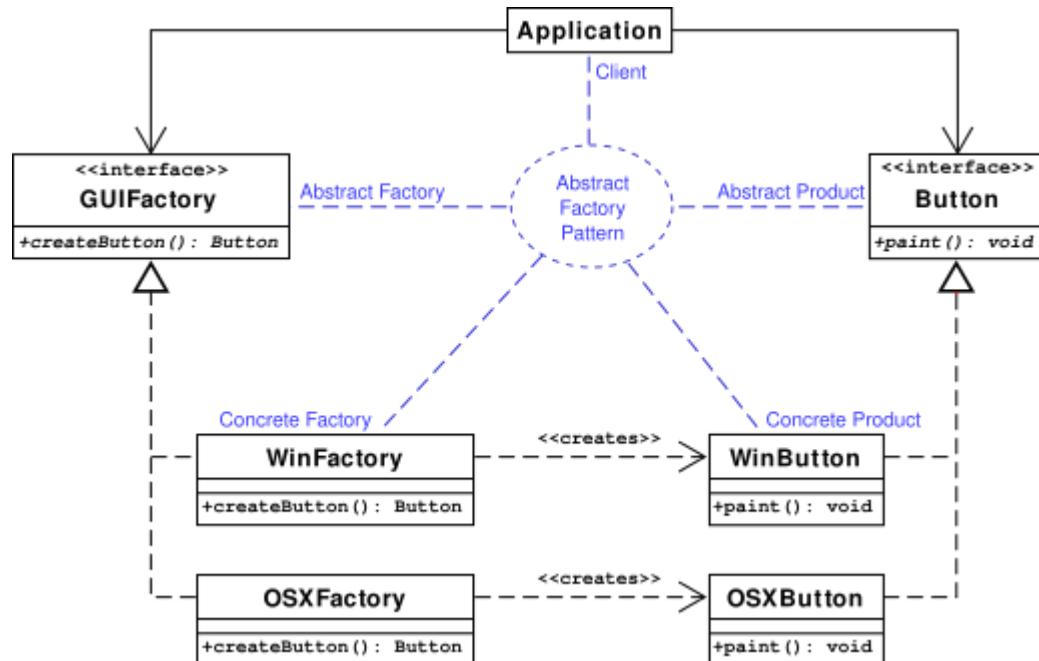
# OCP: 변경 요구

- ✓ “나무소리” 사는 소셜보드 솔루션을 만들어서 판매하고 있었습니다.
- ✓ 솔루션은 고객의 요구에 맞추어서, 상당한 부분에 대해 customization 작업을 수행합니다.
- ✓ 최근에 작업을 시작한 고객사에서는 데이터 접근 방식에 대한 변경을 요구했습니다.
- ✓ 나무소리 게시판 솔루션은 SQL 매팅 프레임워크인 MyBatis를 기본 프레임워크로 사용하였습니다.



# 0CP: 변경 요구 – 디자인 패턴

- ✓ 데이터 접근 인터페이스는 여러 개 (product) 입니다. → BoardStore, PostStore
- ✓ 여러 개의 (product)은 한꺼번에 세트로 생성하여야 합니다. ← AbstractFactory
- ✓ 개별 변경을 막고, 세트로 확장을 하도록 설계할 때 도움이 되는 패턴입니다.

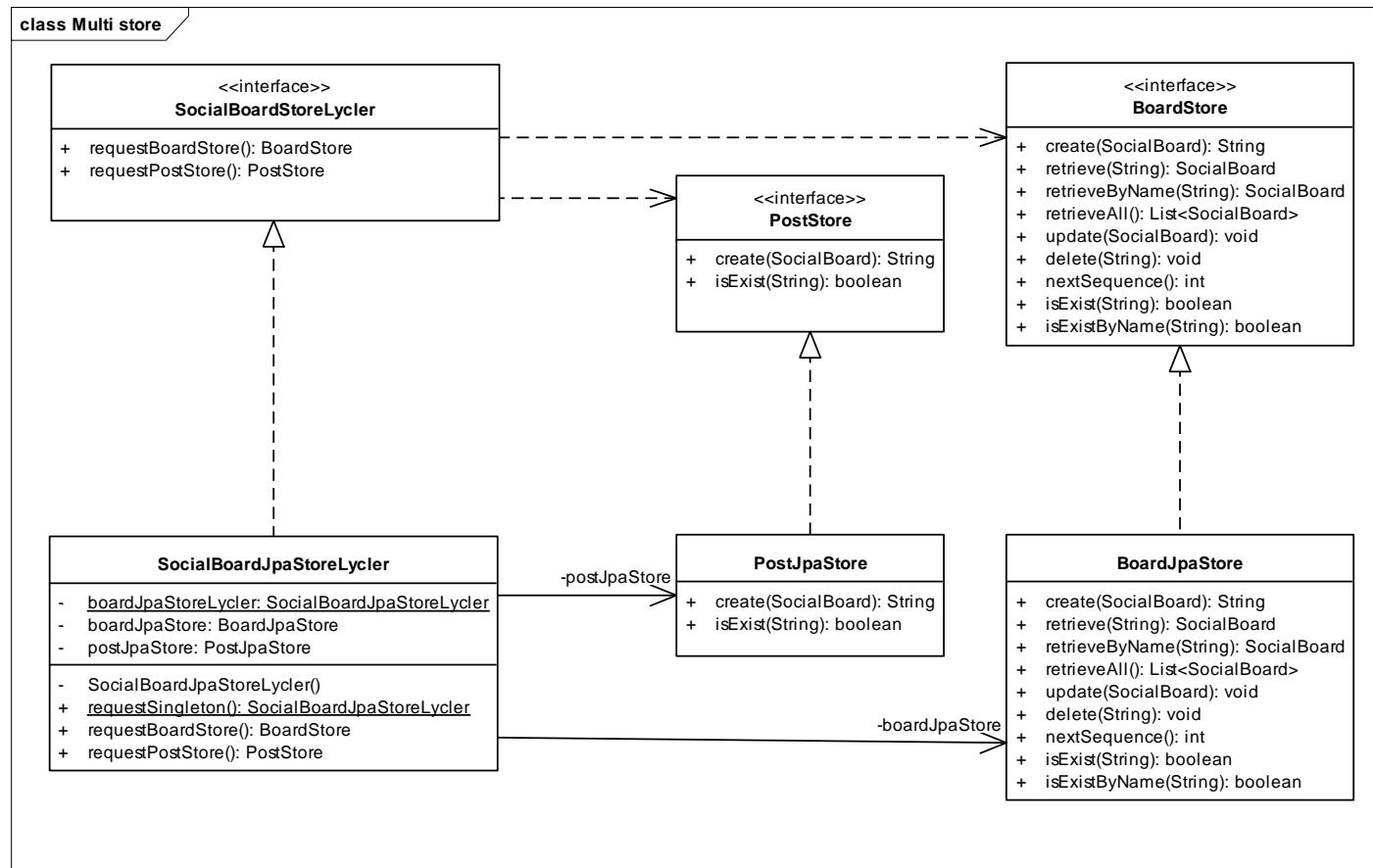


[출처: 위키백과] [https://en.wikipedia.org/wiki/Abstract\\_factory\\_pattern](https://en.wikipedia.org/wiki/Abstract_factory_pattern)

# OCP: 현재 구조

- ✓ 현재 구조는 SocialBoardStoreLypler와 BoardStore 인터페이스를 기반으로 설계하였습니다.
- ✓ 언제든지 해당 인터페이스를 구현함으로써 확장할 수 있습니다.
- ✓ SocialBoardServiceLogic은 인터페이스 기반으로 구현함으로써 변경으로 부터 자유로울 수 있습니다.

코딩 실습 !!



# OCP: Injection (1/2)

- ✓ 생성자에 StoreLypler를 주입하고, 해당 Store 객체는 Lypler로부터 얻어 오는 구조로 개발합니다.
- ✓ 프로그램에서는 boardStore 인터페이스를 이용하여 개발을 합니다.
- ✓ 어떤 Lypler(LifeCycler의 줄임말)가 주입되는 프로그램은 관계없이 동작합니다.

```
public class SocialBoardServiceLogic implements SocialBoardService {
    //
    private BoardStore boardStore;
    private PostStore postStore;

    public SocialBoardServiceLogic(StoreLypler storeLypler) {
        //
        this.boardStore = storeLypler.requestBoardStore();
        this.postStore = storeLypler.requestPostStore();
    }

    @Override
    public String registerSocialBoard(String boardName, boolean commentable) {
        //
        if (boardStore.isExistByName(boardName)) {
            throw new RuntimeException("Board name already exists. --> " + boardName);
        }

        SocialBoard board = new SocialBoard(boardName);
        board.setCommentable(commentable);
        boardStore.create(board);
        return board.getUsid();
    }

    @Override
    public void removeSocialBoard(String boardUsid) {
        //
        if (!boardStore.isExist(boardUsid)) {
            throw new RuntimeException("No such a board --> " + boardUsid);
        }
    }
}
```

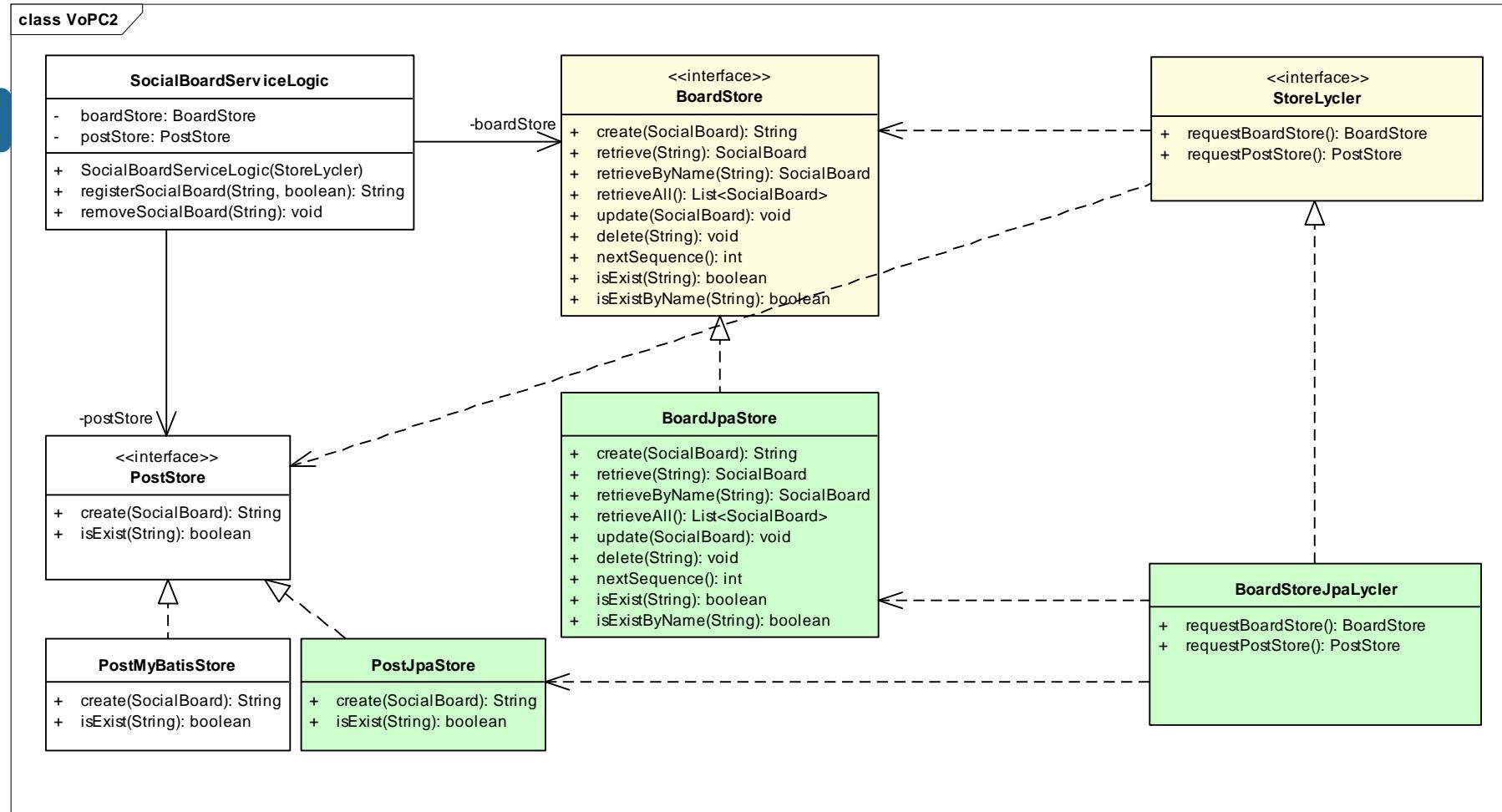
## OCP: Injection (2/2)

- ✓ SocialBoardPublisher는 필요에 따라 원하는 데이터 접근 방식을 선택할 수 있습니다.
- ✓ 주입하는 StoreLypler에 따라, 서비스는 해당 데이터 접근 방식을 따라 갈 것입니다.
- ✓ Publisher는 서비스의 클라이언트이므로 코드 변경은 불가피합니다. 물론, 구성정보로 처리하면 코드 변경을 피할 수 있습니다.

```
public class SocialBoardPublisher {  
    //  
    private static SocialBoardPublisher socialBoardPublisher;  
    private SocialBoardService socialBoardService;  
  
    private SocialBoardPublisher(String dataAccessType) {  
        //  
        StoreLypler storeLypler = null;  
  
        if (dataAccessType.equals("JPA")) {  
            storeLypler = new BoardStoreJpaLypler();  
        } else {  
            storeLypler = new BoardStoreMyBatisLypler();  
        }  
        this.socialBoardService = new SocialBoardServiceLogic(storeLypler);  
    }  
}
```

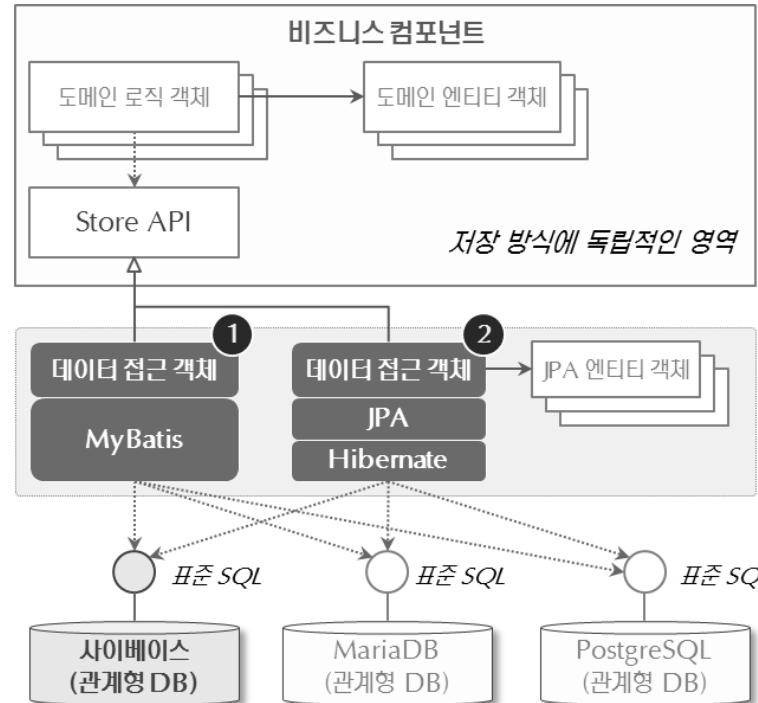


- ✓ 기존 MyBatis 접근방법은 그대로 두고(변경하지 않고), \*JpaStore를 추가함으로써 확장을 하였습니다.
- ✓ OCP 원칙을 지키면서 변화에 대응하는 방법은 추상화 개념을 잘 활용하는 것입니다.



# OCP: 요약

- ✓ 확장이 예측되는 지점에서 인터페이스 기반 설계는 자연스럽게 OCP 기반 확장으로 이어집니다.
- ✓ OCP 기반 확장은 아키텍처 설계에서도 확장 뿐만 아니라 대체(substitution)에서도 많이 사용합니다.
- ✓ 확장 과정에서 새로운 인터페이스 오퍼레이션을 식별함으로써, 기존 기능이 변경되는 경우도 있습니다. 이러한 변경은 불가피하지만, 설계 시점에 변경이나 확장 지점을 보다 정확하게 예측함으로써 줄일 수 있습니다.





## 모듈: 협업(collaboration)

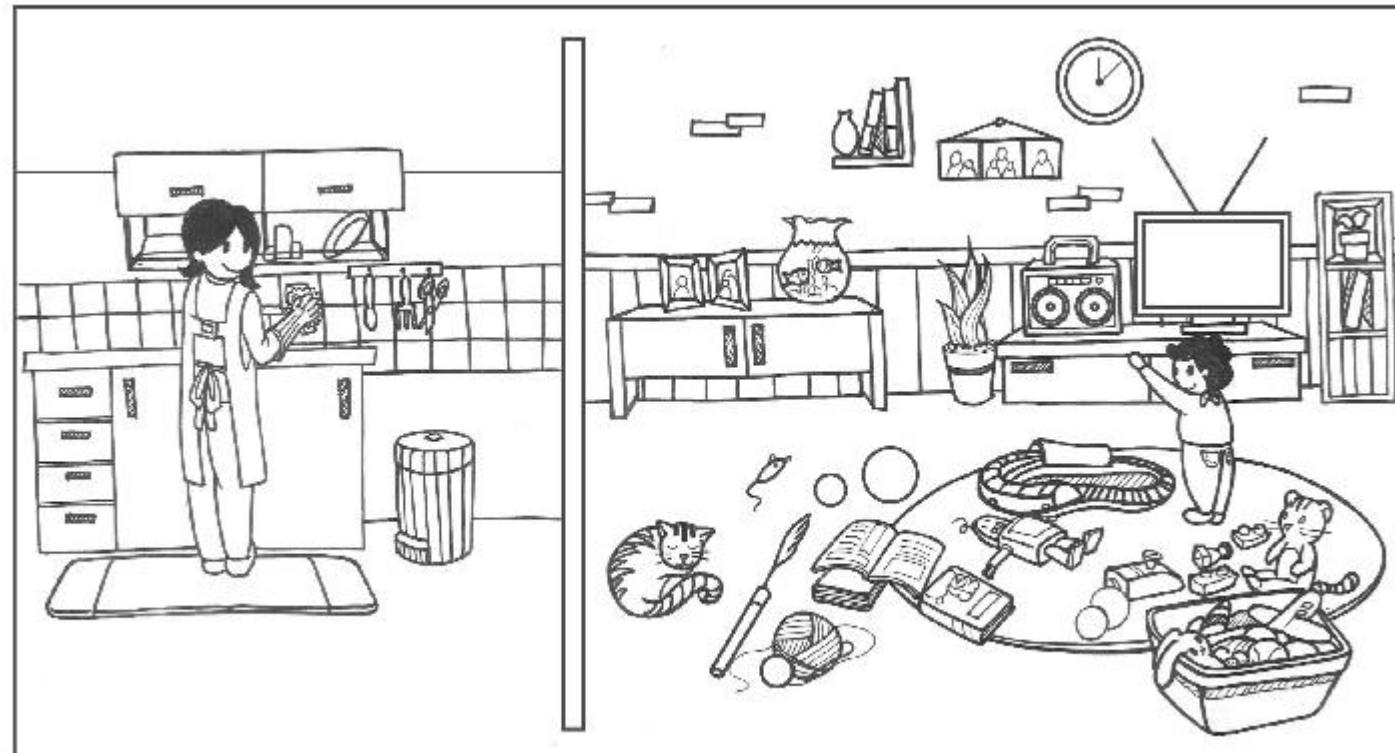
- ❖ 실세계에 존재하는 객체를 찾아내고 협업을 하게 합니다.
- ❖ 시나리오에 참여하는 각 객체의 역할과 책임을 정의합니다.
- ❖ 보이지 않은 객체를 찾아내고, 드러나지 않던 역할과 책임을 부여합니다.
- ❖ 그 결과를 프로그램으로 표현합니다.

## Radio

- ✓ 현장 스케치
- ✓ 시나리오
- ✓ 모델링
- ✓ 프로그래밍
- ✓ 요약

# 라디오 – 현장 스케치

- ✓ 따뜻한 봄날 오후, 엄마는 부엌에서 점심 설거지를 하고 있습니다.
- ✓ 민수는 거실에서 혼자 기차 놀이를 하고 있습니다.
- ✓ 엄마는 갑자리 라디오가 듣고 싶은데, 고무장갑 때문에 라디오 켜기가 불편합니다.
- ✓ 민수에게 켜 달라고 해야겠지요?



# 라디오 - 시나리오

- ✓ 실세계에서 일어나는 일은, 일상 언어(natural language)를 이용하여 다양한 형식으로 표현할 수 있습니다.
- ✓ 글의 형식은 대화문, 서술문, 실용문, 설명문, 메모, 편지, 안내문, 광고문, 등으로 다양할 수 있습니다.
- ✓ 사건이나 상황을 정확하고 빨리 전달하려 할 경우 시나리오 형식을 사용합니다. ← 유스케이스, 사용자 스토리
- ✓ 이번 모듈에서 객체지향 프로그램을 위해 실습할 내용은 다음 시나리오입니다.

[엄마는 부엌에서 설거지를 하고 있고, 민수는 거실에서 놀고 있다. TV 옆에 티브리 라디오가 놓여 있다.  
엄마는 라디오를 듣고 싶지만 젖은 고무장갑을 끼고 있어서 켤 수가 없다. 민수의 도움을 받기로 한다.]

엄마: 민수야 라디오 켤 수 있니?

민수: (음~ 내가 다섯 살이니까 켤 수 있겠지) 네, 켤 수 있어요. 엄마.

엄마: 그래? 그럼 라디오 좀 켜줄래?

민수: 네. (라디오를 켠다.)

라디오: 뉴스를 말씀드리겠습니다..

엄마: 고마워, 그런데 소리가 작은 것 같네. 소리 좀 높여줄래?

민수: 네. 엄마. (소리를 한 단 높인다.) 높였어요.

라디오: 뉴스를 말씀드리겠습니다...

엄마: 흄... 그런데, 아직도 소리가 좀 작아요. 조금 더 높여 주세요.

민수: 네. (다시 소리를 높인다.) 높였어요.

라디오: 뉴스를 말씀드리겠습니다..

엄마: 민수야 고마워. 참 잘 들리는구나.

# 라디오 – 시나리오

- ✓ “라디오” 시나리오는 자연 언어로 표현한 대화체 문장입니다. 이것을 프로그래밍으로 표현할 수 있을까요?
- ✓ 프로그램으로 표현하고, 실세계에서 엄마와 아이가 대화하듯이 실행을 할 수 있을까요?
- ✓ 물론, 프로그램 속의 아이와 엄마는 자신들이 하고 있는 것을 아래와 같이 열심히 출력해 주어야 겠지요.
- ✓ 자연 언어 → UML 모델링 언어 → Java 프로그래밍 언어라는 흐름을 지나서 아래와 같은 결과를 보여 주세요

코딩 실습 →

Menu

.....  
0. Program exit

1. 일하면서 라디오 켜기  
2. 라디오 소리 조정

.....  
Select number: 1

<KimPD:Director> Yeongmi씨, 라디오 들으면서 일하실래요?

<Yeongmi:Mom> 그럴까요, 라디오를 듣겠습니다.

<Yeongmi:Mom> 민수야, 라디오 켤 수 있니?

<Minsoo:Child> 라디오를 켤 수 있느냐구요?

<Minsoo:Child> 예, 켤 수 있어요.

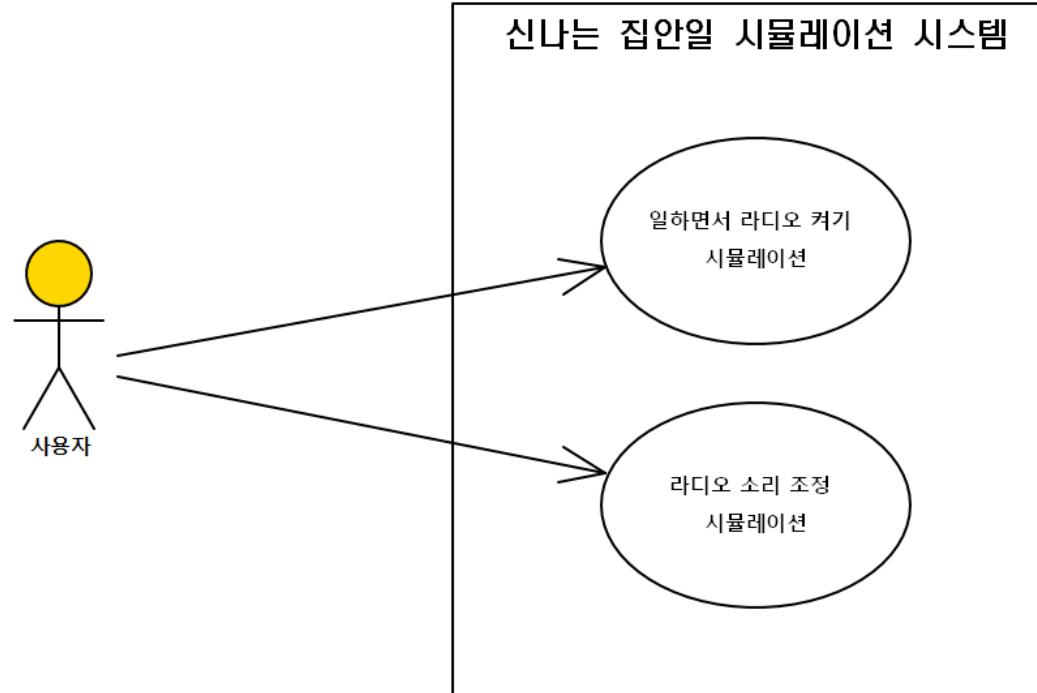
<Yeongmi:Mom> 그래, 그럼 라디오 좀 켜 줄래?

<Minsoo:Child> 예, 라디오 켤께요.

<Tivoli:Radio> [볼륨:1] 아, 아, 오늘의 뉴스를 말씀드리겠습니다...

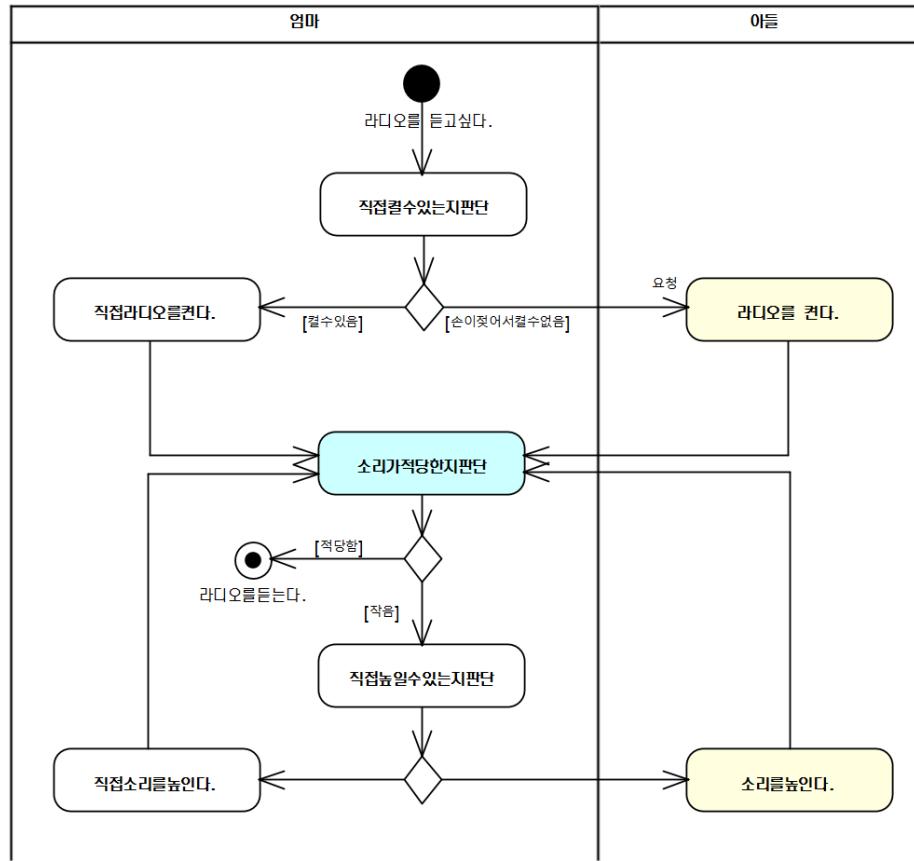
# 라디오 – 모델

- ✓ 실제 세계에서 벌어지는 일을 시스템 세계 안에서 표현하려할 때 만나는 첫번째 관문은 UML을 이용합니다.
- ✓ 시스템 요구사항을 명세하고, 그 명세를 충족하는 시스템을 모델링하는 것입니다.
- ✓ 이 시스템은 거실에서 일어난 일을 시스템 세계로 옮기는 작업으로 “신나는 집안일 시뮬레이터”라고 부릅니다.



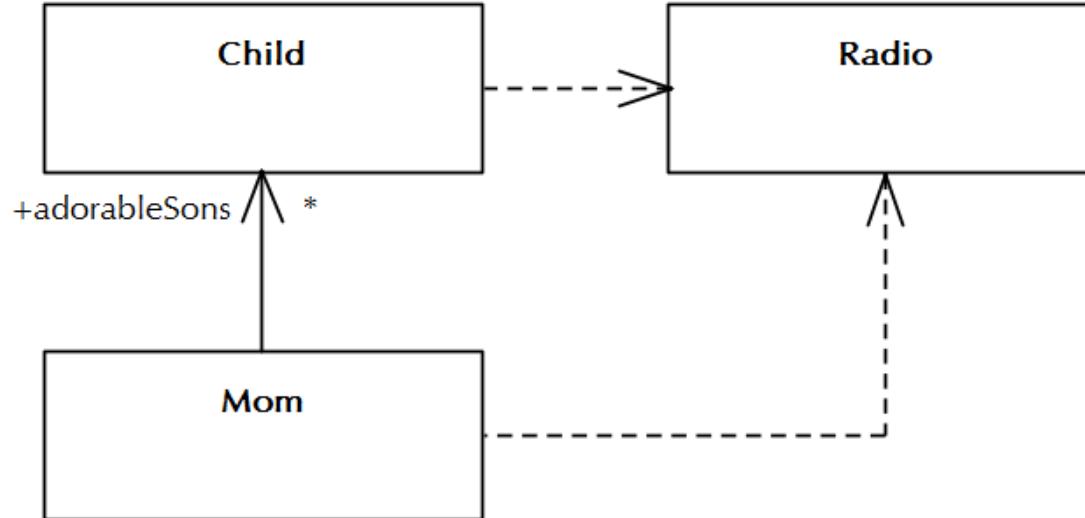
# 라디오 – 모델

- ✓ 요구사항을 이해하고 명세하는데 사용하는 도구에는 flow chart 등 여러가지가 있습니다.
- ✓ 아래 그림은 시스템 영역 안에서 발생하는 업무 흐름을 이해하여 표현하는 액티비티 다이어그램입니다.
- ✓ 엔지니어는 이 다이어그램을 작성하면서 전체 그림과 이야기의 흐름을 파악합니다.
- ✓ 요구사항 명세 단계에서는 시간 낭비를 하지 않고 큰 흐름을 파악하는데 초점을 두어야 합니다.



## 라디오 – 모델: 참여 클래스1

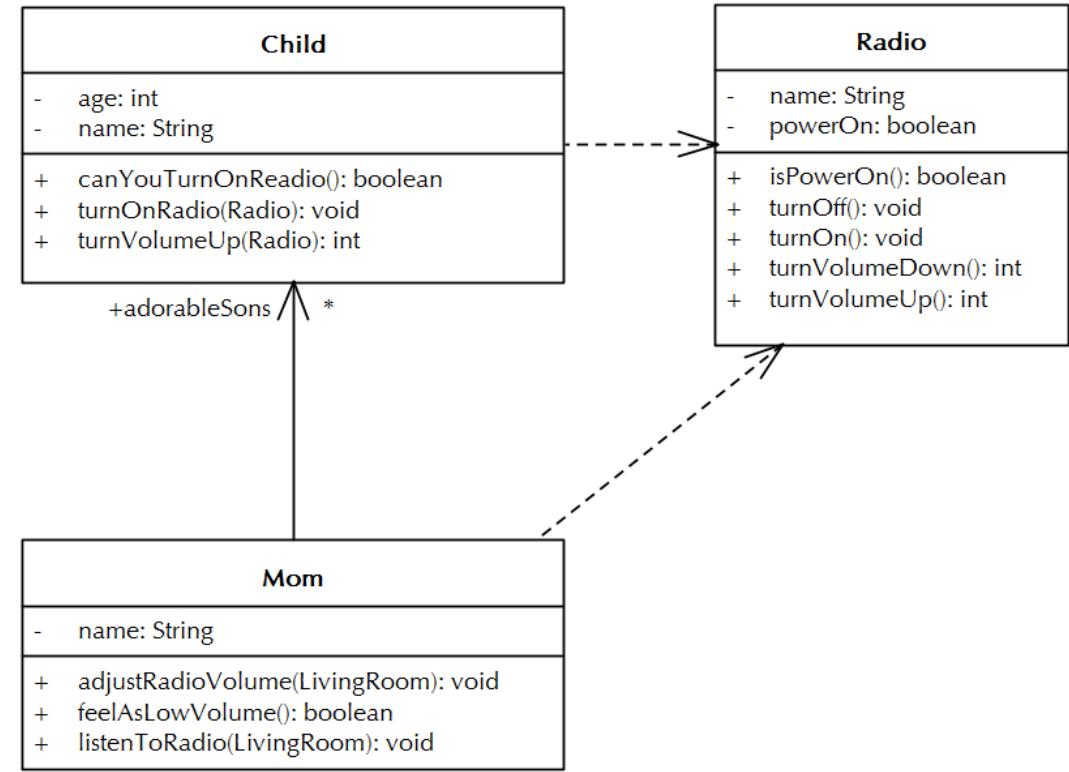
- ✓ 요구사항 명세를 마무리 하고 각 클래스가 어떤 역할과 책임을 가지고 시나리오에 참여하는지 고민을 합니다.
- ✓ 전체 시나리오를 진행되려면 각 클래스가 어떤 것인지를 알고, 어떤 활동을 해야 하는가를 생각해 봅니다.
- ✓ 처음 요구사항 명세에서 발견할 수 있는 클래스들은 엄마, 아이, 그리고 라디오 세 개입니다.



# 라디오 – 모델: 참여 클래스2

- ✓ 전체 시나리오 흐름을 생각하면서 각 클래스의 역할과 책임을 고민합니다.
- ✓ 각 클래스의 책임을 자세히 들여다 보면, 각 책임은 “하나 이상의 메소드” 형태로 나타냅니다.
- ✓ 시나리오를 진행하기 위해 클래스가 내부 또는 외부의 요청을 받아서 수행 가능한 작업이 메소드입니다.

엄마의 역할과 책임
<ul style="list-style-type: none"><li>✓ 아들인 민수가 거실에 있다는 사실은 안다.</li><li>✓ 라디오가 거실에 있다는 사실을 안다.</li><li>✓ 민수가 라디오를 켤 수 있는지는 물어봐야 한다는 것을 안다.</li><li>✓ 민수에게 라디오 켜기를 요청할 수 있다.</li><li>✓ 민수에게 라디오 소리를 높여 달라고 요청할 수 있다.</li><li>✓ 스스로 라디오를 켤 수 있다.</li><li>✓ 스스로 라디오 소리를 높일 수 있다.</li><li>✓ 소리가 적당한지 판단할 수 있다.</li></ul>
아이의 역할과 책임
<ul style="list-style-type: none"><li>✓ 스스로 라디오를 켤 수 있는지 판단한다.</li><li>✓ 라디오를 주면 라디오를 켜거나 끌 수 있다.</li><li>✓ 라디오를 주면 라디오의 소리를 높이거나 낮출 수 있다.</li></ul>
라디오의 역할과 책임
<ul style="list-style-type: none"><li>✓ On/Off 여부 질문에 대답을 한다.</li><li>✓ 끄거나 켤 수 있도록 인터페이스를 제공한다.</li><li>✓ 소리를 높이거나 낮출 수 있는 서비스를 제공한다.</li><li>✓ 소리의 범위를 갖고 있어서 범위 안에서 조정한다.</li></ul>

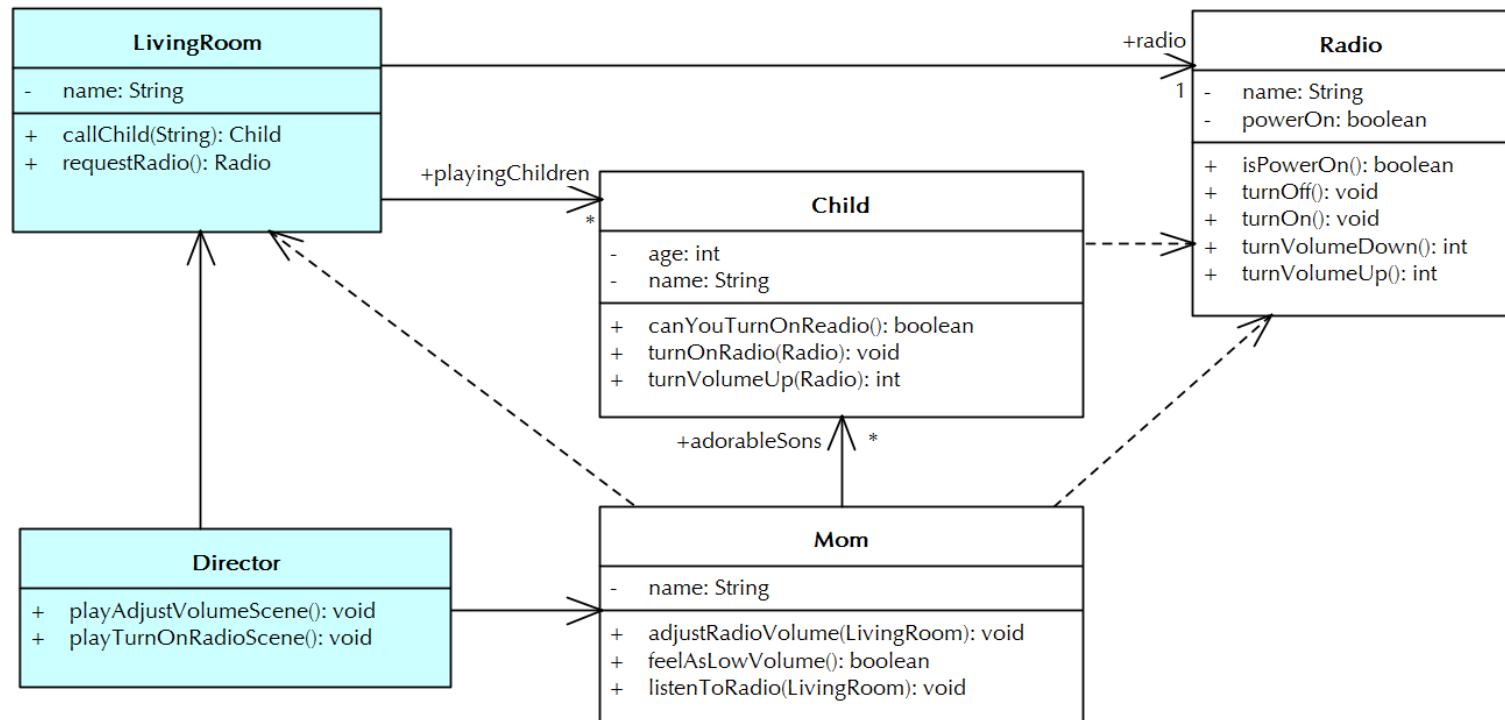


# 라디오 – 모델: 참여 클래스3

- ✓ 엄마, 아이, 라디오가 함께 협업하는 공간인 거실(LivingRoom)은 우리가 찾아야 할 클래스입니다.
- ✓ LivingRoom은 라디오, 민수 등을 모두 껴안아 주는 존재이며, 그 자체가 이 시나리오의 무대 역할을 합니다.
- ✓ 무대를 준비하고 무대 위의 클래스 간의 관계를 설정해 주는 연출가(Director)클래스가 필요합니다.
- ✓ 연출가에 거실 클래스가 추가된, 총 다섯 개의 클래스가 협업하는 시나리오를 멋지게 진행해 봅시다.

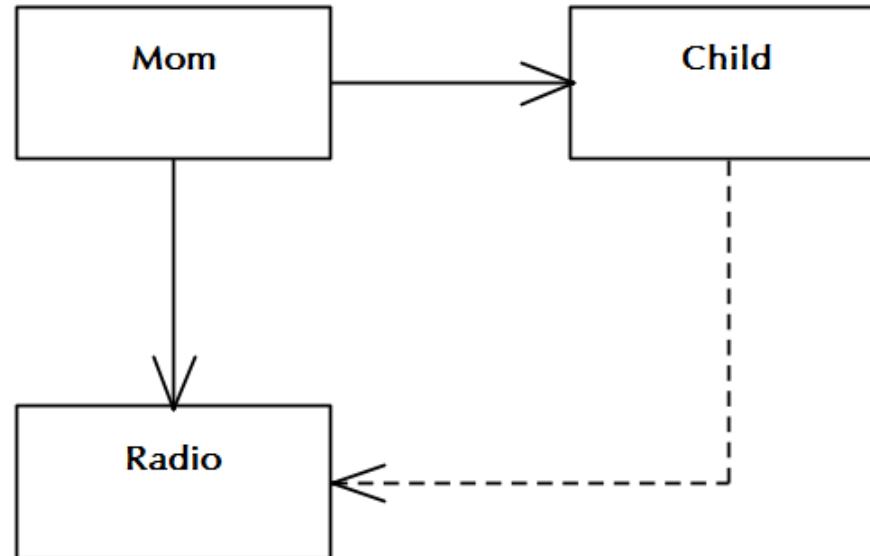
거실의 역할과 책임
✓ 라디오를 갖고 있다.
✓ 아이들이 놀 공간을 제공한다.
✓ 누군가 아이를 찾으면 불러준다.
✓ 향후 어떤 가전제품이든 받아 준다.

거실의 역할과 책임
✓ 무대를 준비한다.
✓ 라디오 켜기 장면을 연출하고 진행한다.
✓ 소리 조정 장면을 연출하고 진행한다.



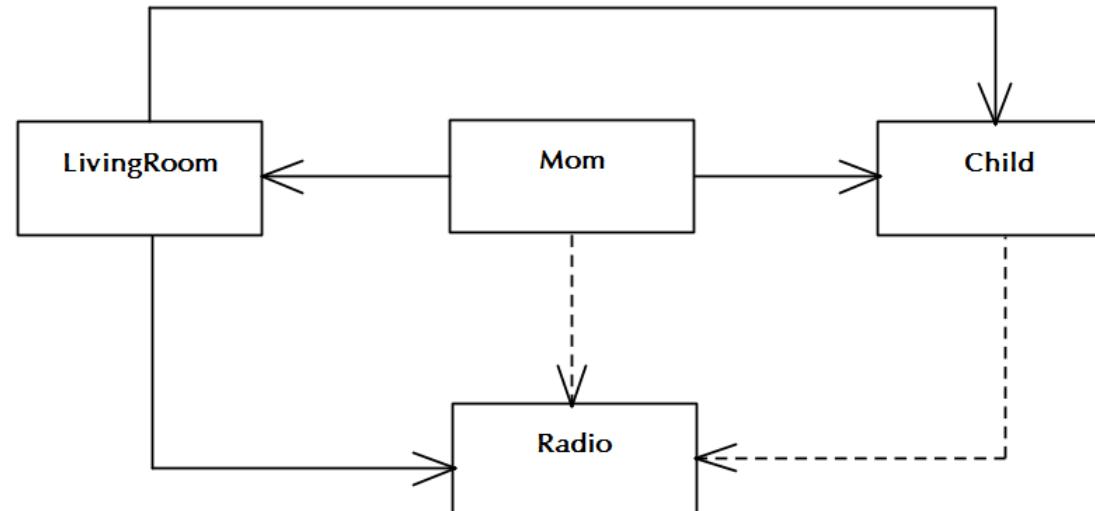
## 라디오 – 모델: 관계1

- ✓ 어떤 관계는 연관(association, 실선)관계이고, 어떤 관계는 의존(dependency, 점선)관계로 표현합니다.
- ✓ 상속(inheritance) 관계는 다른 관계 (연관 관계, 의존 관계 등)와 확실하게 구분할 수 있습니다.
- ✓ 하지만, 연관관계와 의존관계는 어느 정도 상황이나 주변 객체의 관계에 따라 변경되는 특성이 있습니다.
- ✓ 이 시나리오에서는 Mom이 Radio에 대한 소유권을 갖고 있는 모습이 자연스럽지 않습니다.



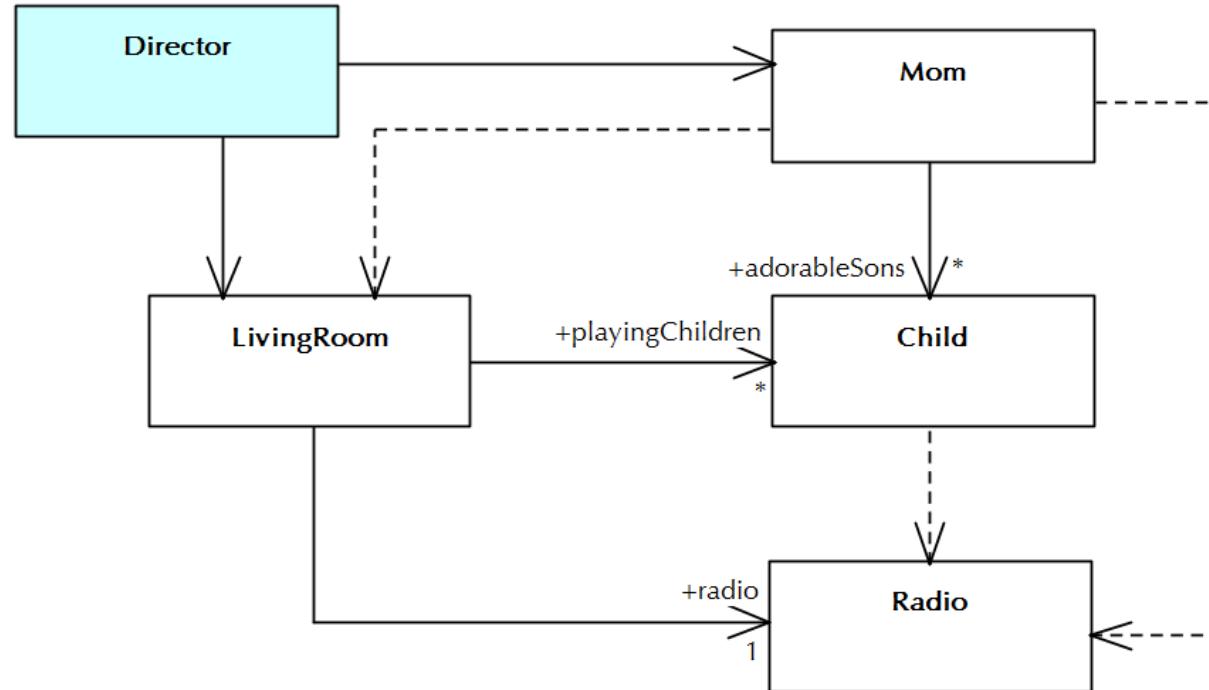
## 라디오 – 모델: 관계2

- ✓ LivingRoom이 Radio를 가지고 있으면 Mom은 Radio를 갖고 있어야 하는 책임으로부터 해방됩니다.
- ✓ LivingRoom과 Child는 일정 시간 지속적인 관계를 가지기 때문에 연관(association) 관계로 표현합니다.
- ✓ LivingRoom은 대본 진행을 위한 무대 역할을 담당하고 Mom은 LivingRoom과 협업하며 시나리오를 진행합니다. Mom과 LivingRoom 둘 중에 누가 시나리오를 이끌어 갈 것인가?



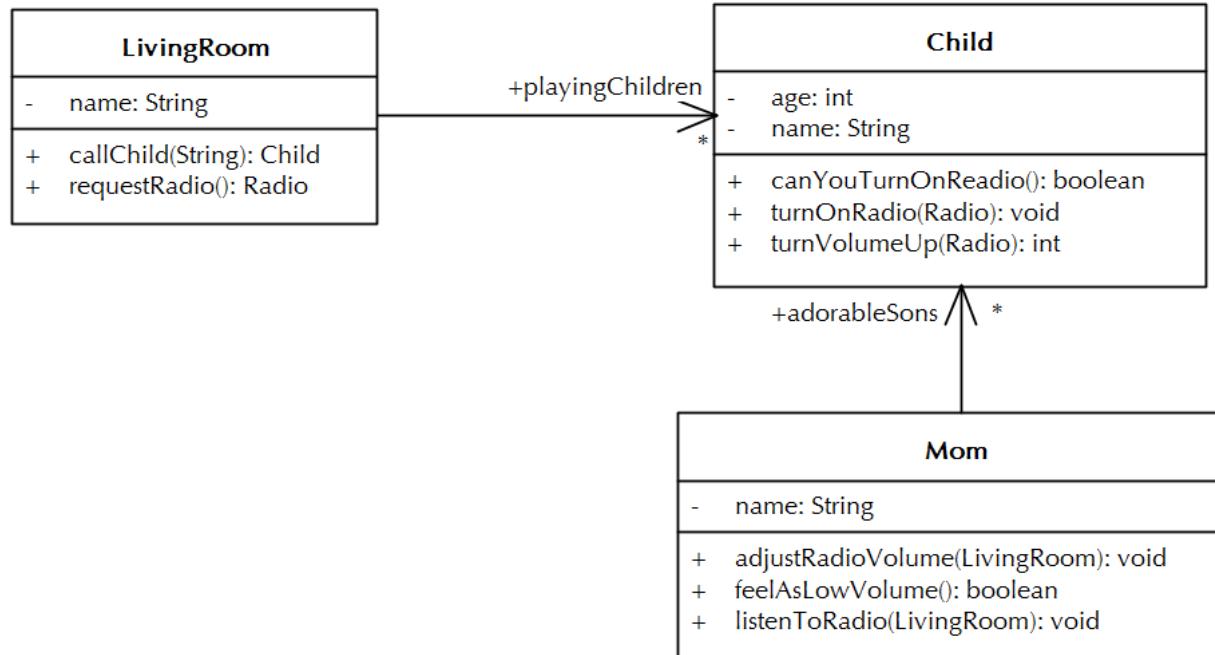
## 라디오 – 모델: 관계3

- ✓ 어떤 클래스가 협업하는 대상 클래스가 있고, 아무도 그 클래스를 잡고 있지 않다면 그 클래스를 직접 잡고 있어야 합니다. 즉, 연관(association) 관계를 갖고 있어야 합니다.
- ✓ 모델이 개선되면서 새로운 클래스가 등장하고 그 대상 클래스를 대신 잡아 주고 필요할 때 내게 넘겨줄 수 있다면, 그래서 필요할 때만 사용할 수 있다면 관계는 의존(dependency)관계로 느슨하게 변합니다.



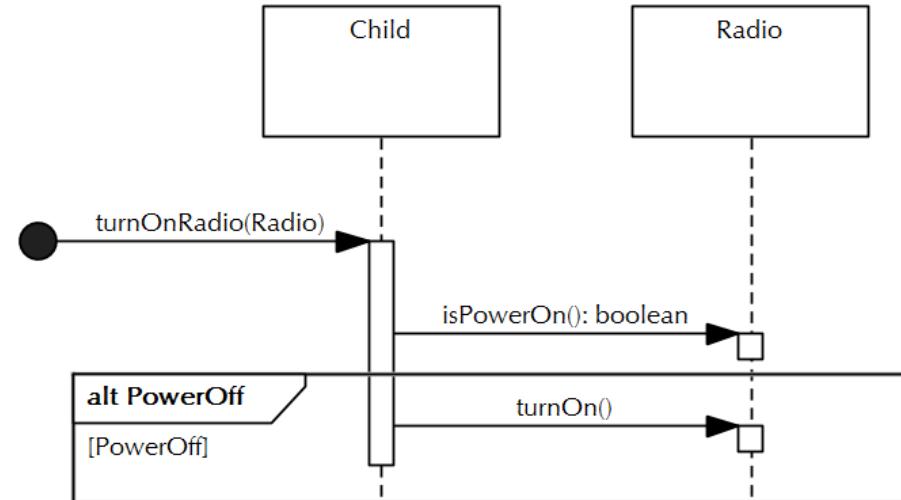
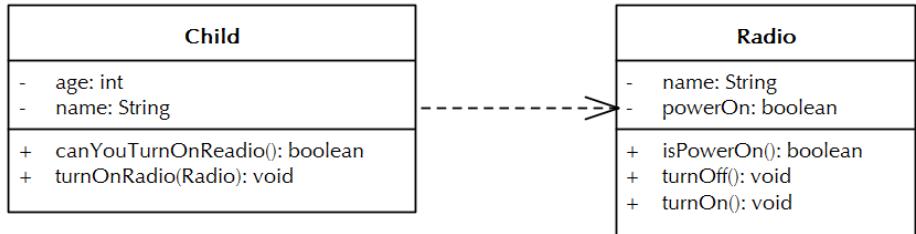
# 라디오 – 모델: 역할

- ✓ 세 개의 클래스 역할은 연관 관계를 가지는 두 클래스 간 항해(navigation)방향에서 발생합니다.
- ✓ 연관관계를 맺고 있는 상대방이 누구이며 어떤 관계인가에 따라 역할 이름은 달라집니다.
- ✓ 실세계에서도 A라는 사람은 아이들에게는 아빠이지만, 아내에게는 남편이며, 직장 후배들에게는 선배입니다.



# 라디오 – 모델: 협업 1 [라디오 켜기]

- ✓ 여러 객체가 참여하는 협업을 정의하기 전에 두 객체 사이에 간단한 상호작용을 모델로 표현해 봅니다.
- ✓ Child가 Radio를 켜는 과정을 표현합니다. Radio를 켤 때만 알고 있으면 되므로 의존 관계로 충분합니다.
- ✓ Child는 Radio가 제공하는 isPowerOn()이나 turnOn() 서비스 사용방법을 알아볼 수 있습니다.

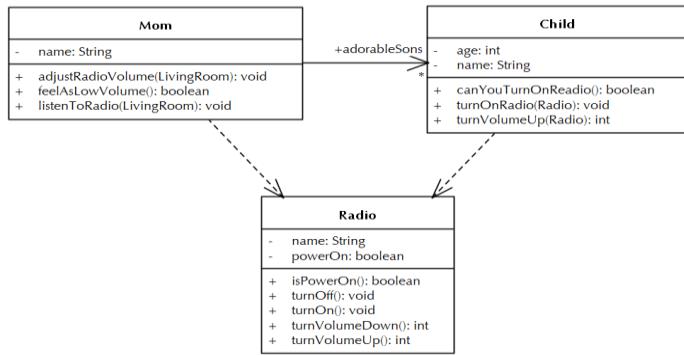


1. Radio상태를 Radio가 가지고 있을 때,
  - 1.1 켜져 있는지 라디오에게 물어 본다. 켜져 있지 않으면 켠다.
  - 1.2 라디오가 켜져 있으면 그냥 리턴하고 그렇지 않으면 켠다.
2. Radio상태를 Child가 가지고 있을 때,
  - 2.1 상태를 알고 있으므로, 켜져 있으면 그대로 둔다.
  - 2.2 상태를 알고 있으므로, 켜져 있지 않으면 켠다.

```
public void turnOn(Radio radio) {  
  
    if (radio.isPowerOn()) {  
        return;  
    }  
    radio.turnOn();  
}
```

# 라디오 – 모델: 협업 2 [요청하기]

- ✓ Mom은 아들에게 라디오를 켤 수 있는지 물어 볼 수 있고, Radio를 켜달라고 부탁할 수 있습니다.
- ✓ Radio가 켜졌는지 확인하고 켜져있다면 바로 리턴합니다. 실제 코드는 상태를 확인하고 대응하는 코드들이 있습니다. 이런 정도의 내용에 집중하다 보면 주요 시나리오를 놓칠 수 있어 모델에서 표현할 필요는 없습니다.

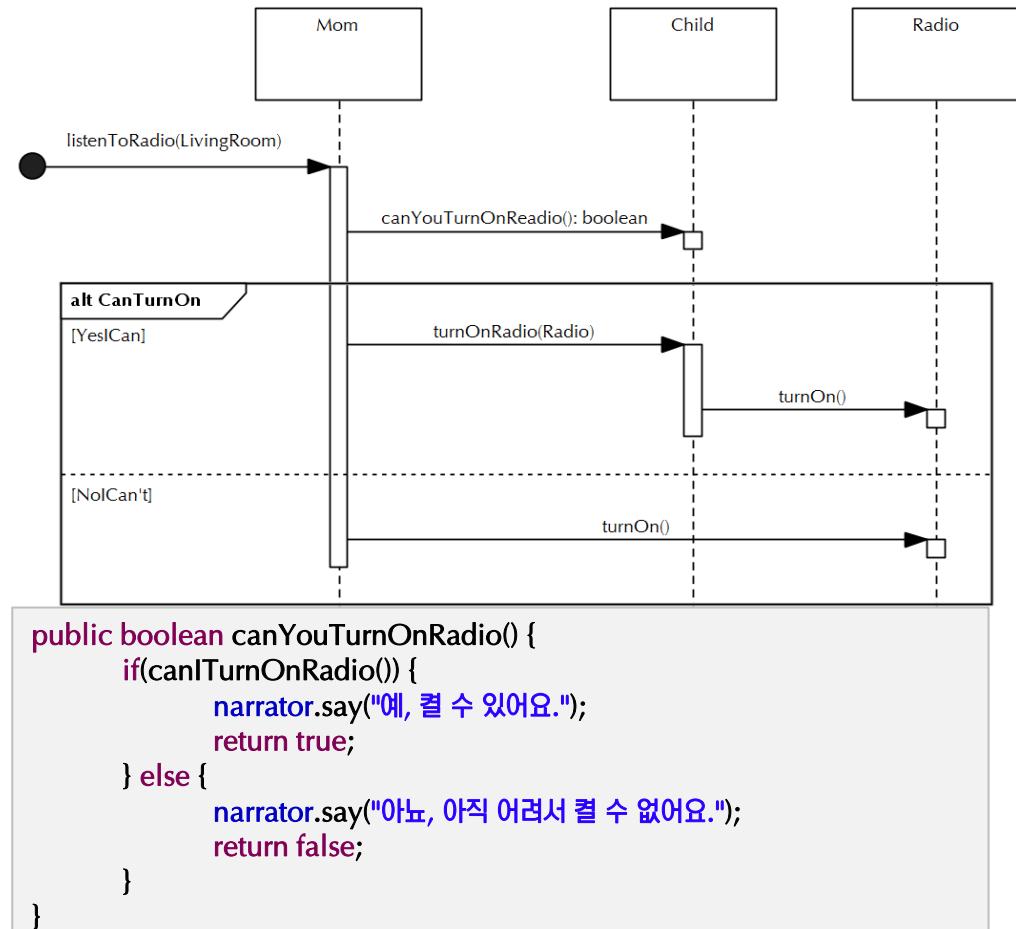


```

public void listenToRadio(LivingRoom livingRoom) {
    //
    Child smartSon = livingRoom.findFirstChild();
    Radio radio = livingRoom.findRadio();

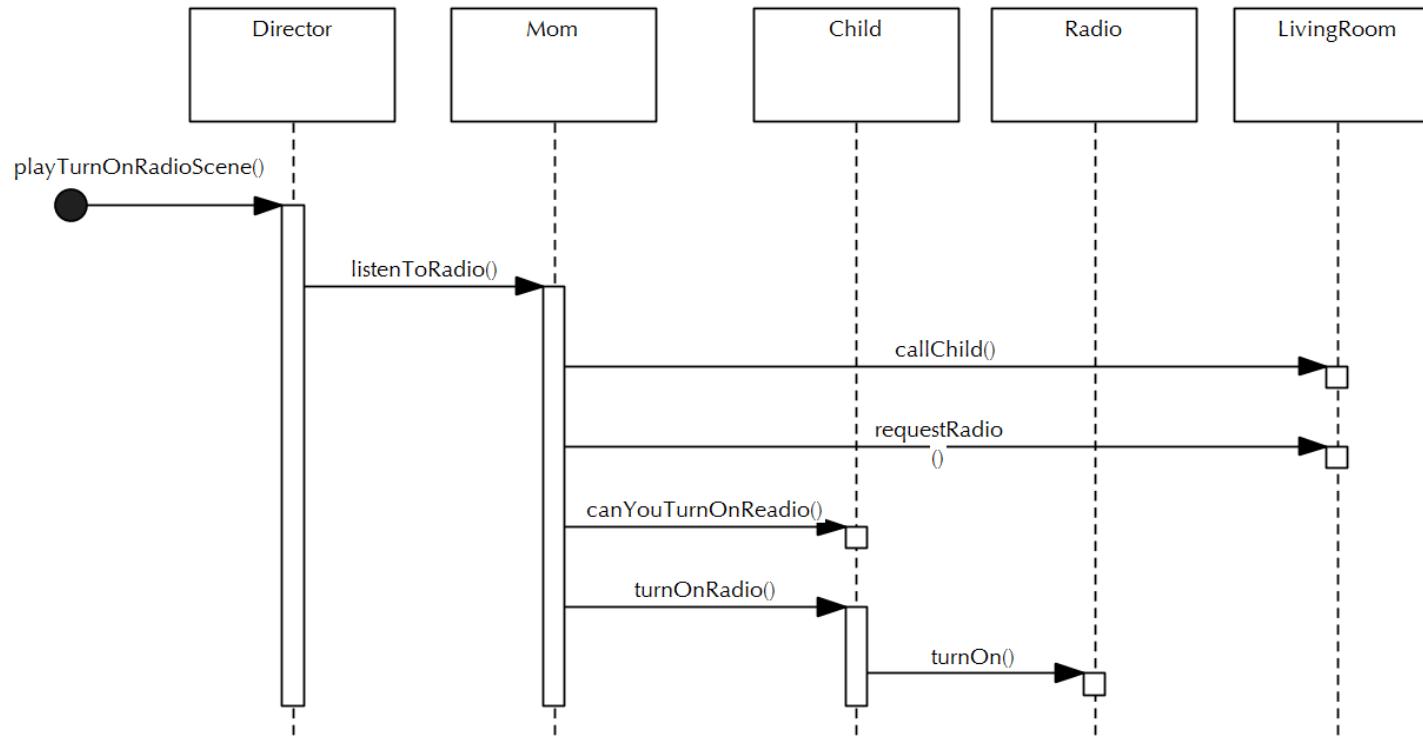
    if (radio.isPowerOn()) {
        return;
    }

    if(smartSon.canYouTurnOnRadio()) {
        smartSon.turnOnRadio(radio);
    } else {
        radio.turnOn();
    }
}
  
```



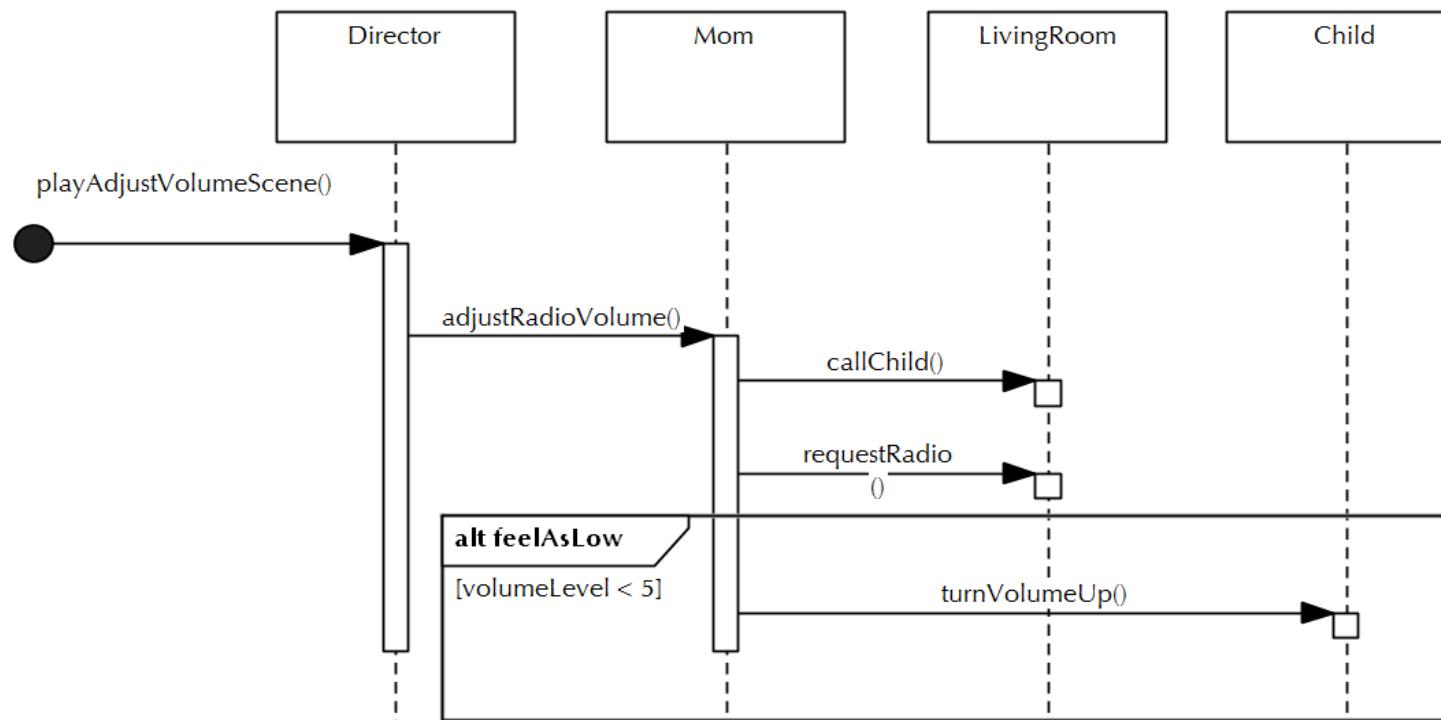
# 라디오 – 모델: 협업 3 [라디오 듣기]

- ✓ 라디오는 두 개의 유스케이스로 정의하고 각 유스케이스에서는 하나 이상의 협업 시나리오를 정의합니다.
- ✓ 앞에서 정의한 흐름에 따르면 복잡한 대안 경로를 가지지 않는 단순한 시나리오입니다.
- ✓ 라디오를 켜는 과정에서 이루어지는 협업 시나리오 (시퀀스 다이어그램)를 그려봅니다.



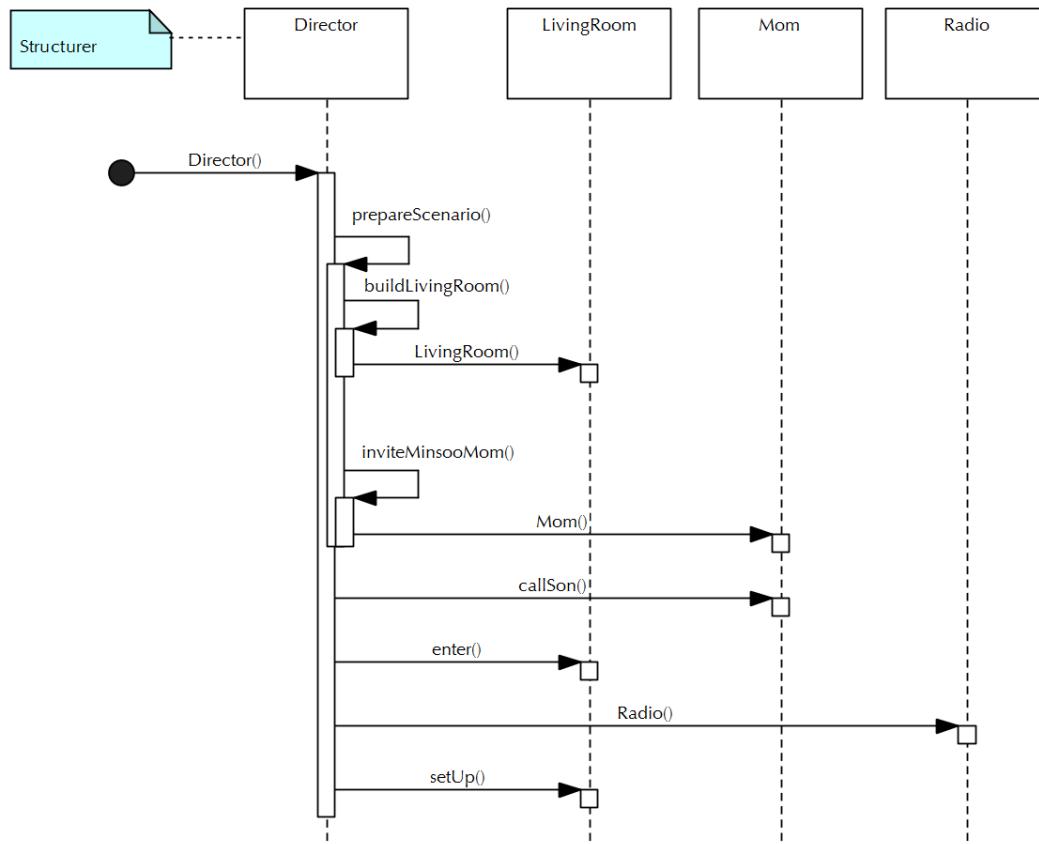
# 라디오 – 모델: 협업4 [소리조정]

- ✓ Radio를 켜고 나면 소리가 잘 안 들려서 Director는 Mom에게 소리 조정을 요청합니다.
- ✓ Mom은 아이가 거실에서 놀고 있는 것을 알고는 아이를 부른 다음, 거실에 있는 Radio를 달라고 합니다.
- ✓ Child에게 소리를 높여달라고 합니다. 소리 수준이 5가 될 때까지 반복해서 요청합니다.



# 라디오 – 모델: 협업5 (무대준비)

- ✓ 시나리오 진행에 필요한 무대는 누가 만드는 것일까요?
- ✓ 거실과 라디오, 엄마와 민수를 누가 무대 위로 옮겨놓았을까요?
- ✓ Director 클래스는 거실을 만들고, 민수 엄마를 초대하고, 아이를 불러서, 거실로 들여 보내고, 라디오를 만들어서 거실에 설치합니다.



```
public class Director {  
    ...  
    private void prepareScenario() {  
  
        this.livingRoom = buildLivingRoom();  
        this.mom = inviteMinsooMom("Yeongmi");  
        livingRoom.enter(mom.callSon("Minsoo"));  
        livingRoom.setUp(shopRadio("Tivoli"));  
    }  
  
    private LivingRoom buildLivingRoom() {  
  
        LivingRoom livingRoom = new LivingRoom();  
        return livingRoom;  
    }  
  
    private Mom inviteMinsooMom(String name) {  
        return new Mom(name);  
    }  
  
    private Radio shopRadio(String brandName) {  
        return new Radio("brandName");  
    }  
}
```

# 라디오 - 프로그래밍

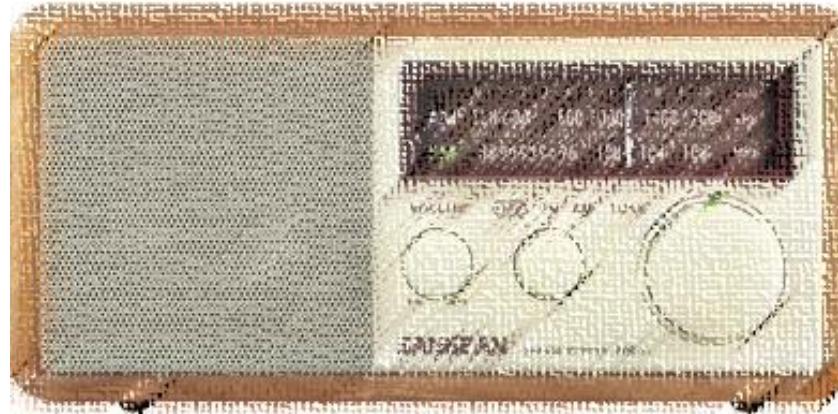
- ✓ 처음 작성했던 프로그램을 버리셔도 됩니다. 물론 그곳으로 부터 시작하는 것도 좋습니다.
- ✓ 지금까지 거실이라는 실세계 공간에서 일어나는 일들을, 참여하는 객체들의 협업 관점에서 살펴보았습니다.
- ✓ 협업에 참여하는 모든 객체들의 행위를 잘 표현할 수 있도록 “객체지향적인” 프로그래밍을 작성해 봅니다.
- ✓ 실세계의 언어로 이해하고, 모델링 언어로 정리하고, 프로그래밍 언어로 확인합니다. 확인이 꼭 필요합니다.

코딩 실습 →

```
public void showMenuAndAction() {  
    //  
    while (true) {  
        System.out.println();  
        System.out.println(".....");  
        System.out.println("  일하면서 라디오 듣기 메뉴");  
        System.out.println(".....");  
        System.out.println("  0. Program exit");  
        System.out.println("  1. 라디오 켜기");  
        System.out.println("  2. 라디오 소리 조정");  
        System.out.println(".....");  
  
        int inputNumber = acceptMenuItem("Select number");  
  
        switch (inputNumber) {  
        //  
        case 1:  
            turnOnTheRadio();  
            break;  
        case 2:  
            adjustTheVolume();  
            break;  
        case 0:  
            exitProgram();  
            return;  
  
        default:  
            System.out.println("Choose again!");  
        }  
    }  
}  
  
public boolean canYouTurnOnRadio() {  
    //  
    narrator.say("라디오를 켤 수 있나요?");  
  
    if(canITurnOnRadio()) {  
        narrator.say("예, 켤 수 있어요.");  
        return true;  
    } else {  
        narrator.say("아뇨, 아직 어려서 켤 수 없어요.");  
        return false;  
    }  
}  
  
public boolean canYouAdjustVolume() {  
    //  
    narrator.say("소리를 조정할 수 있나요?");  
  
    if(canIAdjustVolume()) {  
        narrator.say("예, 할 수 있어요.");  
        return true;  
    } else {  
        narrator.say("아뇨, 아직 어려서 할 수 없어요.");  
        return false;  
    }  
}  
  
public void turnOnRadio(Radio radio) {  
    //  
    if (!canITurnOnRadio()) {  
        narrator.say("저는 라디오를 켤 수 없어요.");  
        return;  
    }  
  
    if (radio.isPowerOn()) {  
        narrator.say("이미 켜져 있는데요...");  
        return;  
    }  
}
```

# 라디오 – 요약

- ✓ 실세계를 하나하나 따져 보면 상상할 수 없을 정도로 복잡한 메커니즘을 갖추고 있으며, 매우 효율적이고 효과적으로 모든 일이 돌아가고 있음을 알 수 있습니다.
- ✓ 객체 지향 사고의 핵심은 실세계의 메커니즘과 효율성을 시스템 세계로 투영하는 일입니다.
- ✓ 그러기 위해서는 실세계의 언어 → 모델링 언어 → 프로그래밍 언어로의 이전에 익숙해져야 합니다.
- ✓ 실세계 매팅의 첫 번째 시나리오인 라디오를 이해하고 프로그래밍 했습니다. 여러분의 생각은 어떠신가요?





## 모듈: 협업과 추상화

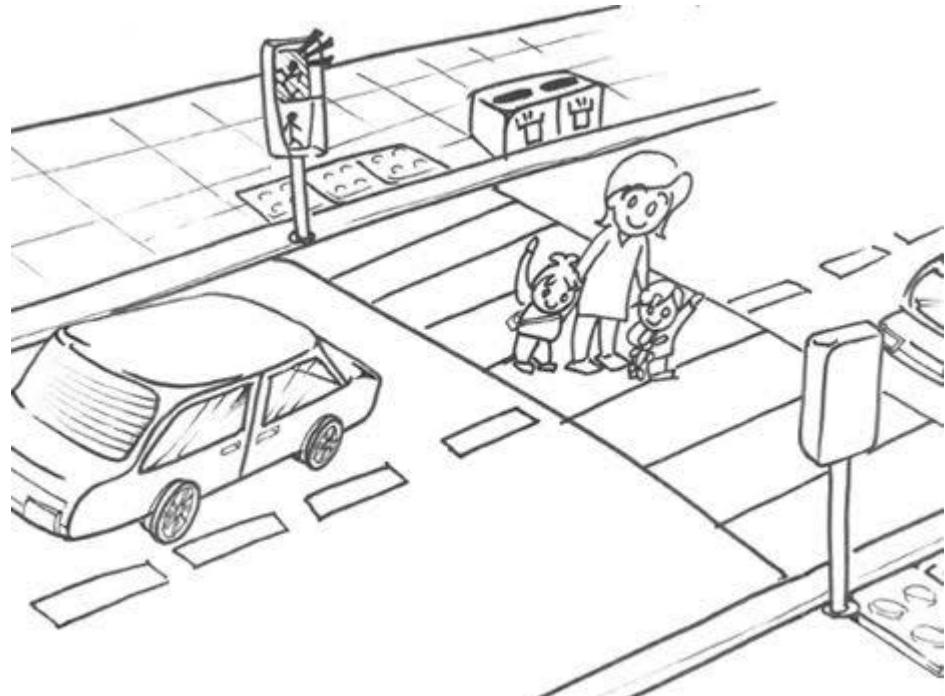
- ❖ 복잡한 대상을 단계적으로 다루는 방법을 이해합니다.
- ❖ 참여 객체 식별, 역할과 책임 식별, 협업 구조 결정 등을 진행합니다.
- ❖ 첫 단추를 끼는 방법을 알아봅니다.

## Crosswalk 1

- ✓ 현장 스케치
- ✓ 시나리오
- ✓ 모델
- ✓ 프로그래밍

# 횡단보도 1 – 현장 스케치

- ✓ 행인이 횡단보도(crosswalk)를 건너는 과정을 자연어, 모델링 언어, 프로그래밍 언어로 표현해 봅니다.
- ✓ 횡단보도 관련 규칙은 모두가 잘 알고 있습니다. 여러분은 횡단보도 도메인 전문가입니다.
- ✓ 현장을 이행하고, 글로 표현함으로써 더 깊게 이해하고, 이를 바탕으로 모델링을 합니다. ← 요구 명세 존재이유
- ✓ 한 번에 구현하기에는 너무 복잡합니다. 조금씩 개발하는, 즉 점점 성장시켜 가는 방식으로 개발합니다.



엄마가 아이들과 횡단보도를 건너려고 횡단보도로 걸어간다. 신호등을 확인하고는 멈춰선다. 적색등에 불이 켜져 있다. 몇 초가 지난 후 녹색등에 불이 들어온다. 곧이어 음성 안내기의 안내소리가 들린다. “녹색등입니다. 건너가십시오.” 이어 알림음이 계속된다. “뚜루르, 뚜루르,...” 이 여성은 빠른 걸음으로 걷기 시작한다. 시선은 맞은 편 신호등 바로 옆에 있는 LCD 남은시간표시기에 고정되어 있다. 20, 19, 18,... 1초 단위로 숫자가 줄어든다. 어느덧 남은시간표시기는 숫자 “5”를 보여준다. 곧바로 음성 안내기의 안내소리가 들려왔다. “위험합니다. 건너지 마십시오.” 이 여성은 급한 마음에 발걸음을 재촉한다. 횡단보도를 막 건넜을 때, 신호기의 적색등에 불이 들어온다. LCD 남은시간표시기의 불이 꺼지며, 음성 안내기가 외친다. “적색등입니다. 건너지 마십시오.” 이어 계속되던 알림음도 멎었다.

# 횡단보도 1 – 시나리오

- ✓ 성장의 첫 단계로 아주 작은 시나리오를 작성했습니다.
- ✓ 주요 객체(Grady Booch의 “Key concept”)를 식별하고, 가장 기본적인 협업을 이해합니다.
- ✓ 작은 시나리오는 전체를 개략적으로 구성하여도 되고, 단계별로 나누어서 시나리오를 진행해도 됩니다.

(막이 열리면 11번가의 모습이 보이고, 무대 한 가운데 횡단보도가 있다. 보행자인 민수가 등장한다.)

연출가 : 민수씨, 이 횡단보도(이름은 cross11)를 건너가세요.

민수 : 아, 이 횡단보도요? 알겠습니다. 먼저 보행자 신호등을 확인해야겠는데요.

cross11씨, 보행자 신호등 좀 주세요.

cross11 : (보행자 신호등을 건네며) 여기 있어요.

민수: pedSignal씨, 지금 녹색등이 켜졌나요?

pedLight: 아니오, 지금은 적색등이 켜져 있습니다.

민수: 아, 그렇군요. 기다려야겠군요(멈춰선다.)

# 횡단보도 1 – 시나리오: 실행 결과

- ✓ 실행한 결과 화면은 다음과 같습니다.
- ✓ 프로그램 중에서 각 객체들은 자신이 수행하는 일을 기록함으로써 사용자가 어떤 협업이 있는지 알게 해줍니다.
- ✓ 자연 언어 → UML 모델링 언어 → Java 프로그래밍 언어라는 흐름을 지나서 아래와 같은 결과를 보여 주세요.

코딩 실습 →

```
<KimPD:Director> 시나리오를 시작합니다. 큐~  
<KimPD:Director> 횡단보도를 건설합니다.  
<KimPD:Director> 보행자 minsoo 씨가 등장합니다.  
<KimPD:Director> minsoo 씨 cross11 횡단보도를 건너 가세요.  
<minsoo:Pedestrian> cross11를 건너가라구요? 알았어요.  
<minsoo:Pedestrian> cross11 횡단보도, 보행자신호등을 주세요.  
<cross11:Crosswalk> 보행자 신호등 여기 있습니다.  
<minsoo:Pedestrian> pedSignal 보행자신호등, 녹색등이 켜져 있나요?  
<pedSignal:PedestrianSignal> 적색등이 켜져 있습니다.  
<minsoo:Pedestrian> 적색등이 켜져 있군요. 녹색등이 들어올 때까지 기다려야겠군요.
```

# 횡단보도 1 – 시나리오: 실행 결과

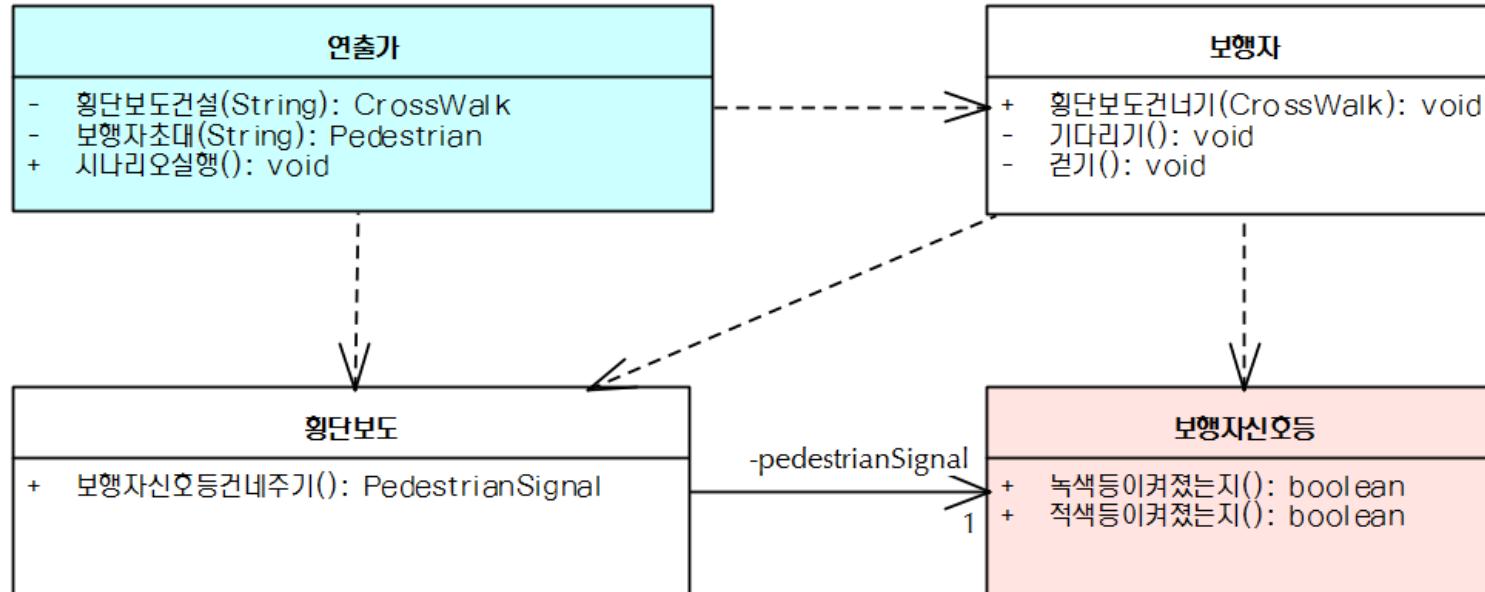
- ✓ 프로그램을 실행하는 주체가 있어야 합니다. 여기서는 김PD(Director 클래스)로 정하겠습니다.
- ✓ kimPd.playScenario()를 호출하면 시나리오를 시작하는데, 절차는 다음과 같습니다. 나머지는 프로그래밍...

코딩 실습 →

```
public void playScenario() {  
    //  
    talking("시나리오를 시작합니다. 큐~");  
  
    // 1. Build a crosswalk named "cross11".  
    String crosswalkName = "cross11";  
    Crosswalk crosswalk = buildCrosswalk(crosswalkName);  
  
    // 2. Invite a pedestrian named "minsoo".  
    String pedestrianName = "minsoo";  
    Pedestrian pedestrian = invitePedestrian(pedestrianName);  
  
    // 3. Order to cross.  
    talking(String.format("%s 씨 %s 횡단보도를 건너 가세요.", pedestrian.getName(), crosswalk.getName()));  
    pedestrian.cross(crosswalk);  
}  
  
private Crosswalk buildCrosswalk(String name) {  
    //  
    talking("횡단보도를 건설합니다.");  
    Crosswalk crosswalk = new Crosswalk(name);  
  
    PedestrianSignal pedestrianSignal = new PedestrianSignal("pedSignal");  
    crosswalk.setPedestrianSignal(pedestrianSignal);  
  
    return crosswalk;  
}  
  
private Pedestrian invitePedestrian(String name) {  
    talking(String.format("보행자 %s 씨가 등장합니다.", name));  
    return new Pedestrian(name);  
}
```

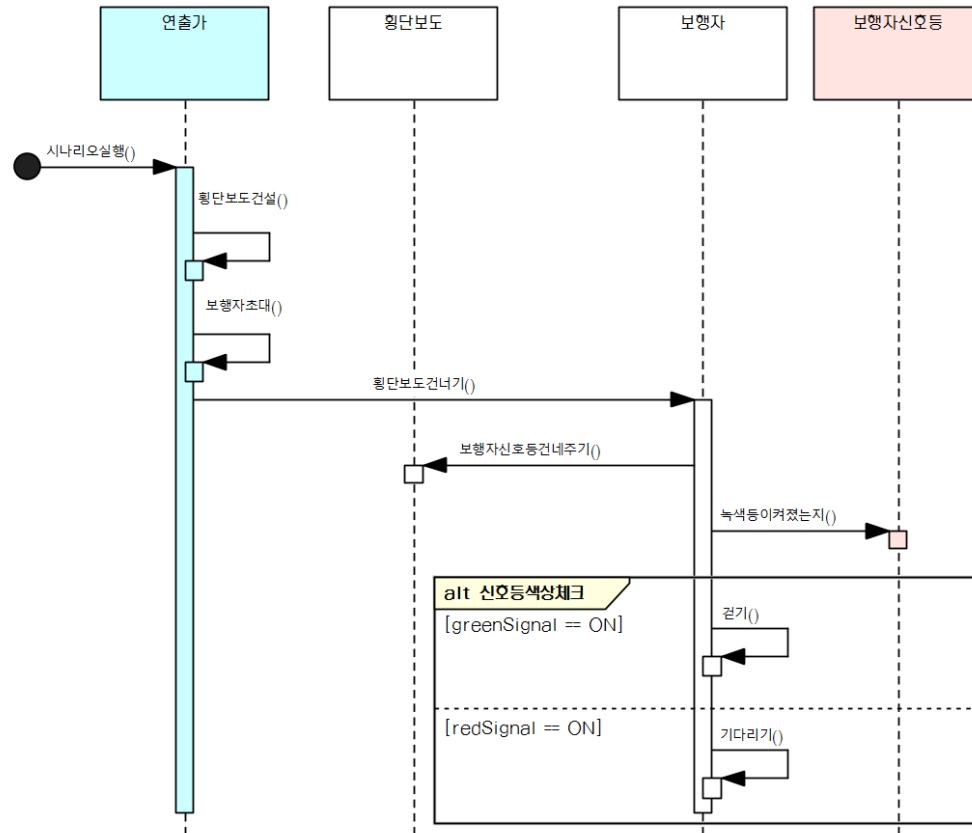
# 횡단보도 1 – 모델:참여 객체들

- ✓ 실제 세계에서 벌어지는 일을 시스템 세계 안에서 표현하려할 때 만나는 첫번째 관문은 UML입니다.
- ✓ 시스템 요구사항을 명세하고, 그 명세를 충족하는 시스템을 모델링하는 것입니다.
- ✓ 참여 클래스들을 클래스 다이어그램으로 표현하여 객체들의 책임과 관계들을 표현해 봅니다.
- ✓ 이 시스템은 연출가가 시나리오를 진행하기 위해 횡단보도와 보행자를 알고 있어야 합니다.



# 횡단보도 1 – 모델: 협업

- ✓ 각 클래스들이 어떤 방식으로 협력하여 시나리오를 진행하는지 살펴봅니다.
- ✓ 객체들 간의 협업을 시간 순서에 따라 표현한 다이어그램이 시퀀스 다이어그램입니다.
- ✓ 시나리오 진행 순서 그래도 시퀀스 다이어그램을 표현합니다.



# 횡단보도 1 – 모델: Director

- ✓ 연출가(Director)는 시나리오를 진행합니다.
- ✓ 연출가는 횡단보도를 건설하고, 보행자를 초대하고, 보행자 민수에게 횡단보도를 건너라고 이야기 합니다.
- ✓ 연출가는 대본에 쓰여진 대로 그대로 연출하고 있습니다.
- ✓ Java 프로그램 소스는 UML 모델로 표현한 클래스 Director 와 메소드 정의 수준에서 일치합니다.

```
public class Director {  
    public static void main(String[] args) {  
        Director director = new Director();  
        director.playScenario();  
    }  
  
    public void playScenario() {  
  
        // 1. Build a crosswalk named "cross11".  
        String xingName = "cross11";  
        CrossWalk cross11 = buildCrossWalk(xingName);  
  
        // 2. Invite a pedestrian named "minsoo".  
        String pedestrianName = "Minsoo";  
        Pedestrian minsoo = invitePedestrian(pedestrianName);  
  
        // 3. Order to cross.  
        minsoo.cross(cross11);  
    }  
  
    ...  
  
    private CrossWalk buildCrossWalk(String name) {  
        return new CrossWalk(name);  
    }  
  
    private Pedestrian invitePedestrian(String name) {  
        return new Pedestrian(name);  
    }  
}
```

# 횡단보도 1 – 모델: Crosswalk

- ✓ 횡단보도의 책임은 보행자 신호등을 갖고 있다가 달라고 하면 건네주는 것입니다.
- ✓ 횡단보도는 사람들이 건널 수 있도록 제공할 뿐만 아니라 개념을 담고 있는 아주 중요한 역할입니다.
- ✓ 그 클래스가 얼마나 중요한 역할을 하는지 알아보는 간단한 방법은 객체를 빼보는 것입니다.

```
public class Crosswalk {  
    //  
    private String name;  
    private PedestrianSignal pedestrianSignal;  
  
    public Crosswalk(String name){  
        //  
        this.name = name;  
        this.pedestrianSignal = new PedestrianSignal(name + ".pedSignal");  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public PedestrianSignal getPedestrianSignal(){  
        return pedestrianSignal;  
    }  
}
```

# 횡단보도 1 – 모델: Pedestrian

- ✓ 보행자(Pedestrian)는 외부로 열린 메소드 cross()에서 길을 건너는데 필요한 판단을 하고 행동을 합니다.
- ✓ 현재 시나리오에서는 그냥 간단하게 현재 상황을 판단하고 내용을 출력해 주는 것으로 실행을 대신합니다.
- ✓ 메소드 talking()는 콘솔로 현재 진행상황을 출력해주는 유ти리티 메소드입니다.

```
public class Pedestrian {  
    private String name;  
  
    public Pedestrian(String name) {  
        this.name = name;  
    }  
    public void cross(CrossWalk crossWalk) {  
  
        talking("건너가라구요? 알았어요.");  
        PedestrianSignal pedestrianSignal = crossWalk.getPedestrianSignal();  
  
        if(pedestrianSignal.isGreenSignalOn()) {  
            talking("파란불이군요.");  
            talking("건너갑니다.");  
        } else {  
            talking("빨간불이군요.");  
            talking("잠시 기다려야겠군요.");  
        }  
    }  
}
```

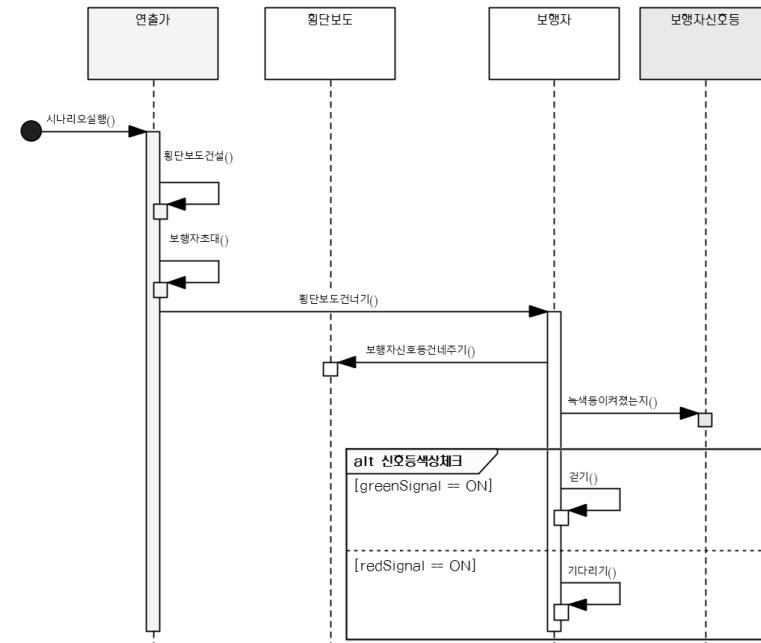
# 횡단보도 1 – 모델: 보행자 신호등(PedestrianSignal)

- ✓ 보행자 신호등은 적색등과 녹색등의 상태를 표현하기 위해 boolean 타입의 속성 두 개를 가지고 있습니다.
- ✓ 메소드 `toString()`을 이용하여 보행자신호등의 상태를 표현하고 있습니다.
- ✓ 신호등이 켜졌는지 물어볼 수 있는 메소드에는, `isGreenLightOn()`과 `isRedLightOn()`가 있습니다.

```
public class PedestrianSignal {  
    //  
    private String name;  
    private boolean redLightOn;  
    private boolean greenLightOn;  
  
    public PedestrianSignal(String name){  
        //  
        this.name = name;  
        this.redLightOn = true;  
        this.greenLightOn = false;  
    }  
  
    public boolean isGreenLightOn(){  
        return greenLightOn;  
    }  
  
    public boolean isRedLightOn() {  
        return redLightOn;  
    }  
    ...  
}
```

# 횡단보도 1 – 요약

- ✓ 첫번째 이해(명세, 모델링, 프로그래밍)에서 가장 중요한 것은 핵심 객체의 역할과 책임을 식별하는 것입니다.
- ✓ 이 정보를 기반으로 협업을 어떻게 하는 것이 좋은가 고민하고, 필요하면 역할과 책임을 조정합니다.
- ✓ 첫 단추를 끼는 것 만큼 중요한 활동입니다. 멀리서, 특히 사용자 관점에서 바라보는 것이 중요합니다.





## 모듈: 협업과 추상화

- ❖ 복잡한 대상을 단계적으로 다루는 방법을 이해합니다.
- ❖ 참여 객체 식별, 역할과 책임 식별, 협업 구조 결정 등을 진행합니다.
- ❖ 첫 번째 단계로부터 시나리오를 확장하는 방법을 배웁니다.

## Crosswalk 2

- ✓ 시나리오
- ✓ 모델
- ✓ 프로그래밍
- ✓ 요약

## 횡단보도 2 – 시나리오

- ✓ 처음 시나리오를 통해서 횡단 보도라는 공간을 이해하고, 그 속에 존재하는 객체들을 찾았습니다.
- ✓ 간단한 협업을 통해서, 각 객체의 역할과 책임을 부여하고는 프로그래밍을 통해 확인했습니다.
- ✓ 이제 시나리오를 조금 더 확장하여, 보다 넓고 깊게 이해하는 시간을 갖겠습니다.
- ✓ 두 번째 시나리오는 다음과 같습니다.

(막이 열리면 11번가의 모습이 보이고, 무대 한 가운데 횡단보도가 있다. 보행자인 민수가 횡단보도 왼쪽에서 등장한다.)

연출가: 민수씨, 11번가 첫 번째 횡단보도인 cross11을 왼쪽에서 건너세요.

민수: 아, 이 횡단보도요? 알겠습니다. 먼저 보행자 신호등을 확인해야 겠는데요. Cross11씨, 오른쪽 보행자 신호등 좀 주세요.

Cross11: (오른쪽 보행자 신호등(Cross11.pedSignal)을 건네며) 여기 있어요.

민수: pedSignal씨, 지금 녹색등이 켜져있나요?

pedSignal: 잠시만요. 확인 좀 하구요. greenSignal, 너 지금 켜져 있니?

greenSignal: 아뇨.

pedSignal: (민수에게) 녹색등은 꺼져 있네요.

민수: 잠시 기다려야겠군요.

## 횡단보도 2 – 시나리오: 실행 결과

- ✓ “횡단보도” 시나리오는 자연 언어로 표현한 대화체 문장입니다. 이것을 프로그래밍으로 표현할 수 있을까요?
- ✓ 횡단보도1과는 다르게 횡단보도가 양쪽으로 존재합니다. 도로의 어떤 쪽 표현을 RoadSide라고 정합니다.
- ✓ 물론, 프로그램 속의 변수 등 자신들이 하고 있는 것을 아래와 같이 열심히 출력해 주어야 겠지요.
- ✓ 자연 언어 → UML 모델링 언어 → Java 프로그래밍 언어라는 흐름을 지나서 아래와 같은 결과를 보여 주세요.

코딩 실습 →

<director:Director> 시나리오를 시작해 볼까요?

<builder:CrosswalkBuilder> 횡단보도를 건설합니다. cross11  
<cross11:Crosswalk> Add pedestrian signal on --> Left  
<cross11:Crosswalk> Add pedestrian signal on --> Right  
<builder:CrosswalkBuilder> 보행자 신호등을 설치했습니다.  
<builder:CrosswalkBuilder> 횡단보도 건설이 끝났습니다.

<director:Director> minsoo씨, cross11을 왼쪽에서 건너가세요.

<minsoo:Pedestrian> 건너가라구요? 알았어요.

<minsoo:Pedestrian> 반대편 신호등을 확인해야겠군요.

<greenLight:GreenLight> 녹색등이 깨져 있습니다.

<minsoo:Pedestrian> 빨간불이군요.

<minsoo:Pedestrian> 잠시 기다려야겠군요.

<director:Director> 시나리오가 모두 끝났습니다. 감사합니다.

# 횡단보도 2 – 시나리오: 확장 (Roadside)

- ✓ 횡단보도는 여러 장비들이 양쪽에 대칭적으로 존재합니다. 그 개념을 시나리오에 담아봅니다.
- ✓ 왼쪽, 오른쪽의 개념이 존재하므로 표현해야 합니다. “반대편”의 개념의 필요성을 이해해야 합니다.

다음과 같은 표현이 가능해야합니다

- ✓ 횡단보도의 왼쪽 - RoadSide.Left
- ✓ 횡단보도의 오른쪽 - RoadSide.Right
- ✓ 출발하는 쪽의 맞은편 - startingSide.opposite()

```
private boolean isCrossAllowed(RoadSide startingSide, Crosswalk crosswalk) {  
  
    talker.say("건너갈 수 있는지 확인 좀 할까요.");  
    talker.say("반대편 신호등을 확인해야겠군요.");  
  
    PedestrianSignal oppositeSignal = null;  
  
    if (startingSide == RoadSide.Right) {  
        oppositeSignal = crosswalk.getPedestrianSignal(RoadSide.Left);  
    } else if (startingSide == RoadSide.Left) {  
        oppositeSignal =  
            crosswalk.getPedestrianSignal(RoadSide.Right);  
    } else {  
        throw new RuntimeException("No such a road side. --> " +  
            startingSide);  
    }  
  
    if (oppositeSignal.isGreenSignalOn()) {  
        talker.say("파란불이군요.");  
        ...  
    }  
}
```

```
private boolean isCrossAllowed(RoadSide startingSide, Crosswalk crosswalk) {  
  
    talker.say("건너갈 수 있는지 확인 좀 할까요.");  
    talker.say("반대편 신호등을 확인해야겠군요.");  
  
    PedestrianSignal oppositeSignal =  
        crosswalk.getPedestrianSignal(startingSide.opposite());  
  
    if (oppositeSignal.isGreenSignalOn()) {  
        talker.say("파란불이군요.");  
        ...  
    }  
}
```

**public enum RoadSide {**  
 //  
 Left("왼쪽"),  
 Right("오른쪽");  
 ...  
 **public RoadSide opposite() {**  
 if (this == Left) {  
 return Right;  
 } else {  
 return Left;  
 }  
 }  
}

**<<enumeration>>**  
**RoadSide**

Left
Right
Attributes
- krName: String
- RoadSide(String)
+ krName(): String
+ opposite(): RoadSide

## 횡단보도 2 – 시나리오: 확장 (Signal1)

- ✓ 보행자 신호등이 적색등과 녹색등을 속성으로 가지고 있으면 속성 설정하기가 번거로워집니다.
- ✓ 녹색등, 적색등 말고 주황등이 생겨나게 되면 그에 대응하는 객체가 모델링 영역에 존재하지 않습니다.
- ✓ 이는 객체의 협업으로 문제를 해결하라는 객체지향 프로그래밍의 원칙에 위배됩니다.
- ✓ 두 속성을 객체로 승격시켜주고, 보행자신호등은 객체와 메시지를 주고 받도록 해봅시다.

```
public class PedestrianSignal {  
  
    private boolean redSignalOn;  
    private boolean greenSignalOn;  
    ...  
    public boolean isGreenSignalOn(){  
        return greenSignalOn;  
    }  
    public boolean isRedSignalOn() {  
        return redSignalOn;  
    }  
    private void turnRedSignalOn() {  
        redSignalOn = true;  
        greenSignalOn = false;  
    }  
    private void turnGreenSignalOn() {  
        redSignalOn = false;  
        greenSignalOn = true;  
    }  
}
```

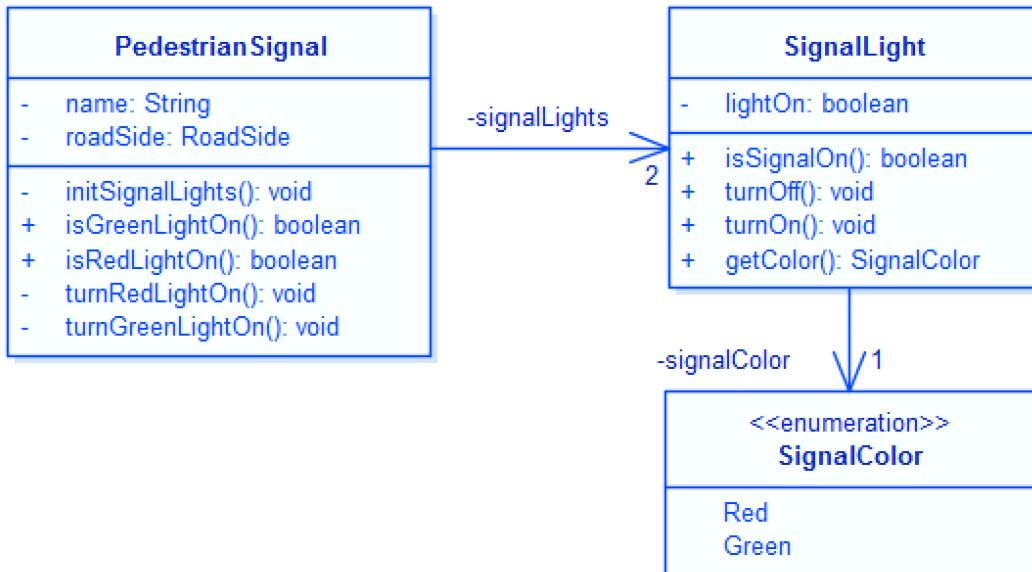


```
public class PedestrianSignal {  
  
    private RedLight redLight;  
    private GreenLight greenLight;  
    ...  
    public boolean isGreenLightOn(){  
        return greenLight.isLightOn();  
    }  
    public boolean isRedLightOn() {  
        return redLight.isLightOn();  
    }  
    private void turnRedLightOn() {  
        redLight.turnOn();  
        greenLight.turnOff();  
    }  
    private void turnGreenLightOn() {  
        redLight.turnOff();  
        greenLight.turnOn();  
    }  
    ...  
}
```

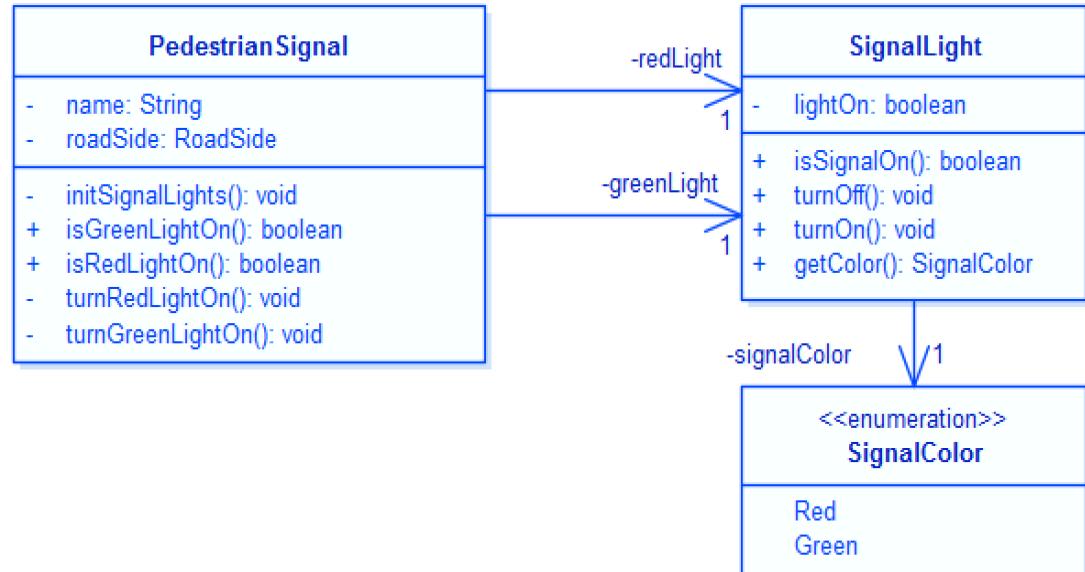
# 횡단보도 2 – 시나리오: 확장 [Signal 2]

- ✓ 적색등과 녹색등을 속성에서 클래스로 옮겨줌으로써 보행자신호등은 자연스럽게 제어할 수 있게 되었습니다.
- ✓ 이 두 개의 등을 클래스로 표현하는 방법에 대해 다양한 의견들이 있습니다.

## 1. 클래스 하나로 설계 (개별 속성 또는 리스트로 설정)



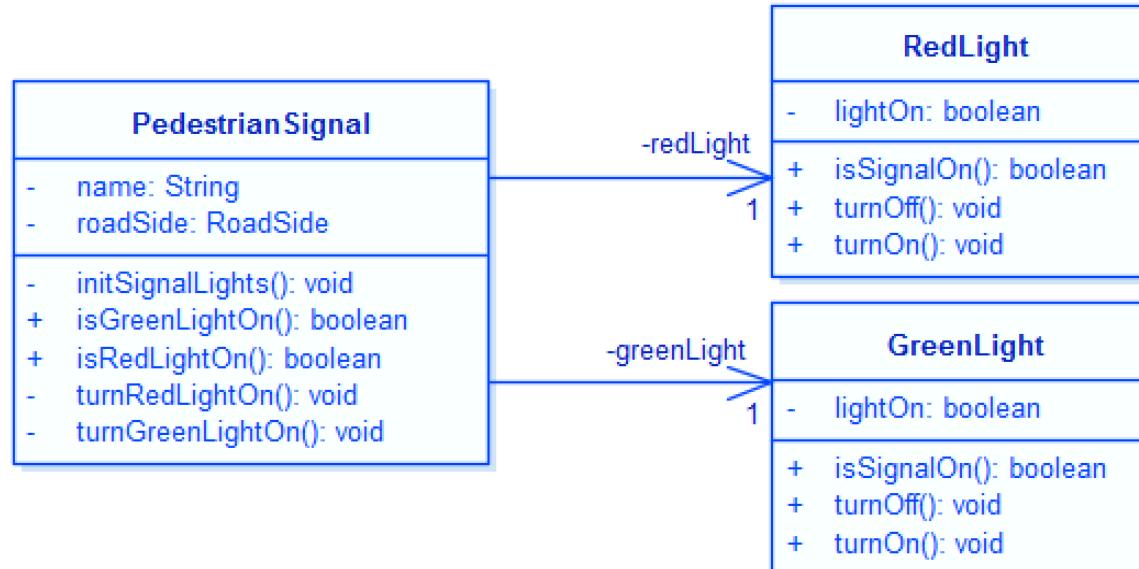
```
class PedestrianSignal {  
    ...  
    private List<SignalLight> signalLights;  
    ...  
}
```



```
class PedestrianSignal {  
    ...  
    private SignalLight redLight;  
    private SignalLight greenLight;  
    ...  
}
```

# 횡단보도 2 – 시나리오: 확장 (Signal 2)

## 2. 클래스 두 개로 설계(개별 속성으로 설정)



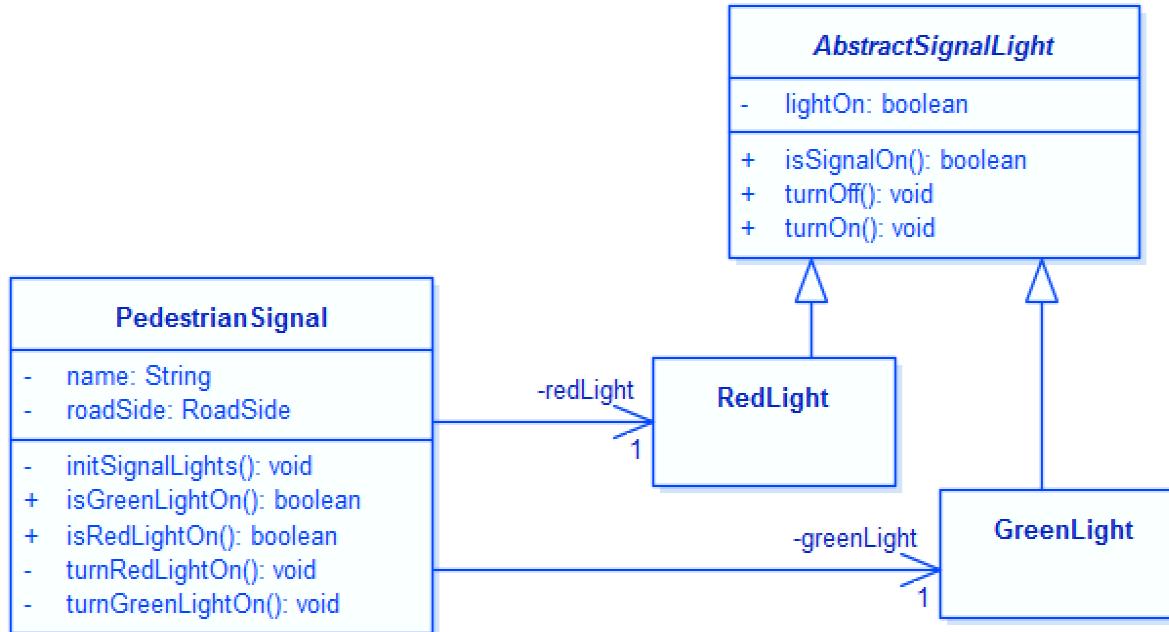
```
class PedestrianSignal {
    ...
    private RedLight redLight;
    private GreenLight greenLight;
    ...
}
```

```
class RedLight {
    ...
    private boolean lightOn;
    public RedLight() {
        this.lightOn = true;
    }
}
```

```
class GreenLight {
    ...
    private boolean lightOn;
    public GreenLight() {
        this.lightOn = true;
    }
}
```

# 횡단보도 2 – 시나리오: 확장 (Signal 2)

## 3. 추상 클래스와 자식 클래스 두 개로 설계(개별 속성 또는 리스트로 설정)



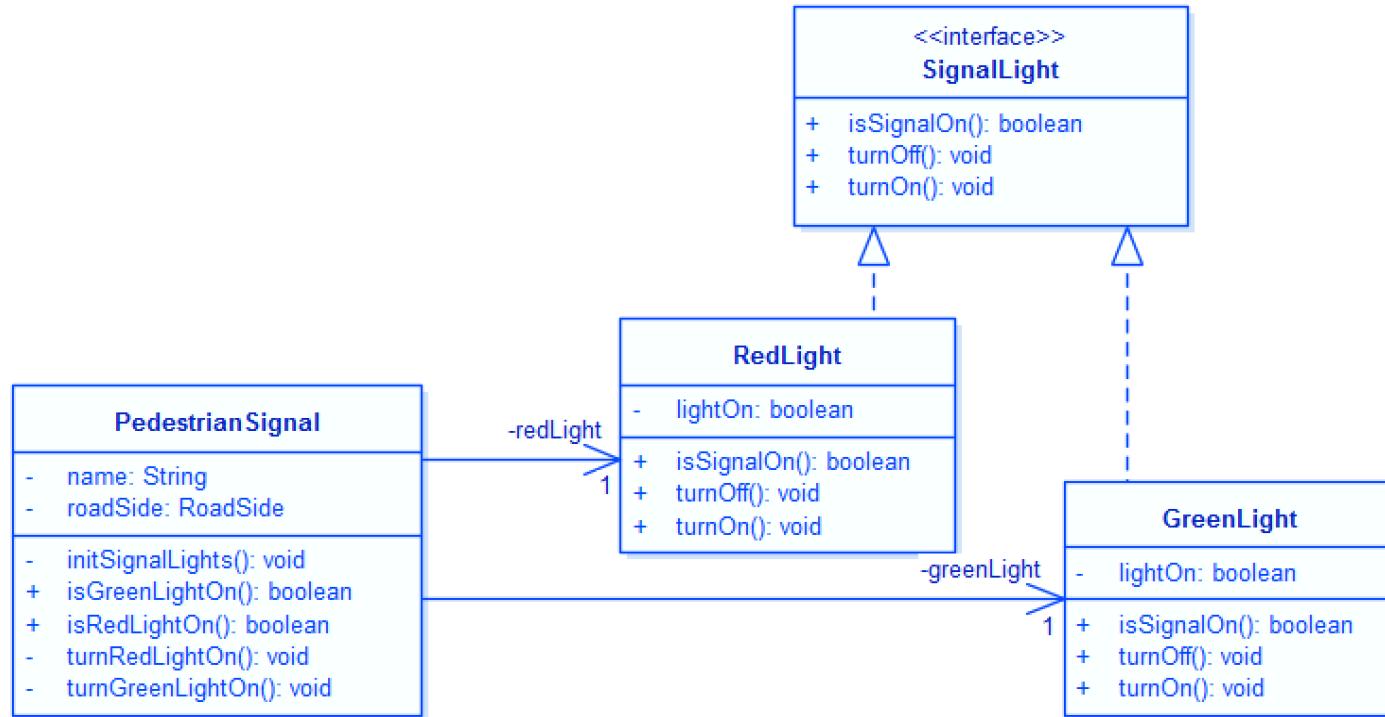
```
class PedestrianSignal {
    ...
    private RedLight redLight;
    private GreenLight greenLight;
    ...
}
```

```
class RedLight {
    public RedLight() {
        super();
    }
}
```

```
class GreenLight {
    public GreenLight() {
        super();
    }
}
```

# 횡단보도 2 – 시나리오: 확장 (Signal 2)

## 4. 인터페이스와 구현 클래스 두 개로 설계 (개발 속성 또는 리스트로 설정)



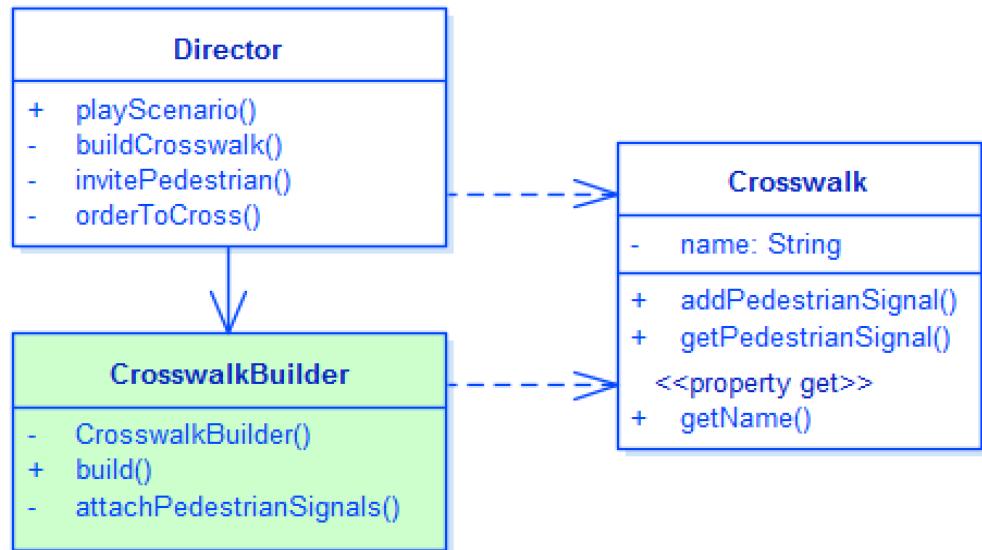
```
class PedestrianSignal {
    ...
    private RedLight redLight;
    private GreenLight greenLight;
    ...
}
```

```
class RedLight implements SignalLight {
    ...
    private boolean lightOn;
    public RedLight() {
        this.lightOn = true;
    }
}
```

```
class GreenLight implements SignalLight {
    ...
    private boolean lightOn;
    public GreenLight() {
        this.lightOn = false;
    }
}
```

# 횡단보도 2 – 시나리오: 확장 (CrosswalkBuilder)

- ✓ 개발을 진행하면서 횡단보도에 보다 많은 장치가 추가될 것이라는 예측은 누구나 할 수 있습니다.
- ✓ 횡단보도를 구축하는 메소드를 클래스로 승격시킴으로써 수많은 "장치 추가" 요구를 수용할 준비를 합니다.
- ✓ 연출가는 전체 시나리오만을 진행하면 되고, 횡단보도 건설은 CrosswalkBuilder가 하면 됩니다.



```
private Crosswalk buildCrosswalk(String name) {
    Crosswalk crosswalk = new Crosswalk(name);

    PedestrianSignal pedestrianSignal = new PedestrianSignal(name);
    crosswalk.setPedestrianSignal(pedestrianSignal);

    return crosswalk;
}
```

```
public class CrosswalkBuilder {
    ...
    public Crosswalk build(String name) {
        Crosswalk crosswalk = new Crosswalk(name);
        this.attachPedestrianSignals(crosswalk);

        return crosswalk;
    }

    private void attachPedestrianSignals(Crosswalk crosswalk) {
        String crosswalkName = crosswalk.getName();

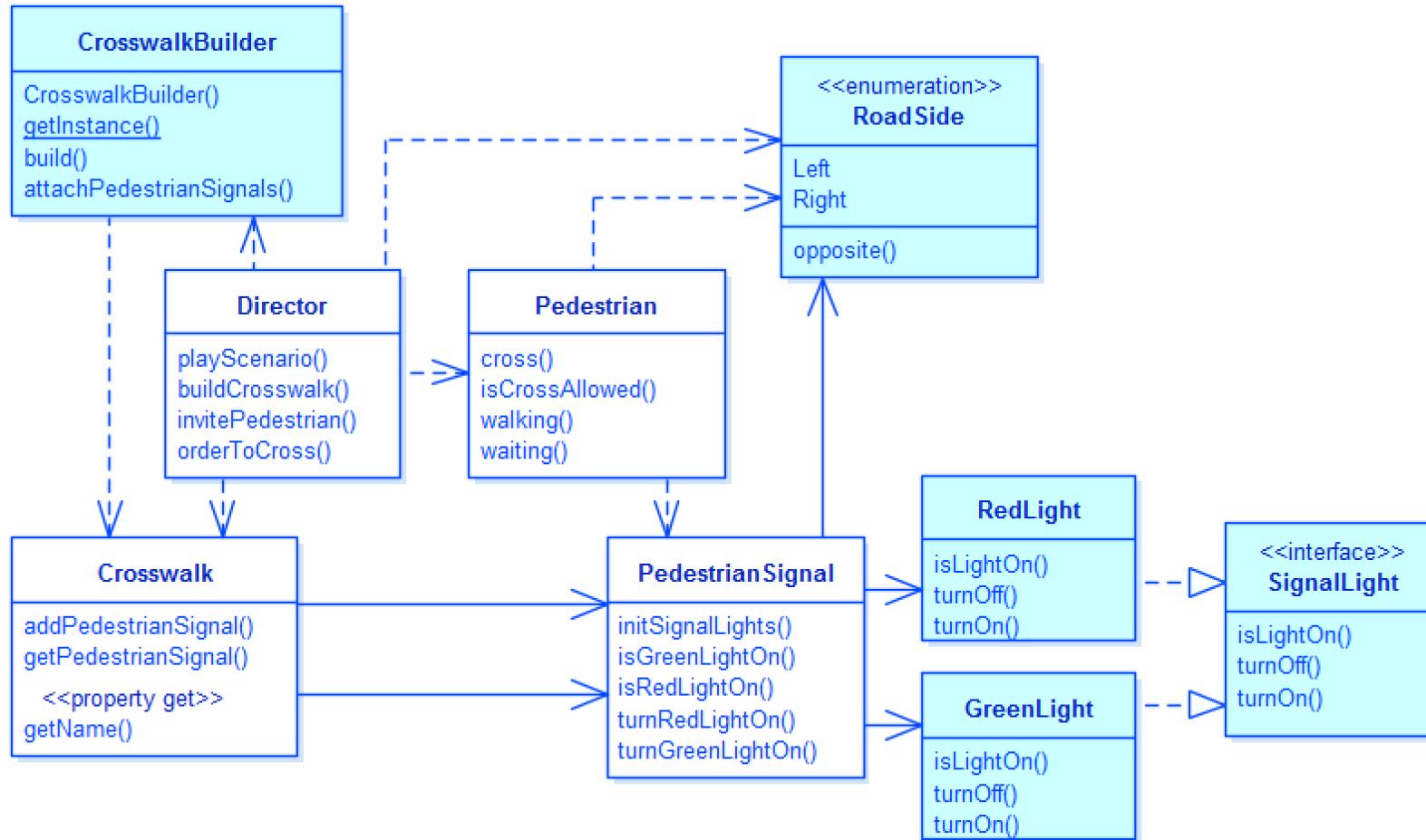
        PedestrianSignal leftLight = new PedestrianSignal(crosswalkName, RoadSide.Left);
        PedestrianSignal rightLight = new PedestrianSignal(crosswalkName, RoadSide.Right);

        crosswalk.addPedestrianSignal(leftLight);
        crosswalk.addPedestrianSignal(rightLight);
    }
}
```

승격

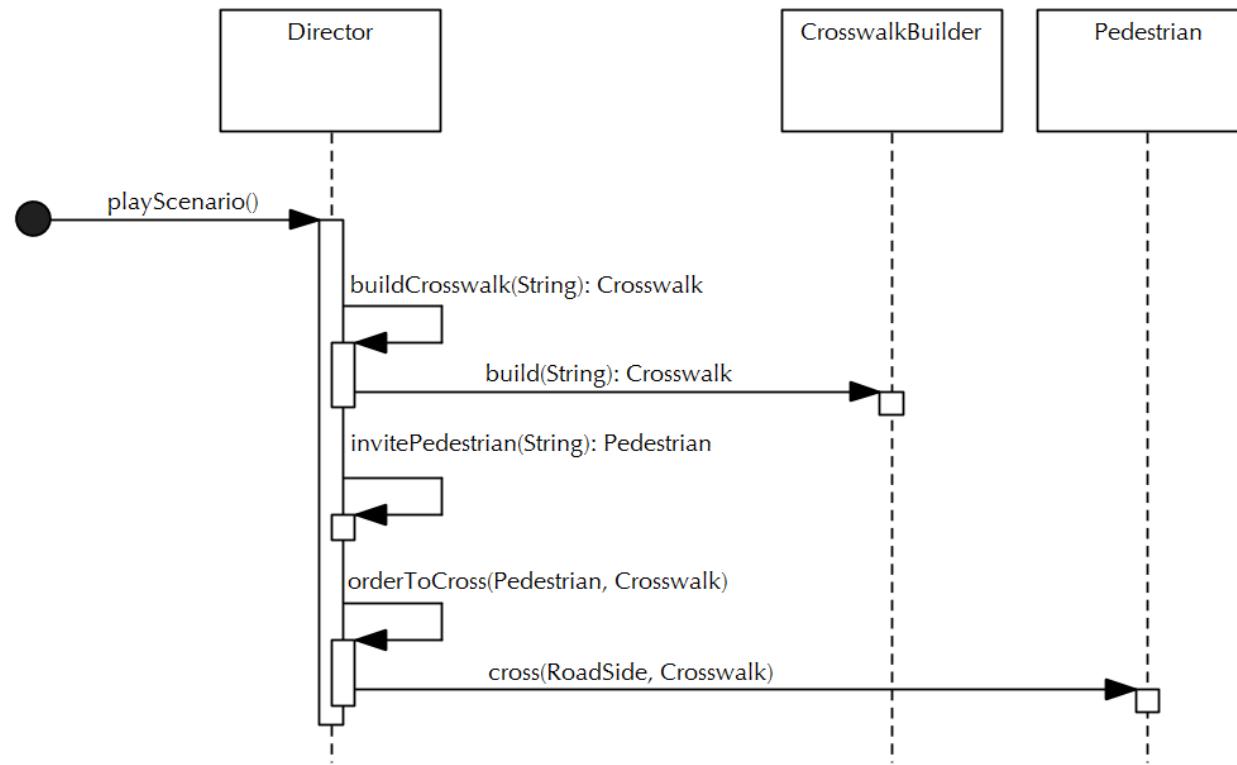
## 횡단보도 2 – 모델: 참여 객체들

- ✓ 속성으로 존재하던 개념을 클래스 수준으로 높여 적색 신호 여부 속성을 RedLight 클래스로 높였습니다.
- ✓ 메소드 수준으로 존재하던 책임을 클래스 수준으로 높여 횡단보도 건설을 CrosswalkBuilder로 높였습니다.
- ✓ 새로 찾아낸 개념인 횡단보도의 양측을 표현하는 정보를 RoadSide 열거형 클래스로 표현하였습니다.



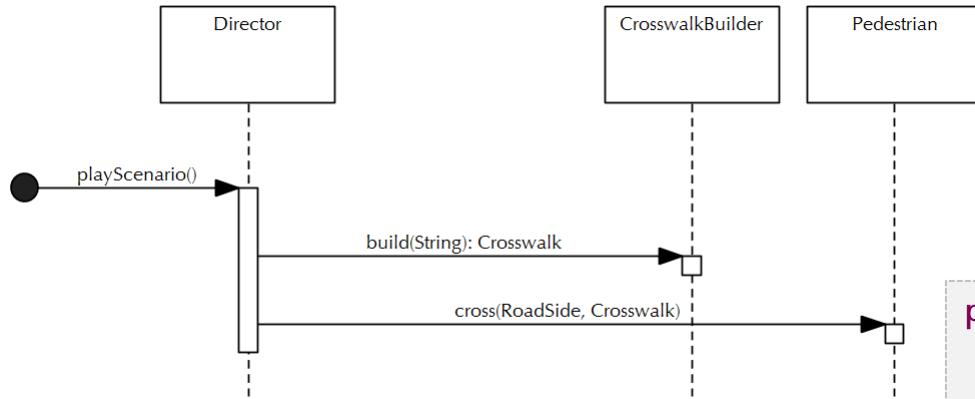
## 횡단보도 2 – 모델: 협업(준비)

- ✓ 시퀀스 다이어그램을 그릴 때 하나의 다이어그램에 전체 시나리오를 모두 표현하는 것은 바람직하지 않습니다.
- ✓ 하나의 시나리오가 복잡해질 경우 다이어그램 관리가 어려워지기 때문에 의미 있는 단위로 나눠서 표현합니다.
- ✓ 책임은 클래스로 나눠져야 하고, 클래스 안에서는 메소드로 나눠져야 합니다.
- ✓ 관심사를 잘 분리하여 설계를 하면 모델 읽기가 편하고 어떤 일을 하는지 명확하게 보입니다.



## 횡단보도 2 – 모델: 협업(준비)

- ✓ `playScenario()` 메소드는 여러 메소드를 순서대로 호출하여 흐름을 구성하고 있습니다.
- ✓ 처리하는 각각의 일들이 복잡해지면 메소드의 라인 수와 복잡도는 아주 빠른 속도로 늘어나는 구조입니다.
- ✓ Director 클래스의 `playScenario()` 메소드는 다른 메소드 호출없이 대부분을 혼자서 처리하므로 세 가지 일(횡단보도 건설, 보행자 초대, 건너기 지시)이 명시적으로 드러나지 않습니다.



```
public void playScenario() {
    // 1. Build a crosswalk named "xingS11".
    String crosswalkName = "cross11";
    Crosswalk crosswalk = CrosswalkBuilder.getInstance().build(crosswalkName);

    // 2. Invite a pedestrian named "minsoo".
    String pedestrianName = "Minsoo";
    Pedestrian pedestrian = new Pedestrian(pedestrianName);

    // 3. Order to cross from left side.
    pedestrian.cross(crosswalk, RoadSide.Left);
}
```

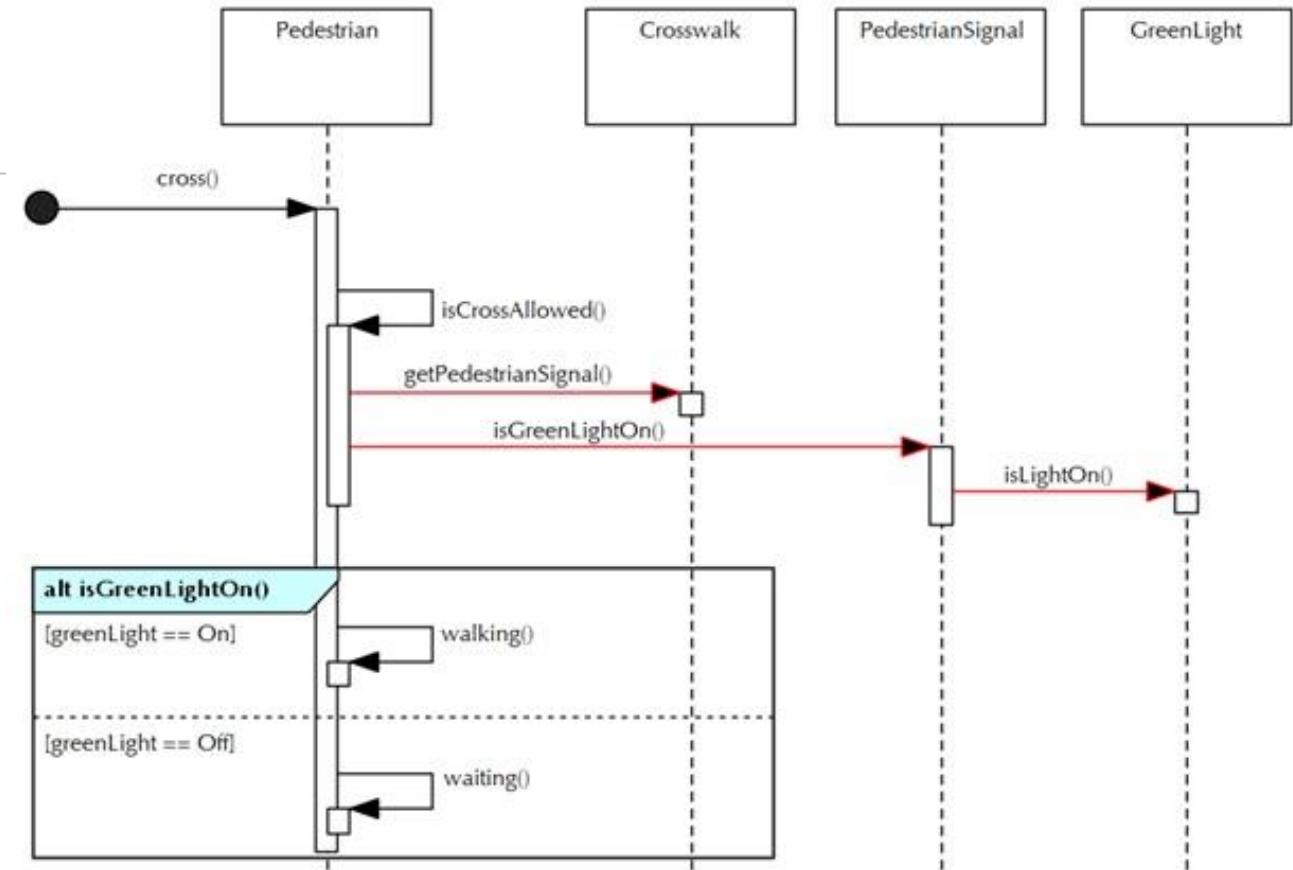
## 횡단보도 2 – 모델: 협업(준비: 건너기)

✓ 보행자는 길을 건너야 할 책임이 있습니다. 아래 시퀀스 다이어그램은 "건너기" 시나리오를 표현합니다.

보행자는 길을 건너야 할 책임이 있습니다.

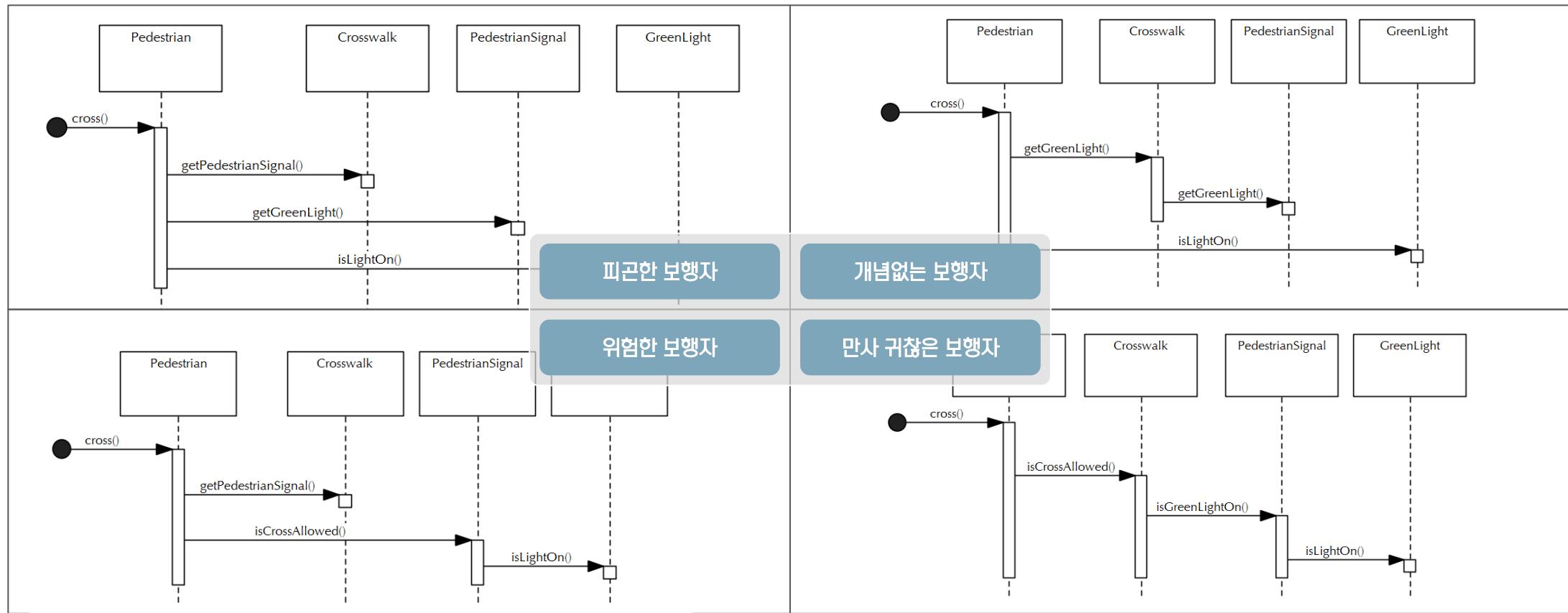
책임을 다하기 위해서 다음 사실을 알고 있어야 합니다.

- ✓ 녹색등에 건널 수 있다.
- ✓ 보행자신호등이 녹색등을 가지고 있다.
- ✓ 횡단보도는 보행자 신호등을 가지고 있다



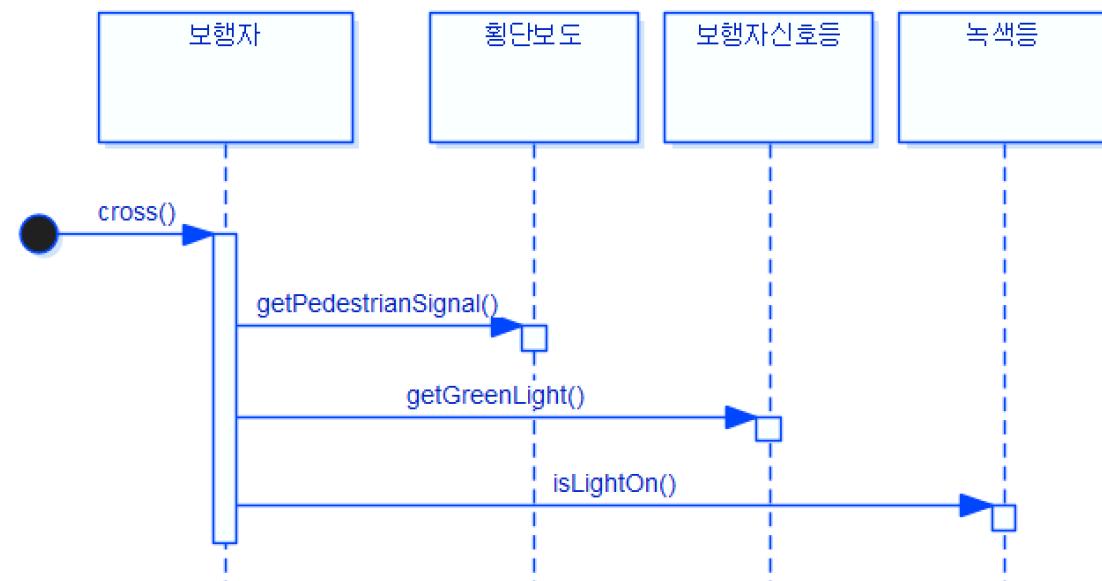
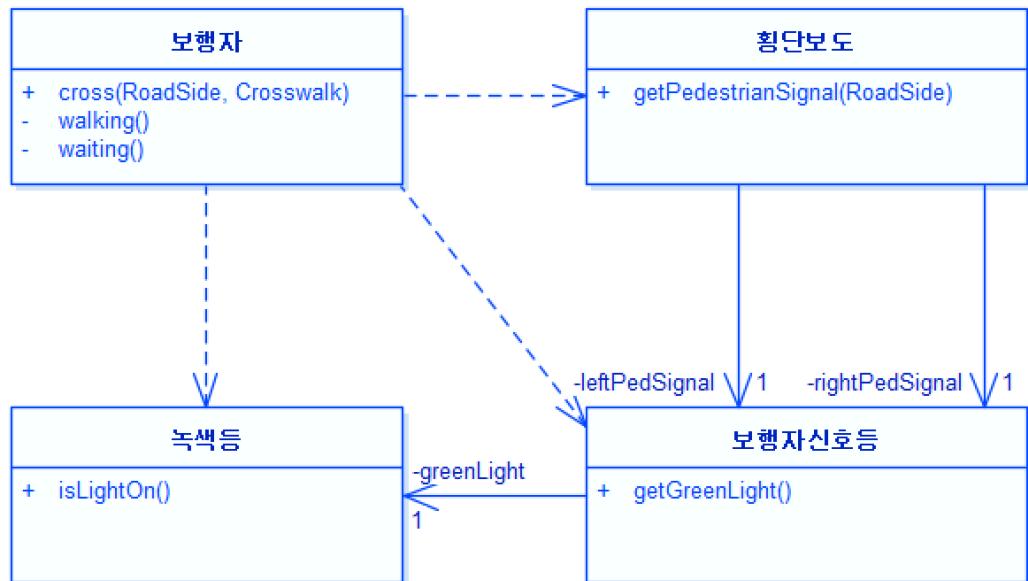
## 횡단보도 2 – 모델: 협업(협업의 다양성)

- ✓ 횡단보도1, 횡단보도2 의 협업 흐름 "건너기"에 어떤 대안들이 있을까요?
- ✓ 시퀀스 다이어그램을 미리 보지 않았다면, 여러분은 앞으로 보여 줄 모델 중 하나를 선택했을지도 모릅니다.
- ✓ 아래의 프로그램은 이상없이 동작합니다. 하지만 협업에 논리적인 오류가 있습니다.
- ✓ 실행된다고 모두 이상이 없는 것이 아닙니다.



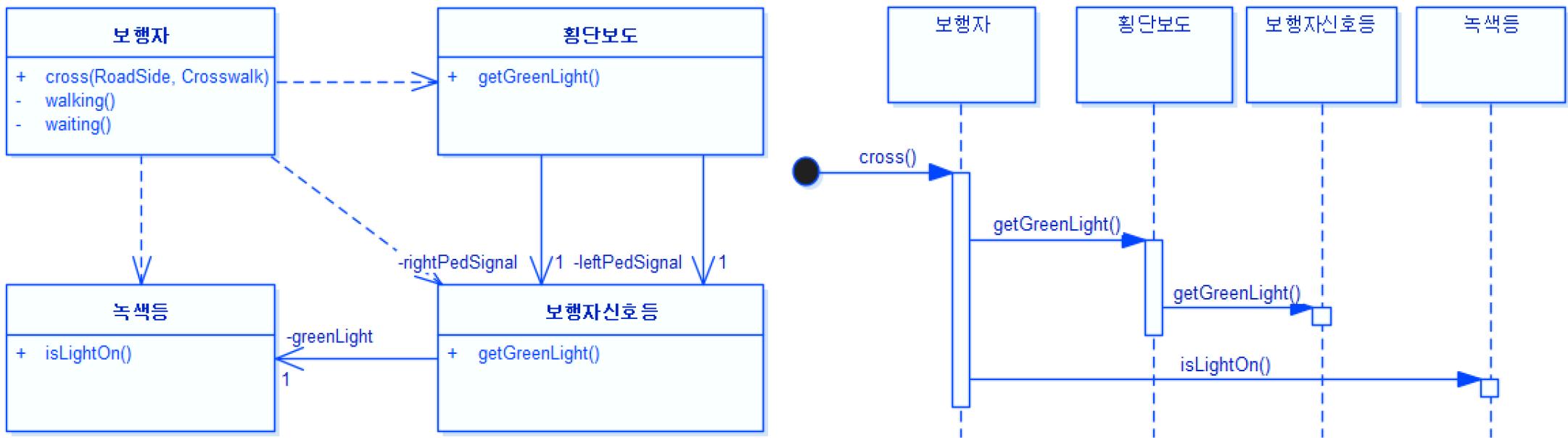
## 횡단보도 2 – 모델: 협업 – 피곤한 보행자

- ✓ 매우 피곤한 보행자 시나리오입니다. 이 보행자는 모든 것을 알고 있어서 피곤합니다.
- ✓ 횡단보도가 보행자 신호등을 갖고 있다는 사실과 보행자 신호등이 녹색등을 갖고 있음을 모두 알고 있습니다.
- ✓ 녹색등을 찾아서 녹색등에서 점등여부(isLightOn())를 물어 보는 방식으로 문제를 해결합니다.



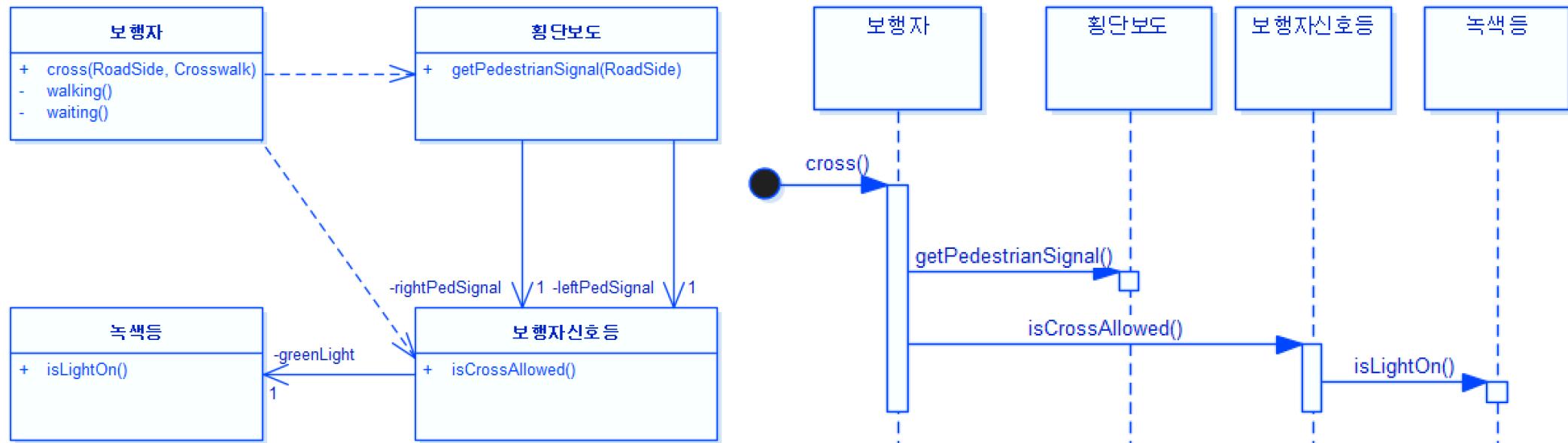
## 횡단보도 2 – 모델: 협업 – 개념없는 보행자

- ✓ 이번 보행자는 횡단보도 컨텍스트에 대한 이해도 없고 참여 객체들에 대한 배려도 없습니다.
- ✓ 횡단보도에게 녹색등을 달라고 합니다. 횡단보도는 여기서 무대를 제공하는 “큰” 어른과 같은 객체입니다.
- ✓ 앞으로 녹색등과 같은 작은 장치들이 많이 장착될 텐데, 횡단보도가 그런 것을 신경쓸 겨룰이 있을까요?



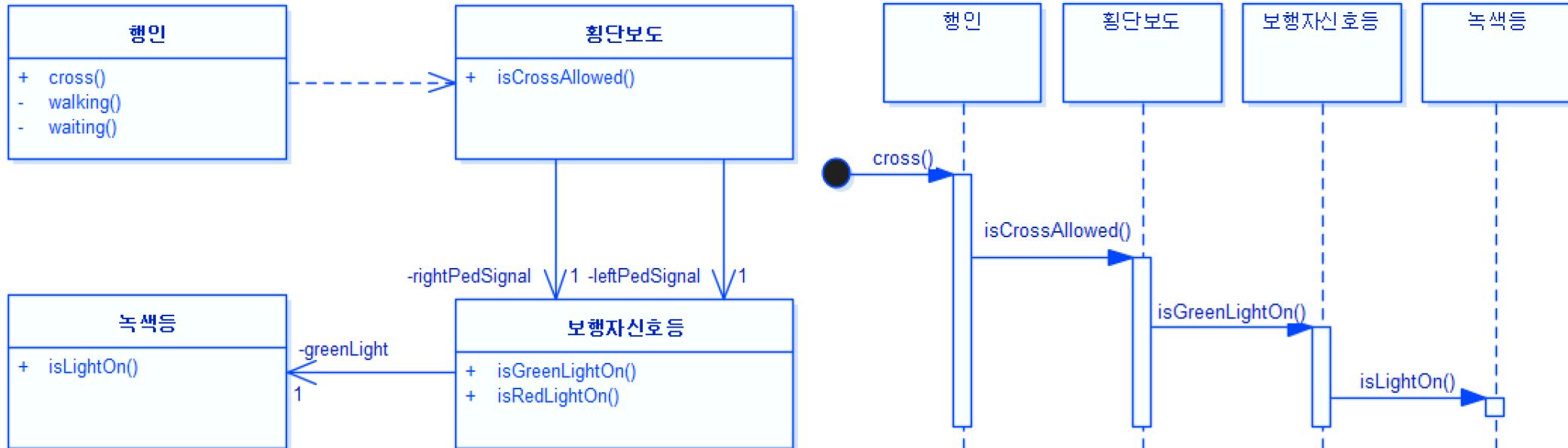
## 횡단보도 2 – 모델: 협업 – 위험한 보행자

- ✓ 이번 보행자는 보행자 신호등에게 “건널 지 여부 판단”을 맡기는 아주 위험하고 대담한 경우입니다.
- ✓ 보행자 신호등에 오류가 있을 경우, 이 보행자는 생명이 위태로울 수 있습니다.
- ✓ 객체는 자신이 무엇을 알아야 하고, 어떤 판단을 해야 하는지 정확히 알아야 합니다. R&R이 중요합니다.



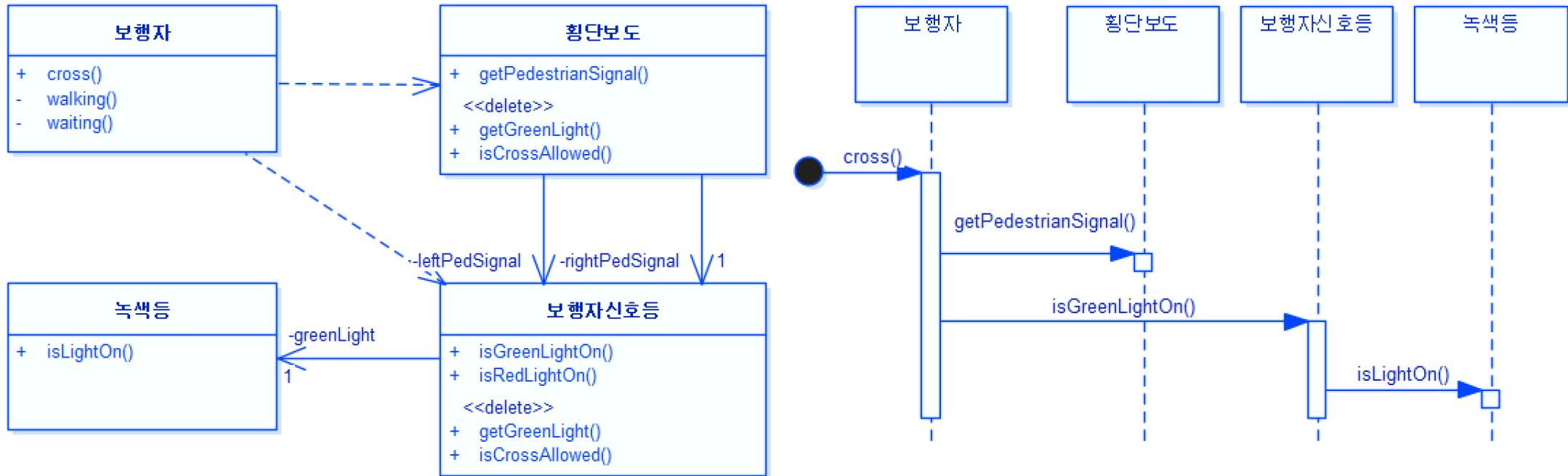
## 횡단보도 2 – 모델: 협업 – 만사 귀찮은 보행자

- ✓ 이번 보행자는 횡단보도에 대고 “건너요 말아요?”라고 물어 보는 만사가 귀찮은 개발자입니다.
- ✓ 횡단 보도 내부 구성에는 관심이 전혀 없으며, 신호등에도 관심이 없습니다.
- ✓ 이런 객체는 컨텍스트에 참여하는 구성원들에 대한 이해가 부족한 객체로, 그 행동을 예측할 수 없습니다.



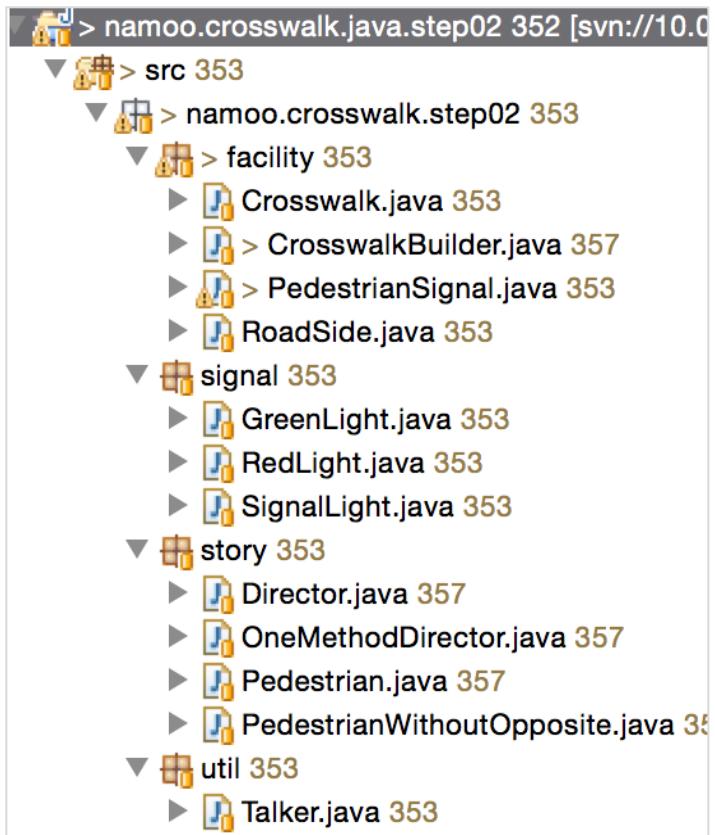
## 횡단보도 2 – 모델: 협업 – 멋진 보행자

- ✓ 횡단보도 컨텍스트를 잘 이해하는 보행자는 누구로부터 “건너기 여부” 정보를 얻어야 할지 잘 알고 있습니다.
- ✓ 객체 관계도 적절하게 조절을 하여, “녹색등”은 알 필요가 없으므로 관계를 단순하게 유지합니다.
- ✓ 무엇보다 녹색등의 소유권을 가진 보행자신호등의 존재 가치를 충분히 인정하여 주고 있습니다.



## 횡단보도 2 – 모델: Name space 확장

- ✓ 각 시나리오를 설계하는 과정에서 새로운 클래스 여러 개가 등장하였습니다.
- ✓ 각 클래스들을 유사한 역할별로 분류하여 공간을 정하고 이름(package)을 부여합니다.



분류	의미
facility	횡단보도 시설에 해당하는 클래스들
signal	신호와 관련된 클래스들
story	시나리오 진행 관련 클래스들
util	유틸리티 클래스들

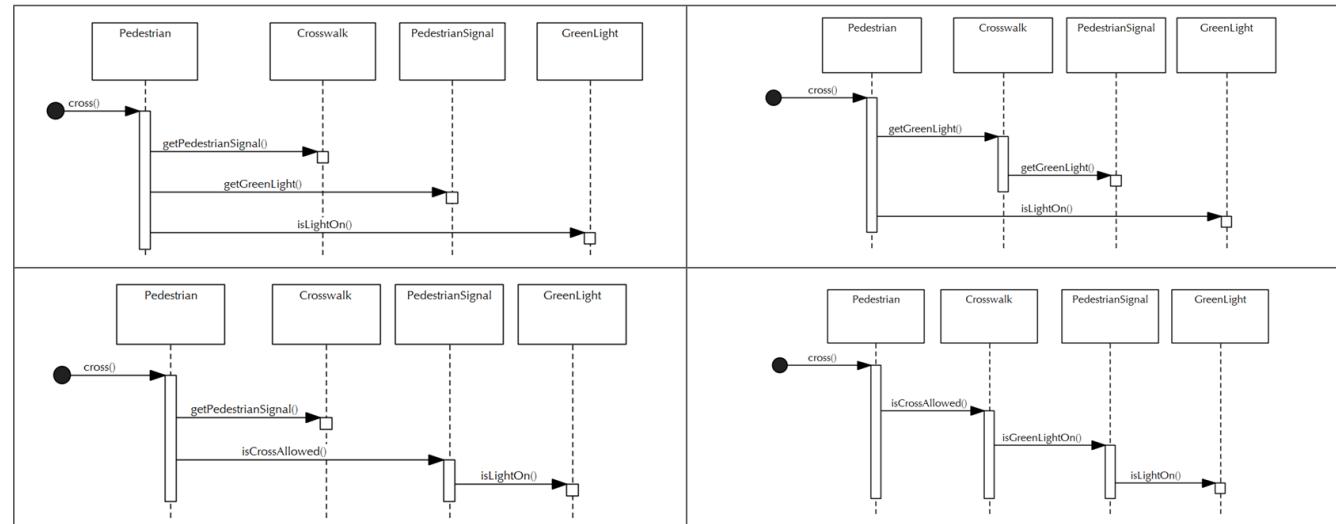
## 횡단보도 2 – 프로그래밍

- ✓ 새로운 객체들이 등장합니다. 특히 개념을 명확하게 하거나 확장하는 객체들이 등장합니다.
- ✓ 각 단계별로 명확하게 하는 활동과 식별하는 활동을 계속합니다.
- ✓ 개발 활동은 스케치 → 서술 → 모델링 → 프로그래밍 과정을 반복하는 과정입니다.

```
public enum RoadSide {  
    //  
    Left("왼쪽"),  
    Right("오른쪽");  
  
    private String krName;  
  
    private RoadSide(String krName) {  
        this.krName = krName;  
    }  
  
    public String krName() {  
        return krName;  
    }  
  
    public RoadSide opposite() {  
        //  
        if (this == Left) {  
            return Right;  
        } else {  
            return Left;  
        }  
    }  
}  
  
public class GreenLight implements SignalLight {  
    //  
    private String name;  
    private boolean lightOn;  
  
    private Talker talker;  
  
    public GreenLight(String parentName){  
        //  
        this.name = "greenLight";  
        this.lightOn = false;  
        this.talker = Talker.newInstance(name, this);  
    }  
  
    @Override  
    public boolean isLightOn(){  
        //  
        if (lightOn) {  
            talker.sayAtRight("녹색등이 켜져 있습니다. ");  
        } else {  
            talker.sayAtRight("녹색등이 꺼져 있습니다. ");  
        }  
  
        return lightOn;  
    }  
  
    @Override  
    public void turnOff(){  
        lightOn = false;  
    }  
}
```

## 횡단보도 2 – 요약

- ✓ 원하는 수준의 시나리오 실행 결과를 확인하고, 리뷰하고 나서 다음 단계로 넘어갑니다.
- ✓ 새로운 개념을 많이 찾거나 명확하게 정의하였습니다.
- ✓ 이미 개발한 내용 중에 대안 시나리오가 있는지 확인하고, 각 대안을 검토하였습니다.
- ✓ 이제 다음으로 넘어갈 차례입니다.





## 모듈: 협업과 추상화

- ❖ 복잡한 대상을 단계적으로 다루는 방법을 이해합니다.
- ❖ 참여 객체 식별, 역할과 책임 식별, 협업 구조 결정 등을 진행합니다.
- ❖ 지속적으로 시나리오를 확장하는 방법을 배웁니다.

## Crosswalk 3

- ✓ 시나리오
- ✓ 모델
- ✓ 프로그래밍
- ✓ 요약

# 횡단보도 3 – 시나리오

- ✓ 시간이 흐르면 신호등이 파란 불로 바뀌고 행인은 건너가는 시나리오를 전개합니다.
- ✓ 횡단보도 1,2로 부터 끝 부분을 확장하여 횡단보도3 시나리오를 전개합니다.

(막이 열리면 11번가의 모습이 보이고, 무대 한 가운데 횡단보도가 있다. 보행자인 민수가 횡단보도 왼쪽에서 등장한다.)

연출가 : 민수씨, 11번가 첫 번째 횡단보도인 xingS11을 왼쪽에서 건너세요.

민수 : 아, 이 횡단보도요? 알겠습니다. 먼저 보행자 신호등을 확인해야 겠는데요. xingS11, 내 맞은 편 보행자신호등 좀 주세요.

xingS11 : (오른쪽 보행자신호등을 건네며) 여기 있어요.

민수 : 보행자신호등, 지금 녹색등이 켜져있나요?

우측보행자신호등 : 잠시만요. 확인 좀 하구요. 녹색등, 지금 켜져 있나요?

녹색등 : 아뇨.

우측보행자신호등 : (민수에게) 녹색등은 깨져있네요.

민수 : 잠시 기다려야겠네요.

... (잠시 시간이 흐르고, 녹색등이 켜진다.)

민수 : 보행자신호등, 지금 녹색등이 켜져있나요?

우측보행자신호등 : 잠시만요. 확인 좀 하구요. 녹색등, 지금 켜져 있나요?

녹색등 : 예.

우측보행자신호등 : (민수에게) 녹색등 켜져있습니다.

민수 : 건너야지. (횡단보도를 가로질러 건너간다.)

연출가 : 수고했습니다. 시나리오를 마칩니다.

# 횡단보도 3 – 시나리오

- ✓ 신호장치들이 신호등을 변경하기 위해 서로 협업하는 모습을 보여줍니다.
- ✓ 보행자가 볼 수 없는 뒤쪽에서 시나리오가 조용히 진행되고 있는데 카메라를 들이대고 있는 상황입니다.
- ✓ 신호제어를 위해서 누가 있어야 하는지, 등장인물에 어떤 책임을 부여할지 등을 고민해봅시다.

코딩 실습 →

(막이 열리면 무대 위에 여러 장치들이 놓여 있다. 신호제어기, 타이머가 있고, 각 보행자신호등 별로 녹색등, 적색등 두 개가 달려있다.)

연출가 : 민수씨, 11번가 첫 번째 횡단보도인 xingS11을 왼쪽에서 건너세요.

... 위의 진행을 반복한다. (30초가 지났다.)

신호제어기 : 자 지금부터 30초를 기준시간으로 하여 신호등을 변경할 겁니다.

타이머 : 신호제어기 30 초 지났습니다.

우선 보행자 신호등, 신호등을 초기화하세요.

신호제어기 : 아, 그래요. 우리의 기준 시간이 30초이니, 0초 남은 셈이로군요.

보행자신호등 : 예, 초기화합니다. 녹색등을 끄고, 적색등은 켰습니다. 이제 준비되었어요.

좌측 보행자신호기, 0초 남았습니다. 우측 보행자신호기, 0초 남았습니다.

신호제어기 : 타이머는 지금부터 1초 단위로 경과시간을 통지해 주세요.

좌측보행자신호기 : 아, 그래요. 시간이 다 되었군요. 신호등을 변경해야지.

타이머 : 예, 신호제어기, 1 초 지났습니다.

녹색등, 지금 켜져있나요?

신호제어기 : 아, 그래요. 우리의 기준 시간이 30초이니, 29초 남은 셈이로군요.

좌측녹색등 : 아뇨 꺼진상태인데요.

왼쪽 보행자 신호기, 29초 남았습니다. 오른쪽 보행자신호기, 29초 남았습니다.

좌측보행자신호기 : 그래요. 그럼 녹색등은 켜구요, 적색등은 끄세요.

좌측보행자신호기 : (아무런 반응이 없다.) 아, 그래요. 29초 남았군요.

좌측녹색등 : 예, 켰습니다.

우측보행자신호기 : (아무런 반응이 없다.) 아, 그래요. 29초 남았군요.

좌측적색등 : 예, 껐습니다.

우측보행자신호기 : 그래요. 그럼 녹색등은 켜구요, 적색등은 끄세요.

우측보행자신호기 : 아, 그래요. 0초나 남았다고요. 신호등 변경해야겠네요.

우측녹색등 : 예, 켰습니다.

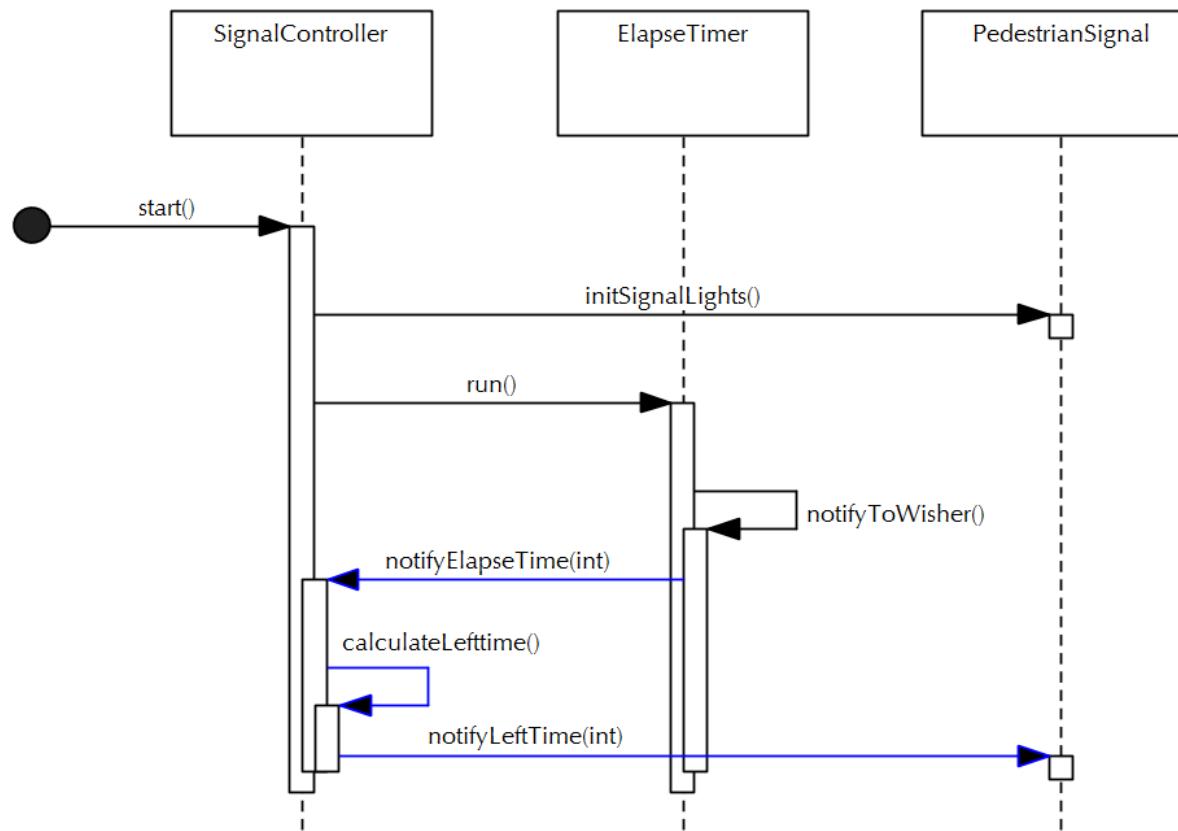
녹색등 지금 켜져있나요?

우측적색등 : 예, 껐습니다.

우측녹색등 : 아뇨 꺼진상태인데요.

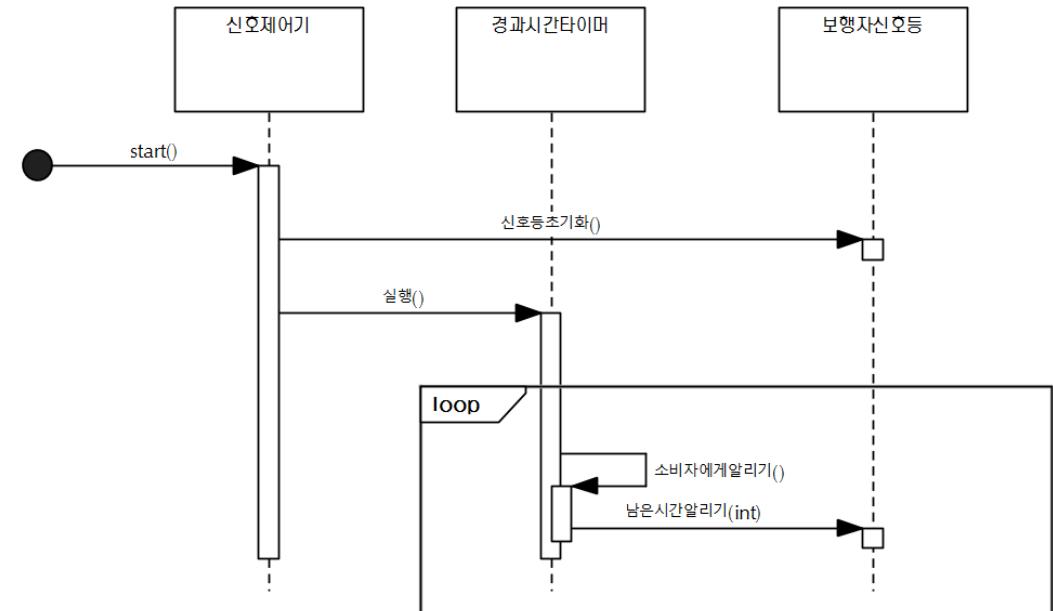
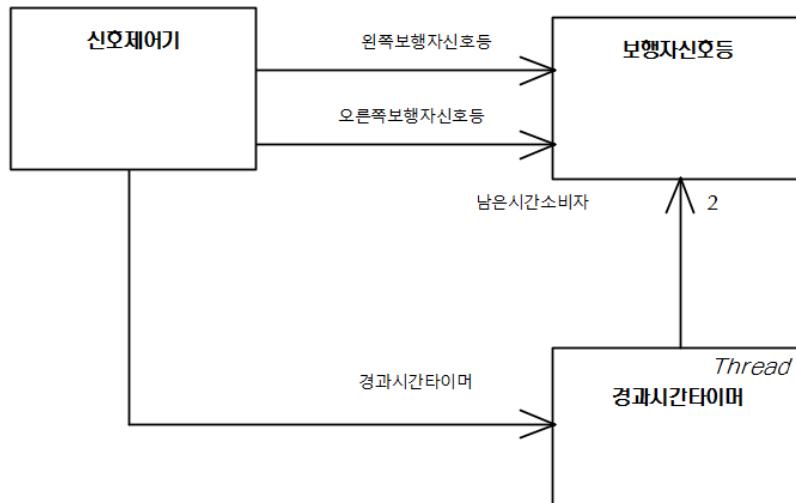
## 횡단보도 3 – 시나리오: 확장 (시간 개념)

- ✓ 모델링의 핵심은 도메인 안에 보이지 않는 곳에 숨어있는 "본질적인 개념"을 찾아서 정의하고 반영합니다.
- ✓ 보행자는 녹색등, 음성 안내기 등 모든 신호등이 변경되기까지 "남은 시간(left time)"을 기준으로 동작합니다.
- ✓ 남은 시간을 계산하려면 기준 시간을 알고 있어야 합니다. 이 문제를 해결할 적임자는 누구일까요?



# 횡단보도 3 – 시나리오: 확장 (누가 시간을 알려주나)

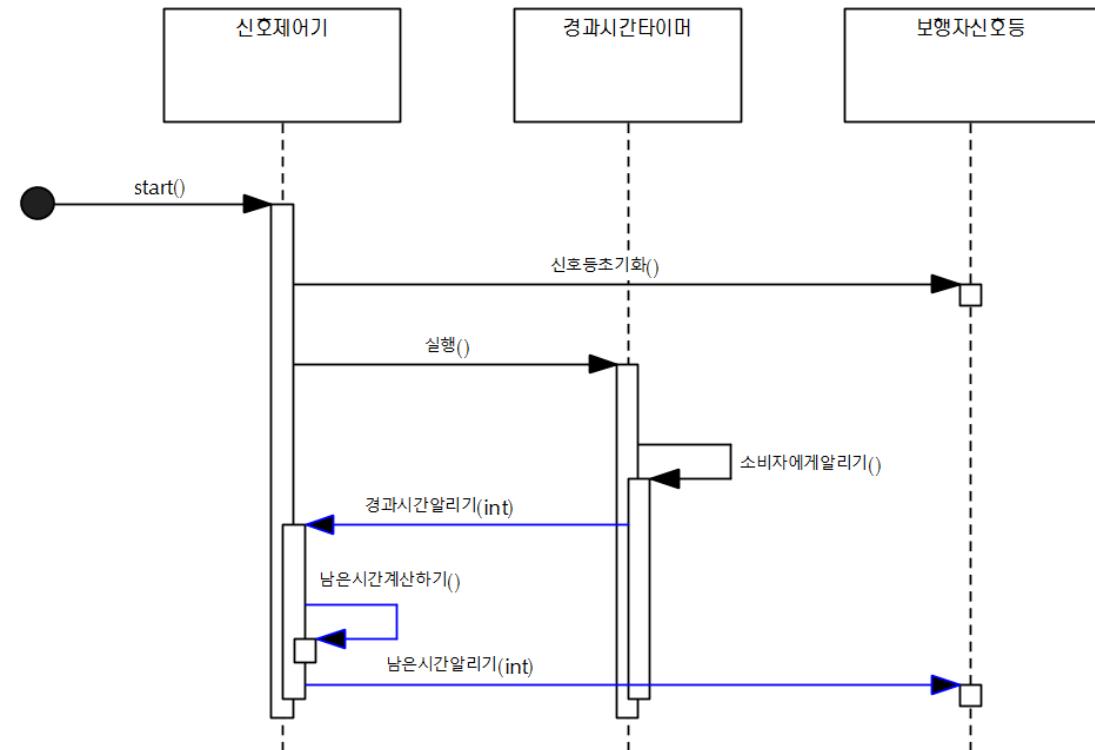
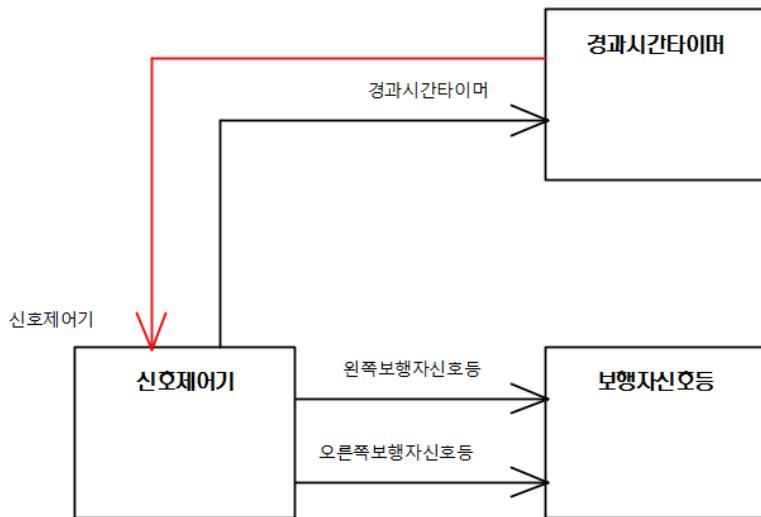
- ✓ 객체 모델링은 정보를 다루는 시스템에서 정보의 생산과 공급, 소비를 둘러싼 이해관계를 잘 조절하는 것입니다.
- ✓ 생산자와 소비자가 직접 거래하도록 설계를 해봅시다.



```
private void notifyLeftTime(){  
    int leftTime = this.baseTime - this.elapsedTime;  
  
    for(LeftTimeWisher leftTimeWisher : this.leftTimeWishers) {  
        leftTimeWisher.notifyLeftTime(leftTime);  
    }  
}
```

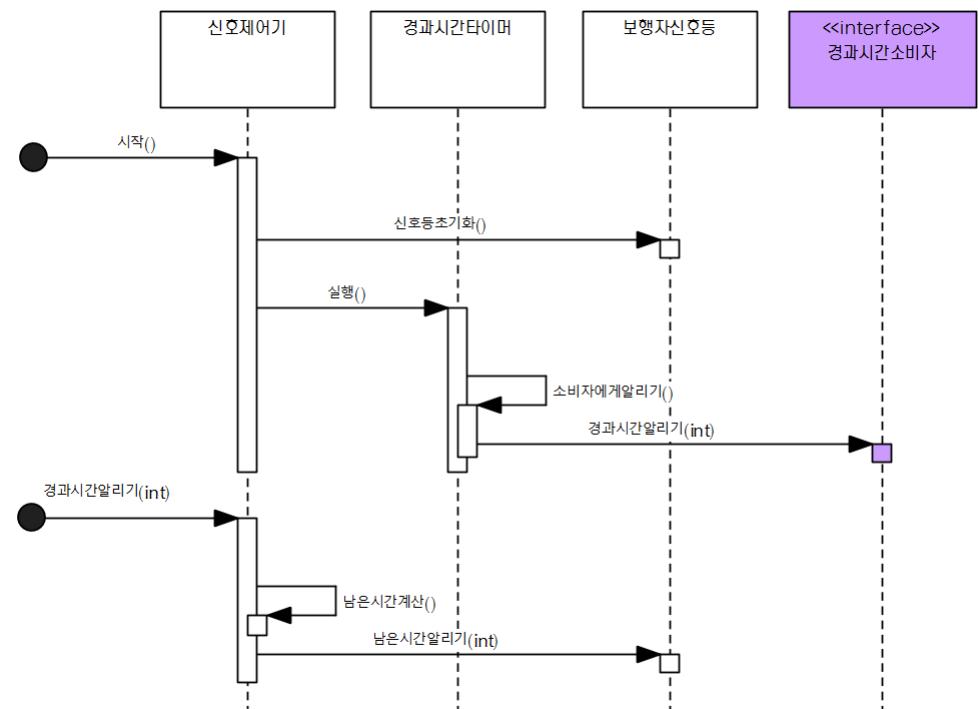
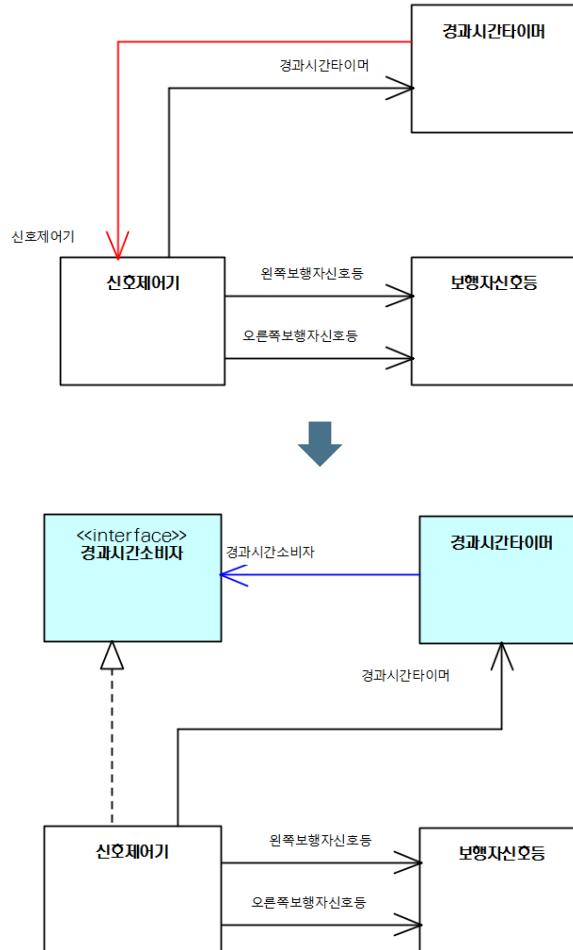
## 횡단보도 3 – 시나리오: 확장 (누가 시간을 알려 주나)

- ✓ 타이머가 직접 시간을 공급할 때의 문제는 관계가 많이 만들어진다는 것입니다.
- ✓ 객체들 간의 관계를 적게 유지하면서 시간을 공급하는 방법을 생각해봅시다.



# 횡단보도 3 – 시나리오: 확장 (전용타이머)

- ✓ 경과시간 타이머는 신호제어기를 가지고 있고 다른 곳에 가서 변경 없이는 사용할 수 없습니다.
- ✓ 어디서든지 사용할 수 있는 범용 타이머로 설계 해봅시다.



## 횡단보도 3 – 시나리오: 확장 (전용타이머)

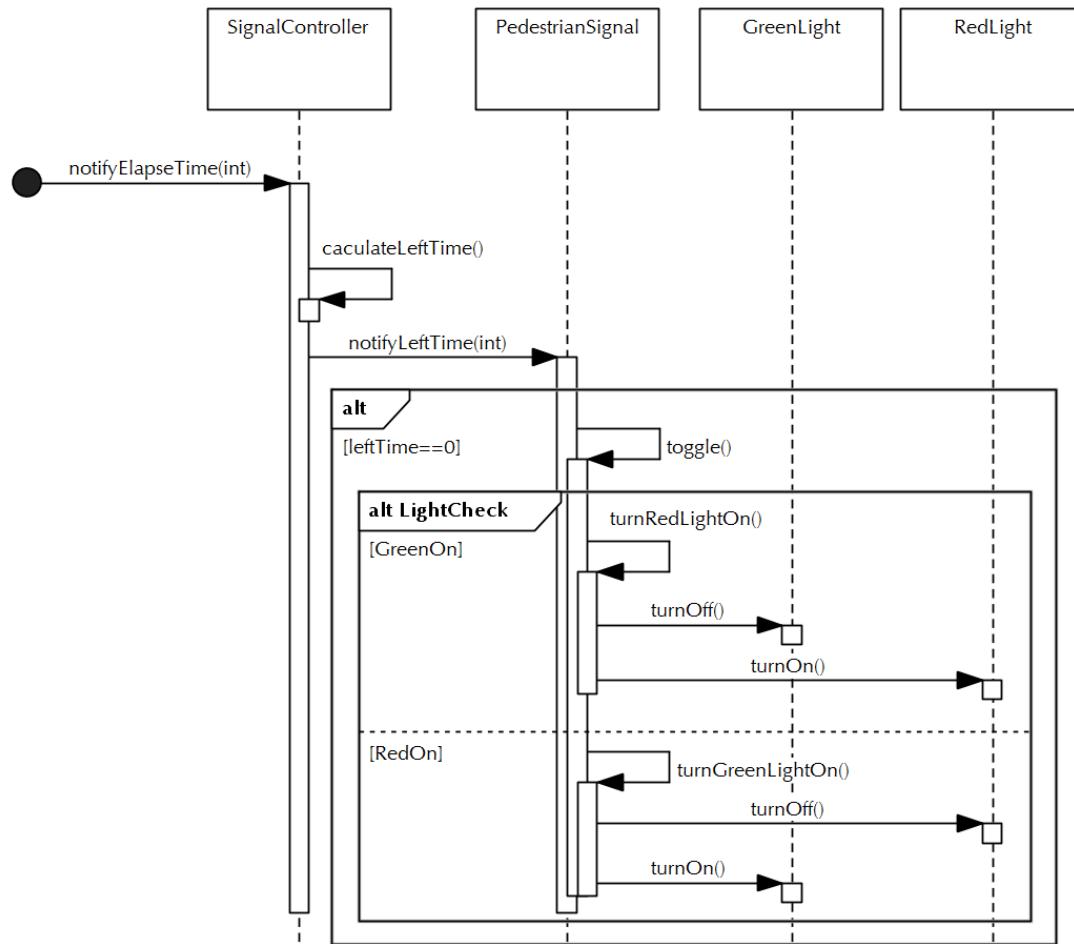
- ✓ 타이머는 `elapseTimeWisher`라는 인터페이스를 구현한 객체로 경과시간을 알려주고 있습니다.
- ✓ 대상이 변경되더라도 `elapseTimeWisher`로 등록해 준다면 타이머는 원활하게 시간을 제공해 줄 것입니다.
- ✓ 남은 시간을 왼쪽 오른쪽 두 개의 보행자신호등에게 보내주는 것을 `ElapseTimeWisher` 통해 볼 수 있습니다.

```
public class ElapseTimer extends Thread {  
  
    private int baseTime;  
    private int newBaseTime;  
    private int elapseTime;  
  
    private SignalController signalController;  
  
    public ElapseTimer(SignalController signalController,  
                      int baseTime){  
        super("goodTimer");  
        ...  
    }  
    ...  
    private void notifyToWisher(){  
        signalController.notifyElapseTime(elapseTime);  
    }  
    ...  
}
```

```
public class SignalController implements ElapseTimeWisher {  
    ...  
    @Override  
    public void notifyElapseTime(int elapseTime) {  
        leftTime = baseTime - elapseTime;  
  
        leftPedestrianSignal.notifyLeftTime(leftTime);  
        rightPedestrianSignal.notifyLeftTime(leftTime);  
    }  
    ...  
}
```

# 횡단보도 3 – 시나리오: 확장 (누가 신호등을 끄고 켜나?)

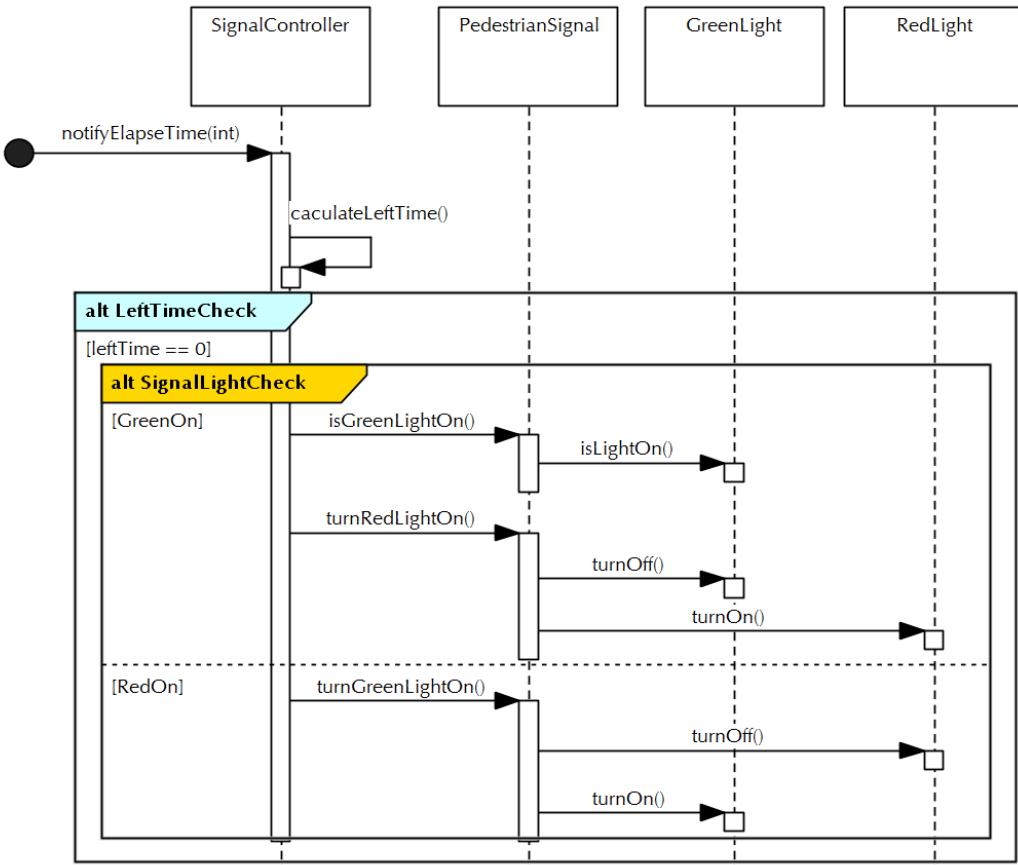
- ✓ 보행자신호등이 적색등과 녹색등 조작을 주도하고 있는 시나리오를 살펴봅시다.



```
public class PedestrianSignal {  
    ...  
    public void notifyLeftTime(int leftTime) {  
        if (leftTime == 0) {  
            toggleLight();  
        }  
    }  
    private void toggleLight() {  
        if (redLight.isLightOn()) {  
            turnGreenLightOn();  
        } else {  
            turnRedLightOn();  
        }  
    }  
    public void turnRedLightOn() {  
        redLight.turnOn();  
        greenLight.turnOff();  
    }  
    public void turnGreenLightOn() {  
        redLight.turnOff();  
        greenLight.turnOn();  
    }  
    ...  
}
```

# 횡단보도 3 – 시나리오: 확장 (누가 신호등을 끄고 켜나?)

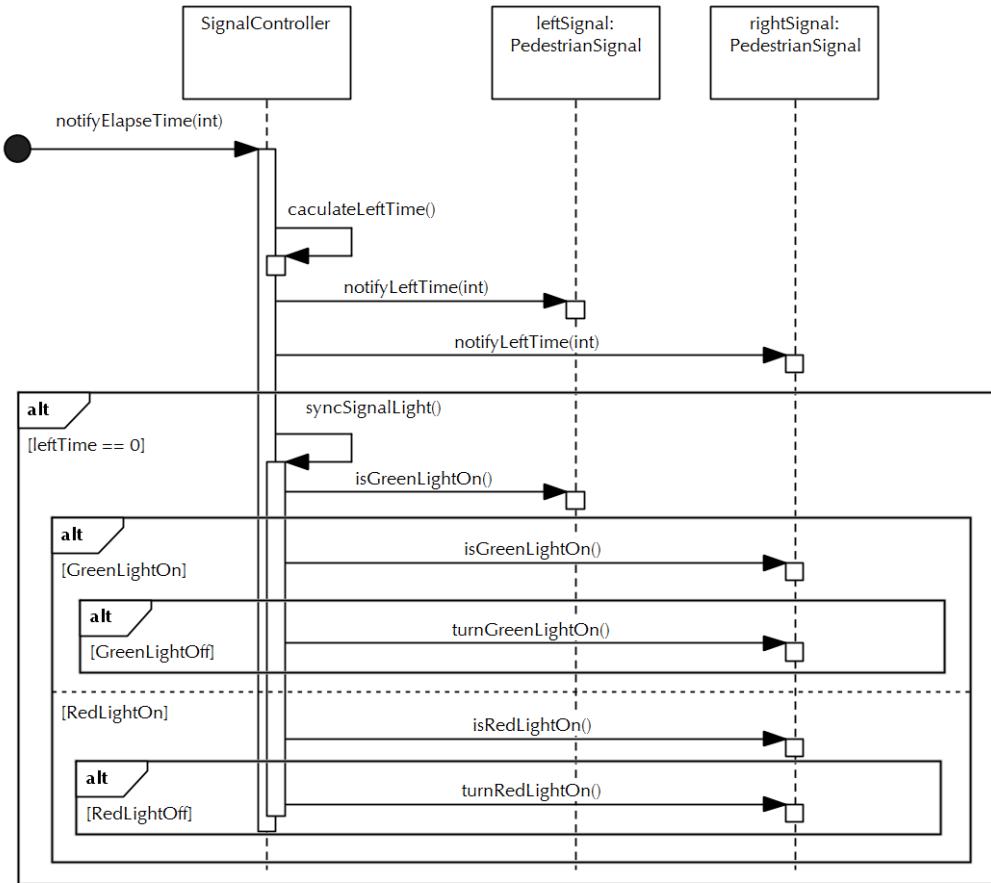
- ✓ 신호제어기가 주장하는 시나리오를 살펴 봅시다.



```
public class SignalController {  
    ...  
    public void notifyElapseTime(int elapseTime) {  
        leftTime = baseTime - elapseTime;  
        if (leftTime != 0) {  
            return;  
        } else {  
            controlPedestrianSignal();  
        }  
    }  
  
    private void controlPedestrianSignal() {  
        if(leftPedestrianSignal.isGreenLightOn()) {  
            leftPedestrianSignal.turnRedLightOn();  
            rightPedestrianSignal.turnRedLightOn();  
        } else {  
            leftPedestrianSignal.turnGreenLightOn();  
            rightPedestrianSignal.turnGreenLightOn();  
        }  
    }  
}
```

# 횡단보도 3 – 시라니오: 확장 (누가 신호등을 끄고 켜나?)

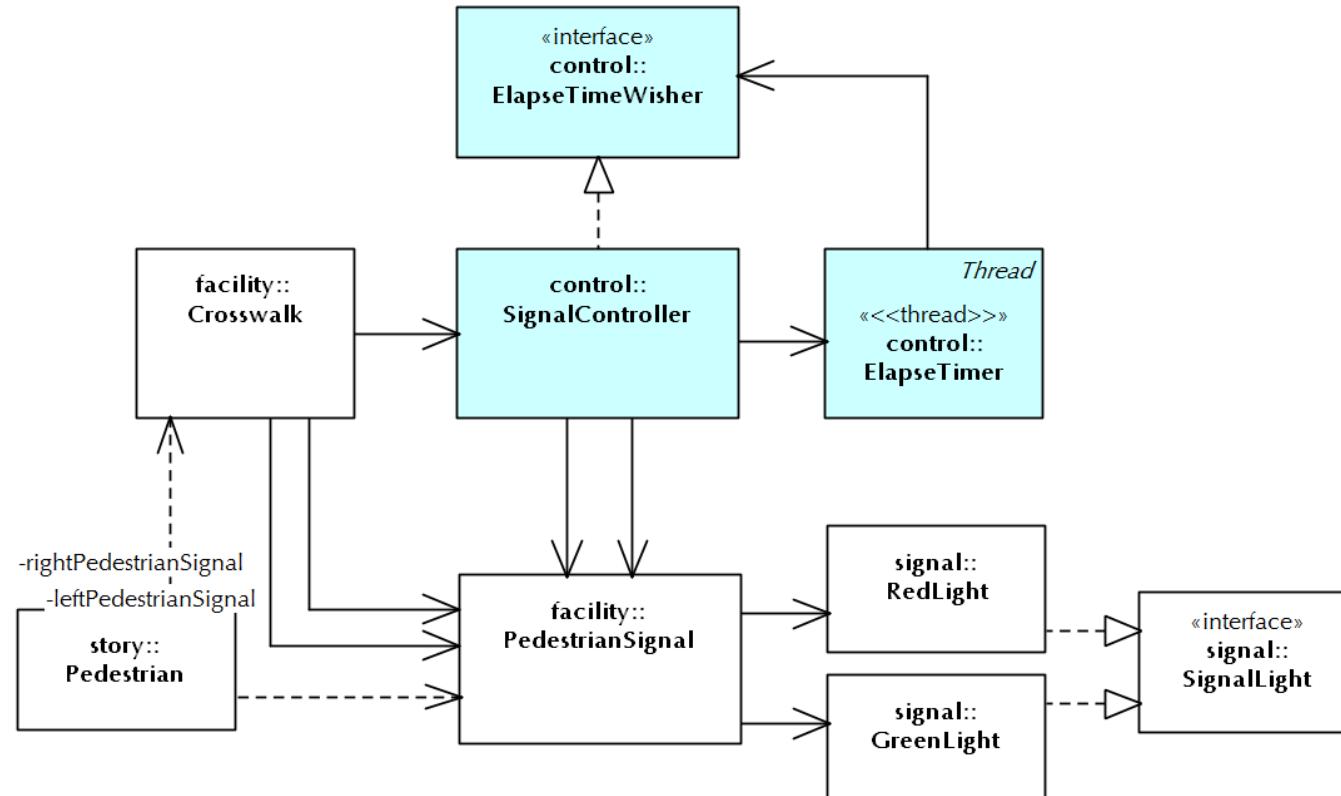
- ✓ 보행자신호등에게 신호등 제어를 위임한 후에 사후 확인하고 보정하는 시나리오를 살펴봅시다.



```
public class SignalController {  
    public void notifyElapseTime(int elapseTime) {  
        leftTime = baseTime - elapseTime;  
  
        leftPedestrianSignal.notifyLeftTime(leftTime);  
        rightPedestrianSignal.notifyLeftTime(leftTime);  
  
        if (leftTime == 0) {  
            syncSignalLight();  
        }  
    }  
  
    private void syncSignalLight() {  
        if (leftPedestrianSignal.isGreenLightOn()) {  
            if (!rightPedestrianSignal.isGreenLightOn()) {  
                rightPedestrianSignal.turnGreenLightOn();  
            }  
        } else {  
            if (!rightPedestrianSignal.isRedLightOn()) {  
                rightPedestrianSignal.turnRedLightOn();  
            }  
        }  
    }  
}
```

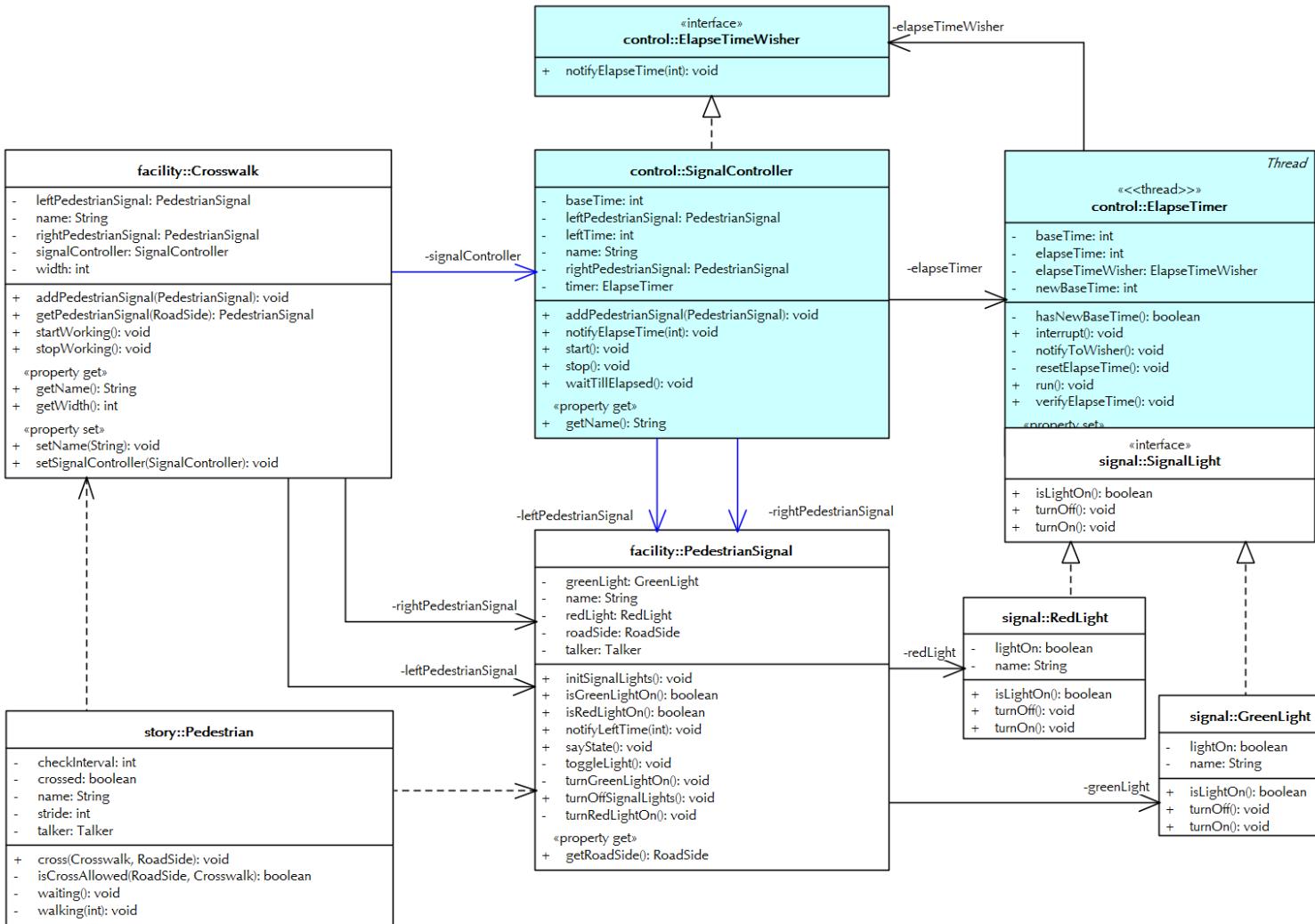
## 횡단보도 3 – 모델: 참여 객체들

- ✓ 소프트웨어를 키워가고 있는 모델러나 프로그래머의 작업은 의사결정의 연속입니다.
- ✓ 새로 초대한 클래스들은 기존 클래스들과 충돌없이 책임을 잘 나누어 협업하도록 관계가 설정되고 있습니다.
- ✓ 다음단계로 성장하기 위하여 참여하는 클래스들을 살펴봅시다.



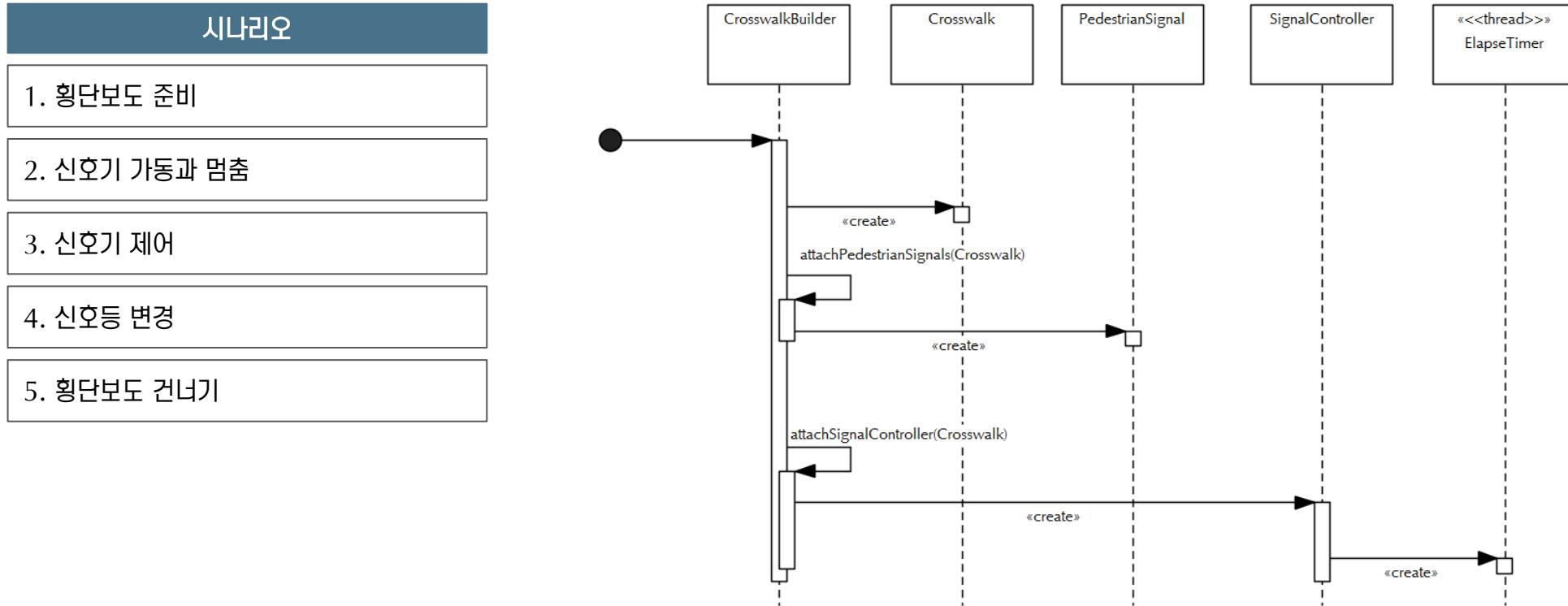
# 횡단보도 3 – 모델: 참여 객체들(상세)

✓ 클래스 다이어그램을 좀 더 자세히 들여다봅니다.



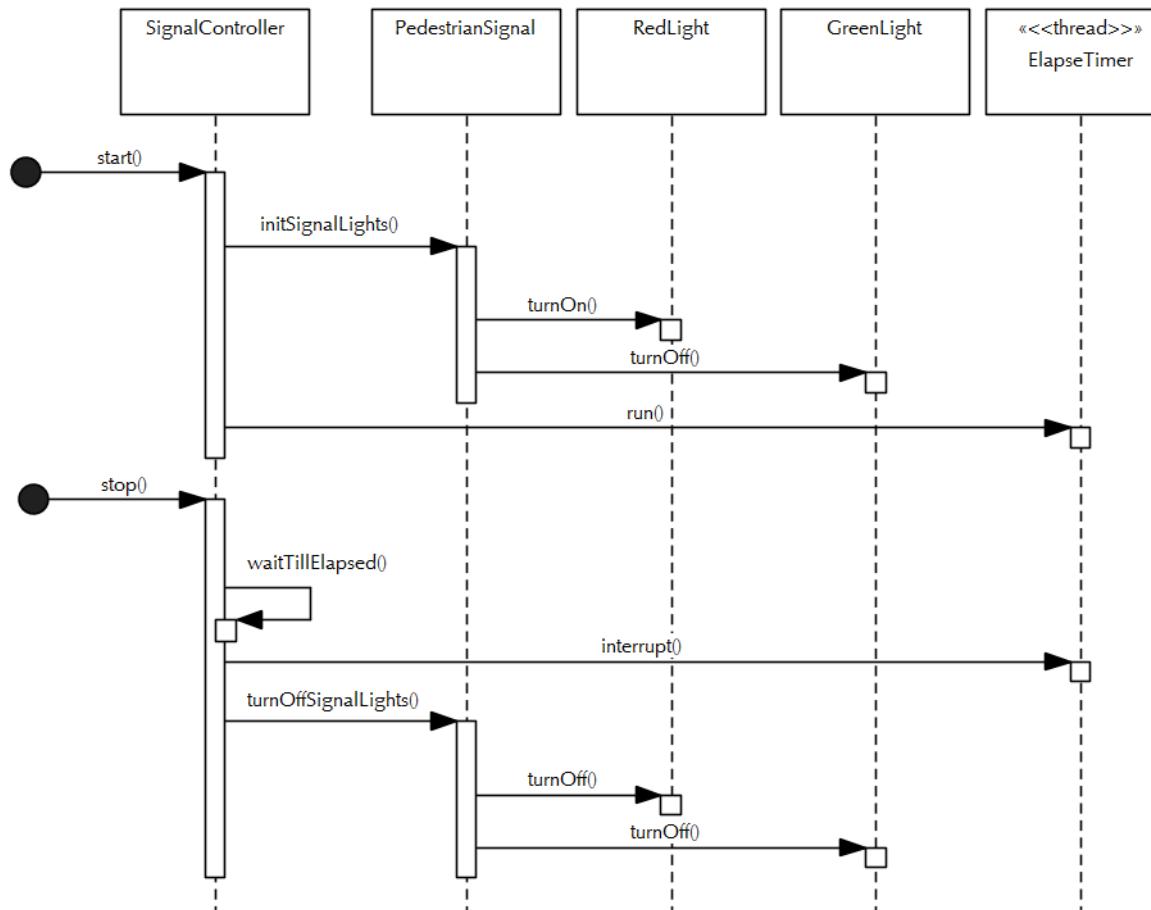
# 횡단보도3 – 모델: 협업1 (횡단보도 준비)

- ✓ 소프트웨어가 성장함에 따라 새로운 장치들이 횡단보도에 설치될 것입니다.
- ✓ 장치들이 추가되면서 장치 간의 관계가 복잡해질 것이 예상됩니다.
- ✓ 설치 절차를 명확하게 정의하고 간결한 규칙을 바탕으로 관계를 구성하지 않으면 금새 복잡해질 수 있습니다.
- ✓ 아래의 시퀀스 다이어그램처럼 장치별로 설치 메소드를 별도로 두어서 성장에 대비하여야 합니다.



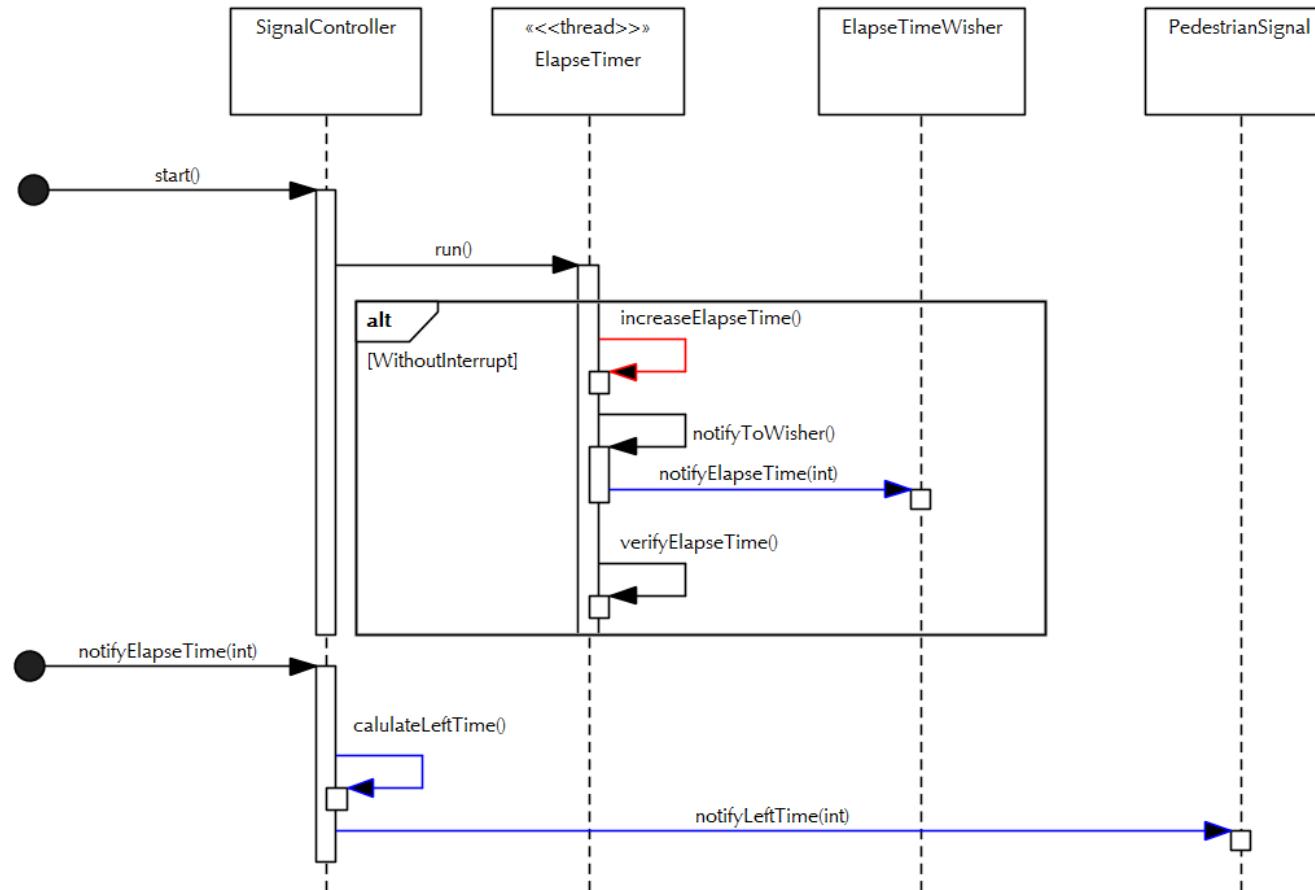
## 횡단보도3 – 모델: 협업2 (신호제어 시작과 종료)

✓ 신호 제어를 시작하는 것과 종료하는 시나리오 입니다. 애플리케이션은 두 개의 스레드로 수행됩니다.



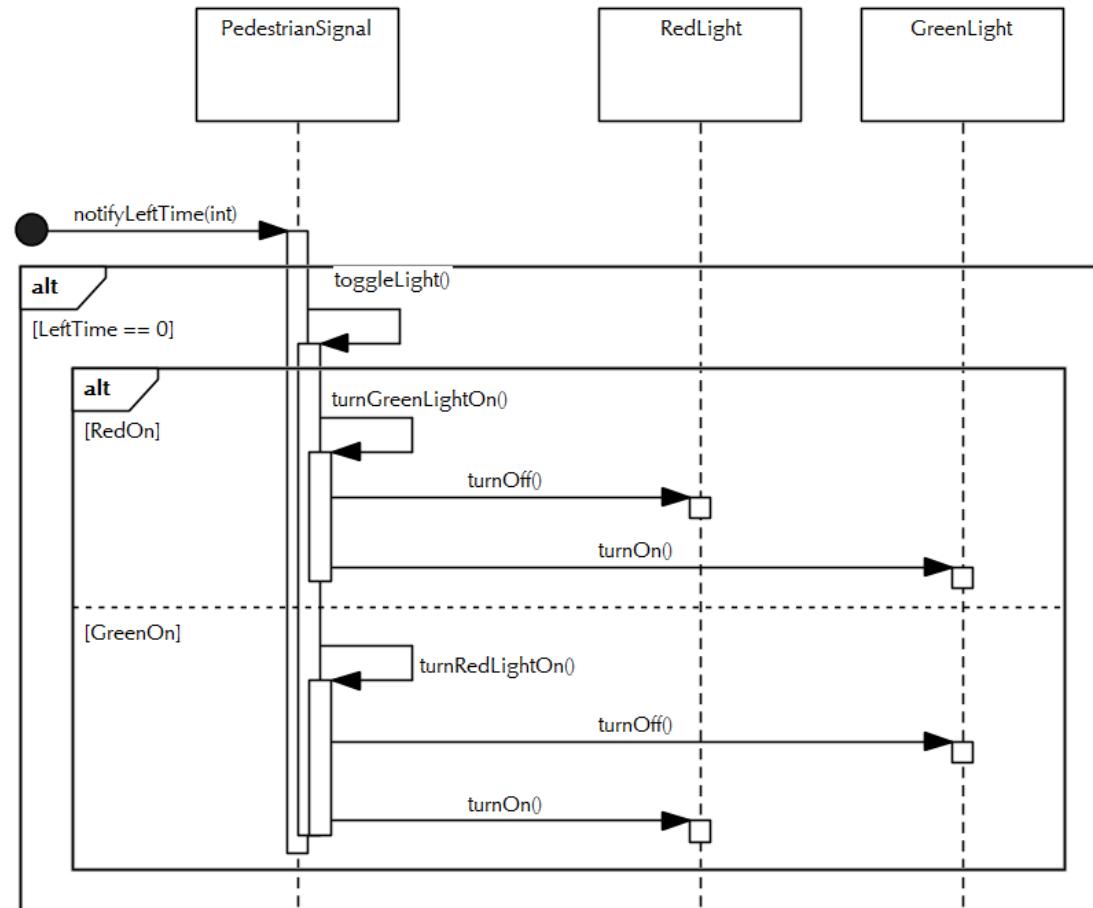
# 횡단보도3 – 모델: 협업3 (신호시간 전달)

- ✓ 신호제어는 제어 시간 전달과 신호등 변경으로 구성됩니다.
- ✓ 신호 시간을 전달하기 위해 참여 클래스들이 협업하는 모습을 살펴봅니다.



## 횡단보도3 – 모델: 협업4 (신호등 변경)

- ✓ 신호등을 녹색등에서 적색등으로 그리고 그 반대로 변경합니다.
- ✓ 보행자신호등은 현재 신호등이 바뀌기 까지 남은 시간을 받습니다.



# 횡단보도3 – 모델: ElapseTimer

- ✓ ElapseTimer는 경과시간을 계산하는 클래스로 횡단보도의 심장과 같은 역할을 합니다.
- ✓ ElapseTimer 클래스는 독립 스레드로 실행되어 합니다.

```
public class ElapseTimer extends Thread {  
    //  
    private int baseTime;  
    private int elapseTime;  
  
    private ElapseTimeWisher elapseTimeWisher;  
  
    public ElapseTimer(ElapseTimeWisher elapseTimeWisher, int baseTime){  
        //  
        super("goodTimer");  
        this.baseTime = baseTime;  
        this.elapseTimeWisher = elapseTimeWisher;  
        ...  
    }  
  
    public void run(){  
        elapseTime = 0;  
        talker.sayAtRight("ElapseTimer를 시작합니다. (기준시간 --> " + baseTime + "초)");  
  
        while(true){  
            if (!increaseSuccessfully()) {  
                break;  
            }  
            notifyElapseTime();  
            resetElapseTime();  
        }  
    }  
}
```

```
private boolean increaseSuccessfully() {  
    //  
    try {  
        Thread.sleep(1000);  
        elapseTime++;  
    } catch (InterruptedException e) {  
        talker.sayAtRight("interrupt 메시지를 받았습니다. ");  
        return false;  
    }  
  
    return true;  
}  
  
private void notifyElapseTime() {  
    //  
    elapseTimeWisher.notifyElapseTime(elapseTime);  
}  
  
private void resetElapseTime() {  
    //  
    if (elapseTime == baseTime) {  
        elapseTime = 0;  
    }  
}
```

## 횡단보도3 – 모델: SignalController

- ✓ SignalController는 ElapseTimer로부터 경과시간을 받는 유일한 객체입니다.
- ✓ ElapseTimer는 한 객체 만을 상대하므로 관계를 통해 상대를 알아야 하는 부담으로부터 자유롭습니다.
- ✓ SignalController는 가장 중요한 “경과 시간”을 혼자 갖게되었으므로 다른 객체에 대한 통제권을 갖습니다.
- ✓ 두 객체 모두 R&R에 만족한 상태가 되었습니다.

```
public class SignalController implements ElapseTimeWisher {
    //
    private int baseTime;
    private int leftTime;
    private String name;
    private ElapseTimer elapseTimer;

    private PedestrianSignal leftPedestrianSignal;
    private PedestrianSignal rightPedestrianSignal;

    public SignalController(int baseTime){
        //
        this.name = "controller";
        this.baseTime = baseTime;
        this.elapseTimer = new ElapseTimer(this, baseTi
        ...
    }
    ...
}

@Override
public void notifyElapseTime(int elapseTime) {
    //
    leftTime = baseTime - elapseTime;

    leftPedestrianSignal.notifyLeftTime(leftTime);
    rightPedestrianSignal.notifyLeftTime(leftTime);

    if (leftTime == 0) {
        syncSignalLight();
    }
}
```

## 횡단보도3 – 모델: SignalController

- ✓ 신호등 변경은 보행자 신호등에게 위임하여, 좌우 신호등이 일치하는지 여부를 확인하여야 합니다.
- ✓ 좌우 신호등이 같은 색상의 등을 켰는지 아래 syncSignalLight()에서 확인합니다.

```
private void syncSignalLight() {
    if (leftPedestrianSignal.isGreenLightOn()) {
        if (!rightPedestrianSignal.isGreenLightOn()) {
            rightPedestrianSignal.turnGreenOnRedOff();
        }
    } else {
        if (rightPedestrianSignal.isGreenLightOn()) {
            rightPedestrianSignal.turnRedOnGreenOff();
        }
    }
}

public void addPedestrianSignal(PedestrianSignal pedestrianSignal){
    RoadSide roadSide = pedestrianSignal.getRoadSide();
    switch (roadSide) {
        case Left:
            leftPedestrianSignal = pedestrianSignal;
            break;
        case Right:
            rightPedestrianSignal = pedestrianSignal;
    }
}
```

## 횡단보도3 – 모델: SignalController

- ✓ start() 메시지를 받으면 제어를 시작하고 stop() 메시지를 받으면 신호등 제어를 마칩니다.
- ✓ 좌우측 신호등을 초기화하고 타이머를 시작하는 방식으로 제어를 시작합니다.
- ✓ 신호 제어를 마치려면 elapseTimer 스레드로 interrupt()메시지를 보냅니다.

```
public void start(){  
    if (leftPedestrianSignal == null || rightPedestrianSignal == null) {  
        throw new RuntimeException("No control target(pedestrian signal).");  
    }  
    leftPedestrianSignal.init();  
    rightPedestrianSignal.init();  
    elapseTimer.start();  
}  
public void stop() {  
    waitTillElapsed();  
    elapseTimer.interrupt();  
}  
private void waitTillElapsed() {  
    talker.sayAtRight("신호등이 바뀔 때까지 기다립니다.");  
    while(leftTime != 1) {  
        sleepForSeconds(1);  
    }  
}
```

## 횡단보도3 – 모델: PedestrianSignal

- ✓ SignalController는 PedestrianSignal로 남은 시간을 notifyLeftTime() 메시지를 통해 알려줍니다.
- ✓ 보행자신호등은 남은 시간이 0이 되면 현재 켜져 있는 신호등을 끄고 꺼져있는 신호등을 켭니다.
- ✓ 적색 신호등을 켜고, 녹색 신호등을 켜는 활동은 동시에 한 메소드 안에서 수행해야 합니다. 필요하면 동기화 수행을 하도록 지정합니다. → synchronized

```
public class PedestrianSignal {  
    public void init() {  
        redLight.turnOn();  
        greenLight.turnOff();  
    }  
    public void notifyLeftTime(int leftTime) {  
        if (leftTime == 0) {  
            toggleLight();  
        }  
    }  
    private void toggleLight() {  
        if (isGreenLightOn()) {  
            turnRedOnGreenOff();  
        } else {  
            turnGreenOnRedOff();  
        }  
    }  
    public void turnRedOnGreenOff() {  
        greenLight.turnOff();  
        redLight.turnOn();  
    }  
}
```

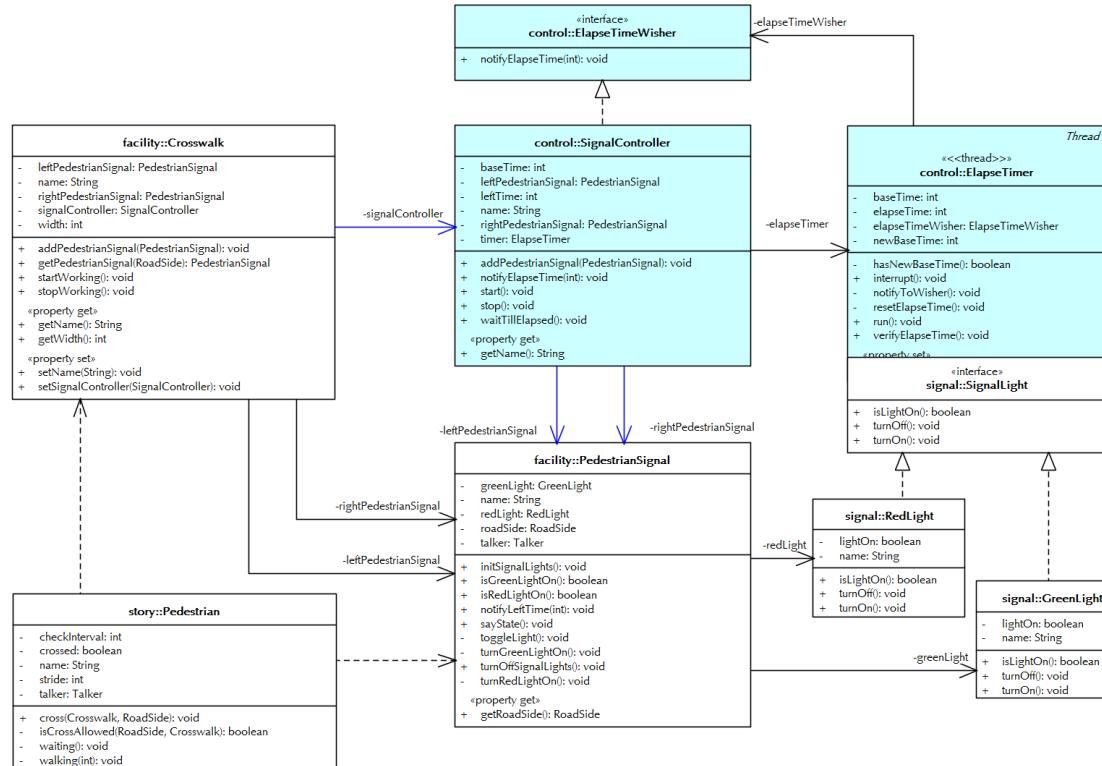
## 횡단보도3 – 모델: CrosswalkBuilder

- ✓ 신호제어기를 추가합니다. build() 메소드에서 장치별로 메소드를 분리합니다.
- ✓ CrosswalkBuilder는 이 도메인의 객체들 간의 관계를 잘 알고 있는 관계 전문가입니다.
- ✓ 대부분의 관계는 선,후가 있으며, Builder는 그 관계를 세심하게 고려하여 순서대로 횡단보도를 구축합니다.

```
public class CrosswalkBuilder {  
  
    public Crosswalk build(String name, int width) {  
        Crosswalk crosswalk = new Crosswalk(name, width);  
        attachPedestrianSignals(crosswalk);  
        attachSignalController(crosswalk);  
        return crosswalk;  
    }  
  
    private void attachPedestrianSignals(Crosswalk crosswalk) {  
        crosswalk.addPedestrianSignal(new PedestrianSignal(RoadSide.Left));  
        crosswalk.addPedestrianSignal(new PedestrianSignal(RoadSide.Right));  
    }  
  
    private void attachSignalController(Crosswalk crosswalk) {  
        int baseTime = 30;  
        crosswalk.setSignalController(new SignalController(baseTime));  
    }  
}
```

# 횡단보도3 – 요약

- ✓ 이제 신호등 제어 기능이 정상적으로 작동합니다. 시간의 흐름에 따라, 신호등이 변경됩니다.
- ✓ 이번 시나리오에서는 기존 개념 확장, 새로운 개념 등장, 역할과 책임 조정 등의 여러 가지를 포함했습니다.
- ✓ 세부 프로그래밍 실습은 강사의 가이드를 따라서 진행합니다.
- ✓ 실습 코드는 일부 클래스나 메소드를 제거한 상태에서 스켈레톤 코드를 제공합니다.



✓ 토론

## 감사합니다...

- ❖ 송태국 ([tsong@nextree.co.kr](mailto:tsong@nextree.co.kr))
- ❖ 넥스트리컨설팅(주) 대표이사
- ❖ 넥스트리소프트(주) 부사장
- ❖ [www.nextree.co.kr](http://www.nextree.co.kr)