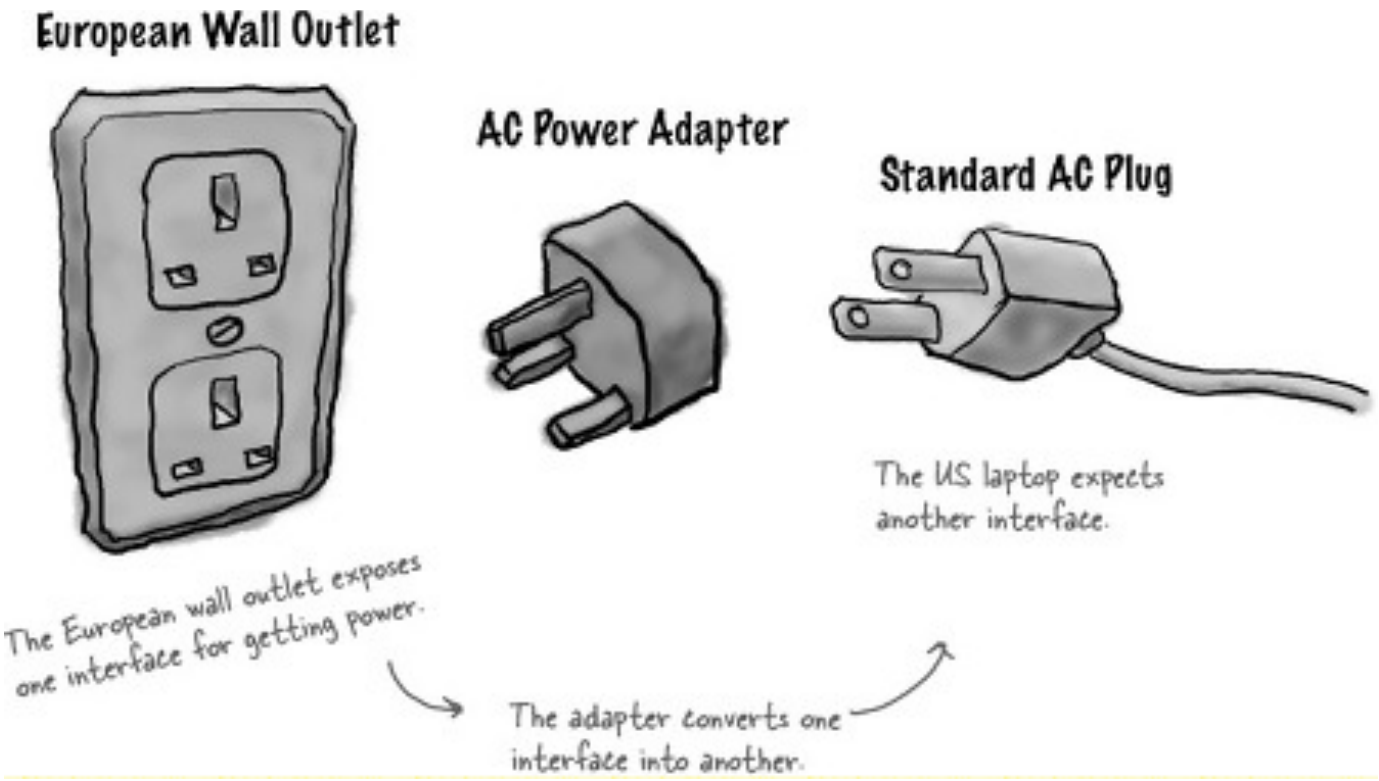
어댑터 패턴/퍼사드 패턴

이관우
kwlee@hansung.ac.kr

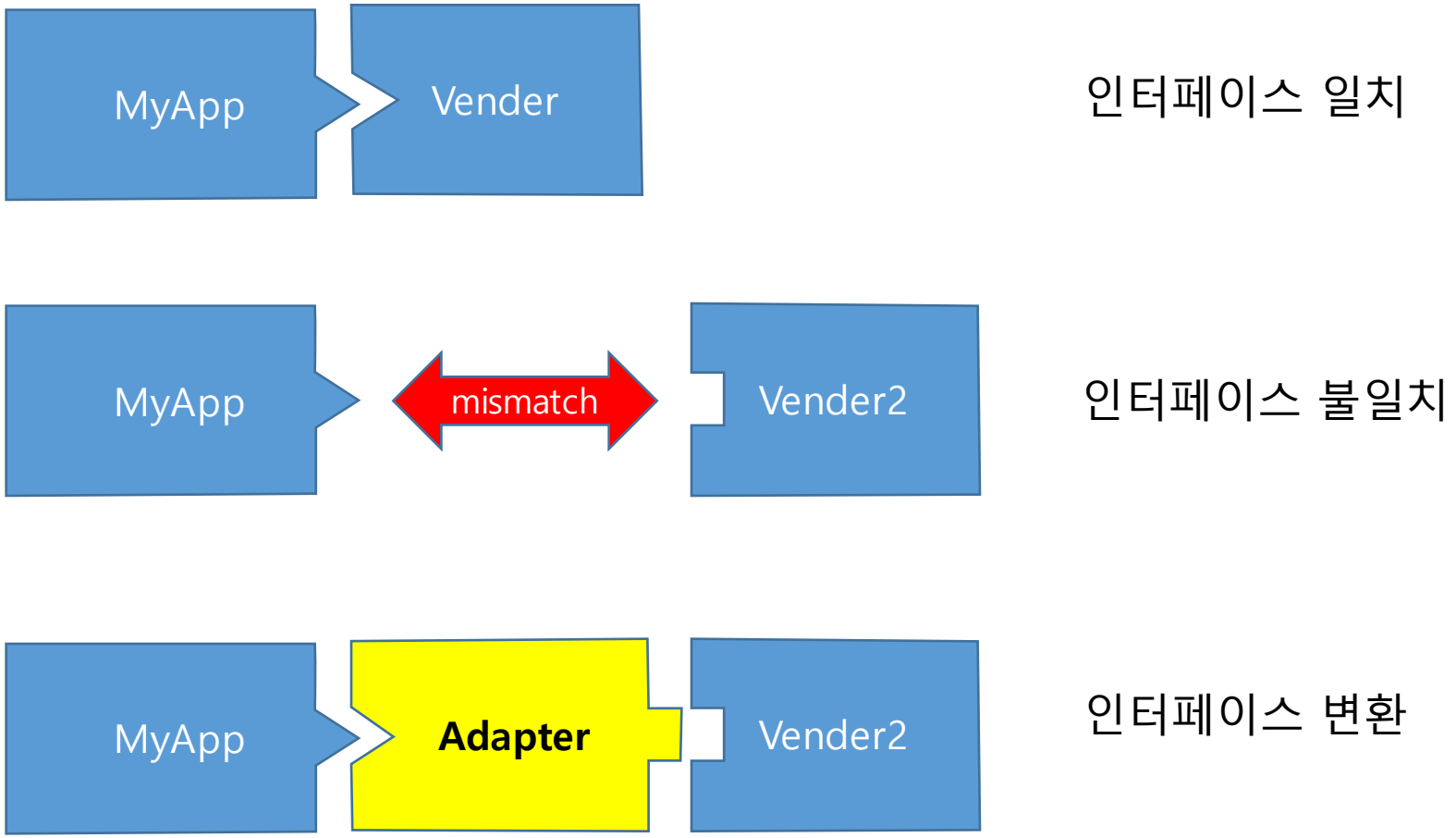
학습 목표

- 인터페이스 변환이 필요한 상황을 이해한다.
- 객체 어댑터 패턴과 클래스 어댑터 패턴의 차이를 이해한다.
- 퍼사드 패턴이 필요한 상황을 이해한다.
- 퍼사드 패턴과 최소 지식 원칙의 관계를 이해한다.

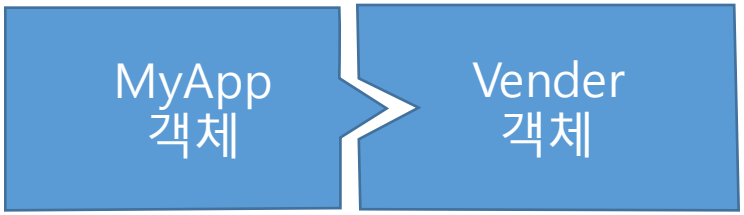
일상 생활에서의 어댑터


European Wall Outlet
AC Power Adapter
Standard AC Plug
The US laptop expects another interface.
The European wall outlet exposes one interface for getting power.
The adapter converts one interface into another.

인터페이스 변환 어댑터


MyApp
Vender
인터페이스 일치
MyApp
mismatch
Vender2
인터페이스 불일치
MyApp
Adapter
Vender2
인터페이스 변환

Example Code

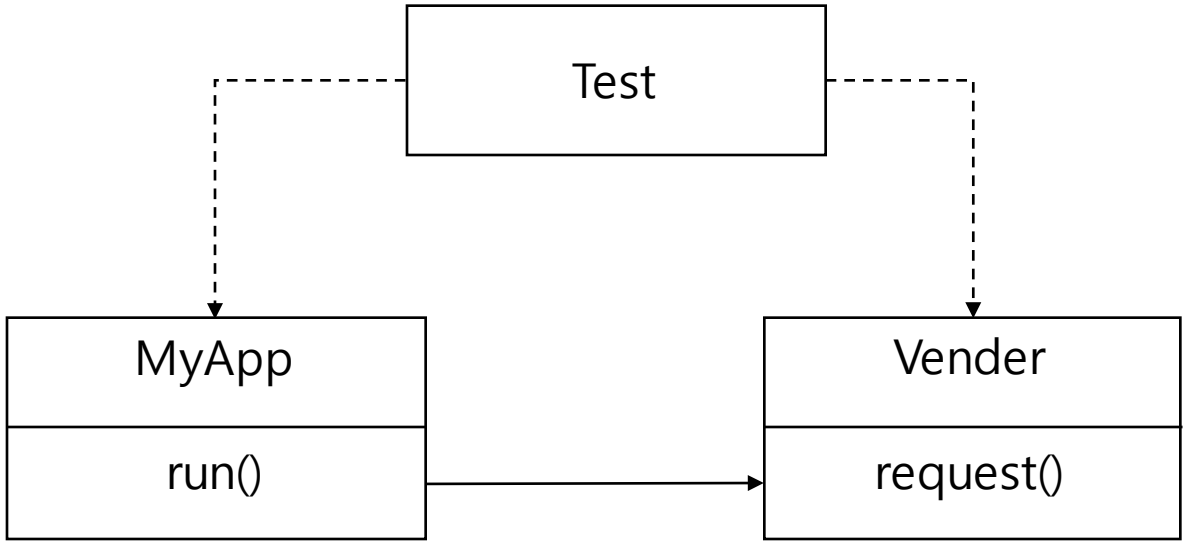



```
public class MyApp{
        Vender vender;                          // Vender 객체 인스턴스

        public MyApp(Vender vender) {
                this.vender = vender;           // vender 멤버변수 초기화
        }
        public void run() {
                vender.request();               // vender의 request() 호출
        }

}
```

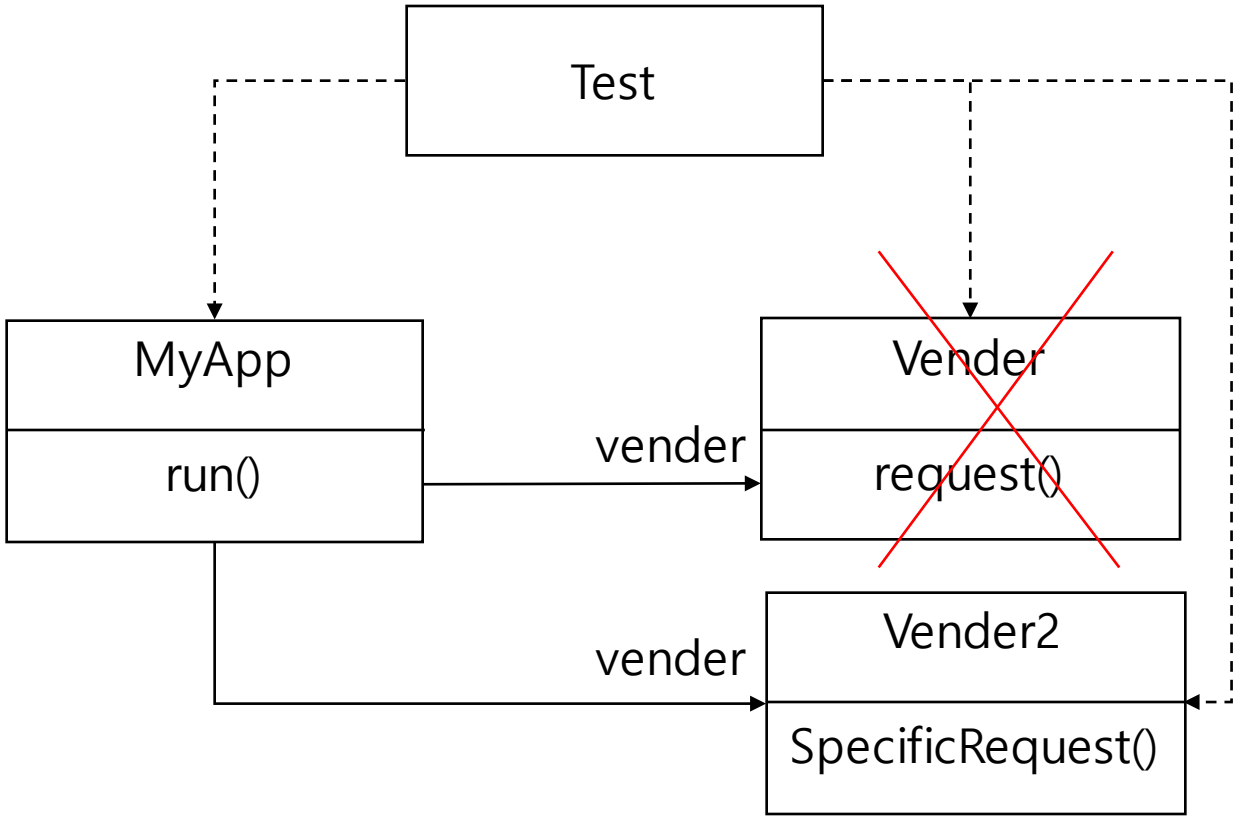
```
public class Vender {
        public void request() {
                System.out.println("A request is served");
        }
}
```

Example Code Test

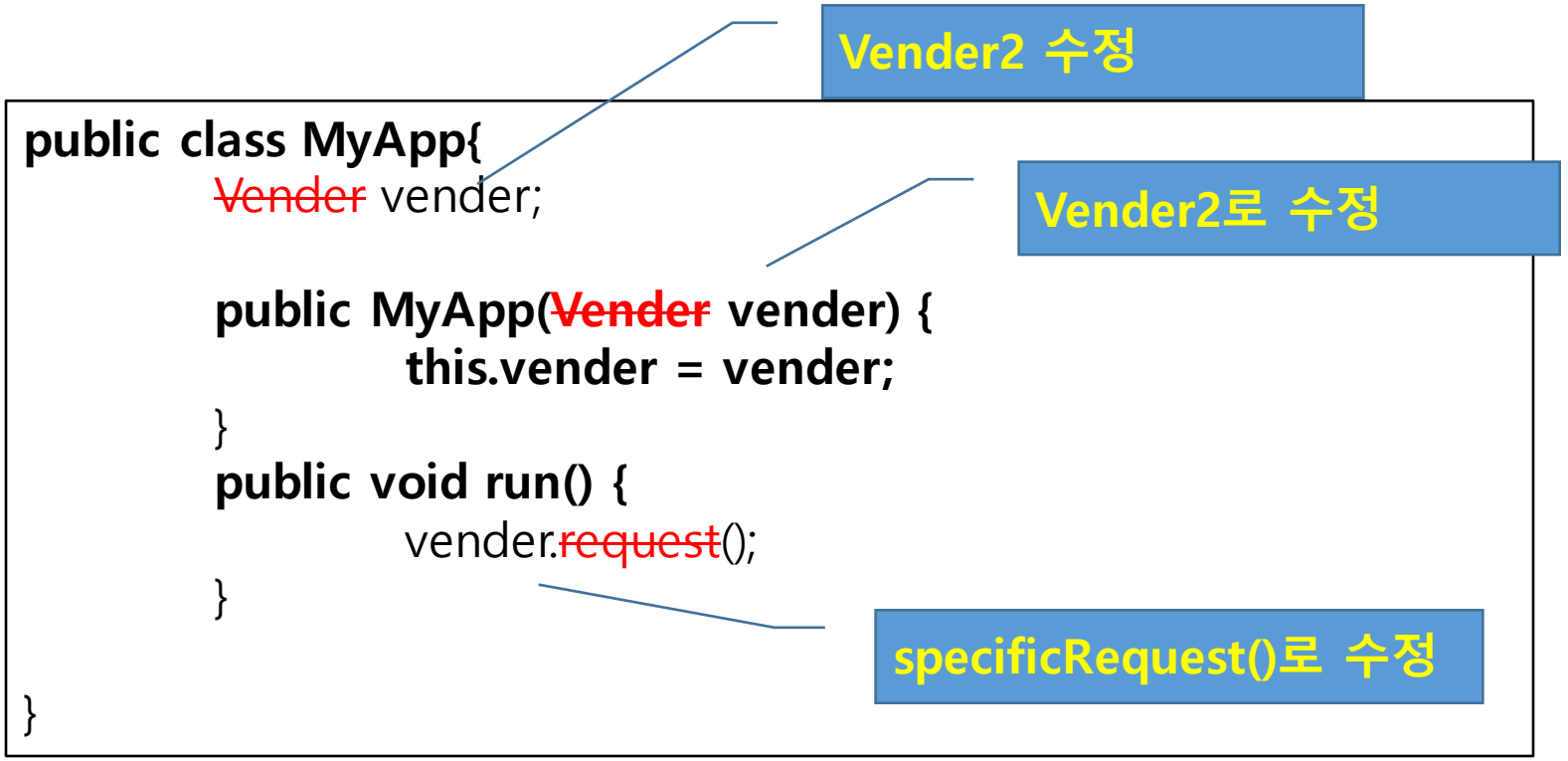
```
public class Test {
    public static void main(String[] args) {
        Vender vender = new Vender();
        MyApp app = new MyApp(vender);
        app.run();
    }
}
```



Vender에서 Vender2로 교체



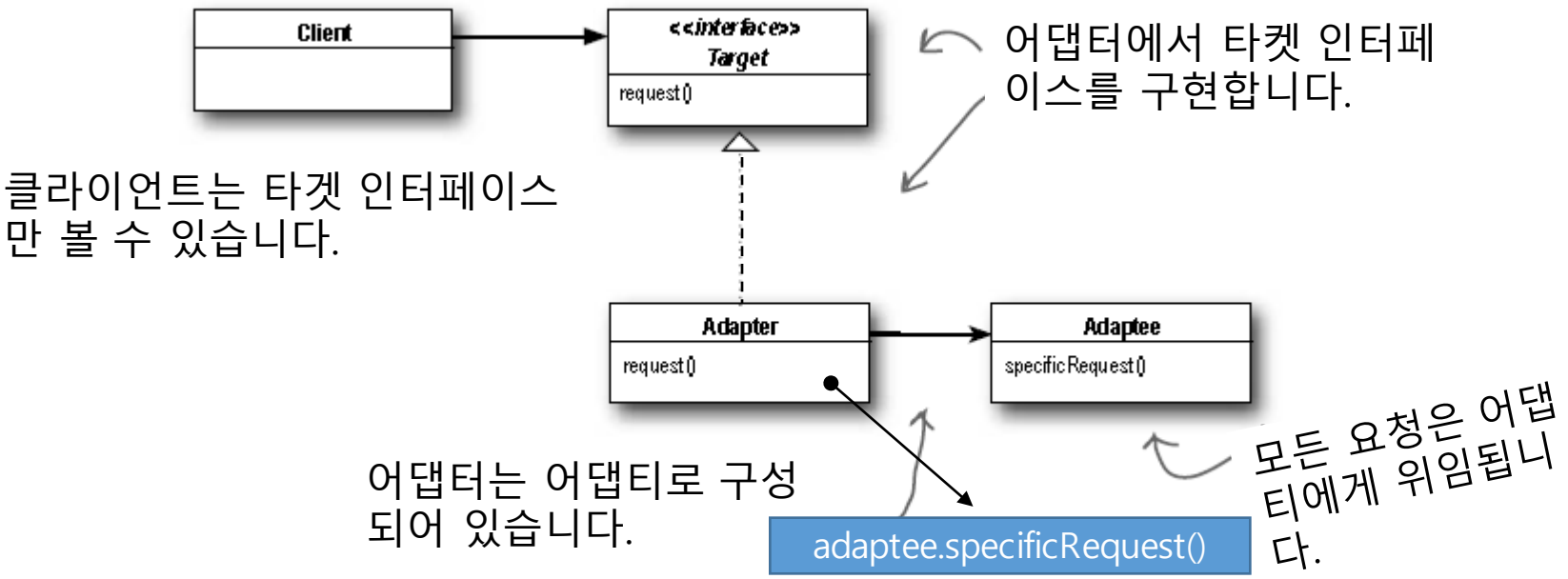
MyApp의 변경 영향



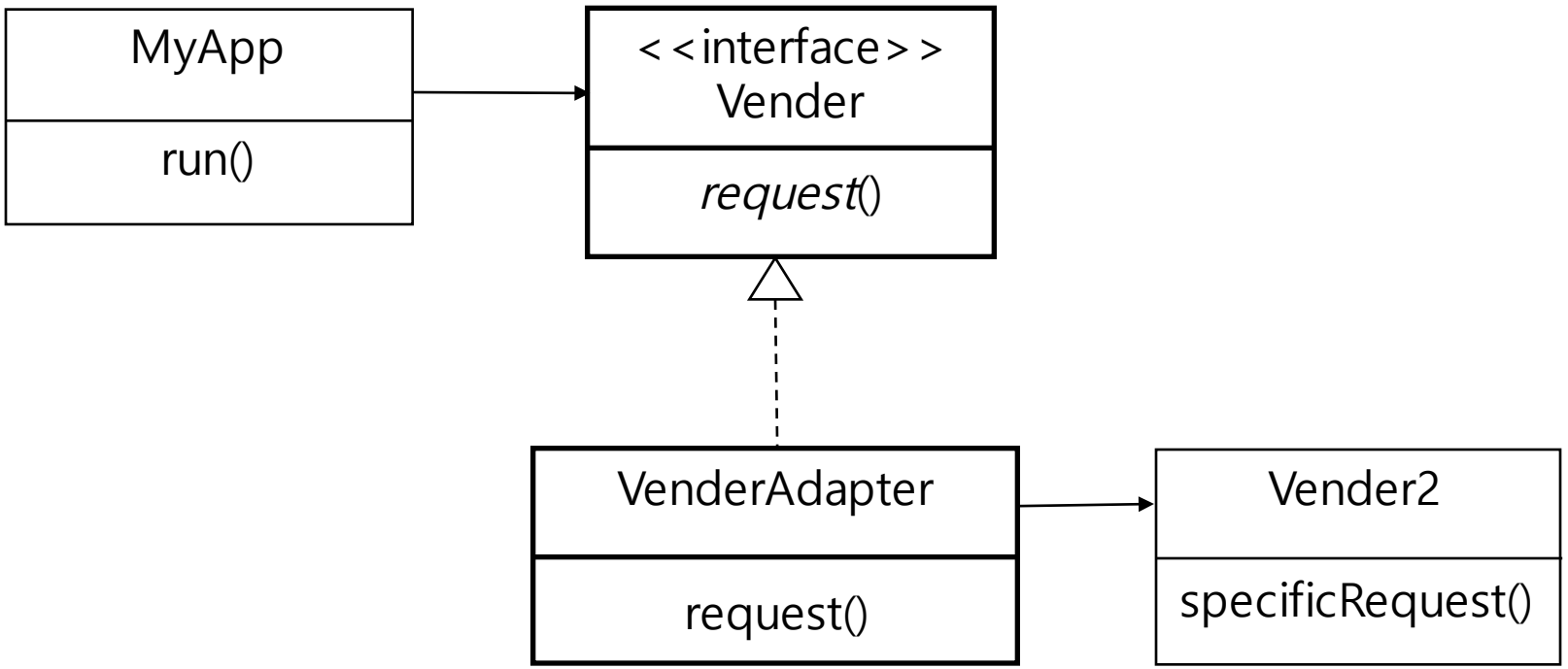

```
public class MyApp{
        Vender vender;

        public MyApp(Vender vender) {
                this.vender = vender;
        }
        public void run() {
                vender.request();
        }

}
```

어댑터 패턴

- 정의
  - 한 클래스의 인터페이스를 클라이언트에서 사용하고자 하는 다른 인터페이스로 변환합니다. 어댑터를 이용하면 인터페이스 호환성 때문에 같이 쓸 수 없는 클래스들을 연결해서 쓸 수 있습니다.



Example Code (애댑터 패턴 적용)



Example Code (애댑터 패턴 적용)

```
public interface Vender {
        void request();             // 타켓 인터페이스 정의
}
```

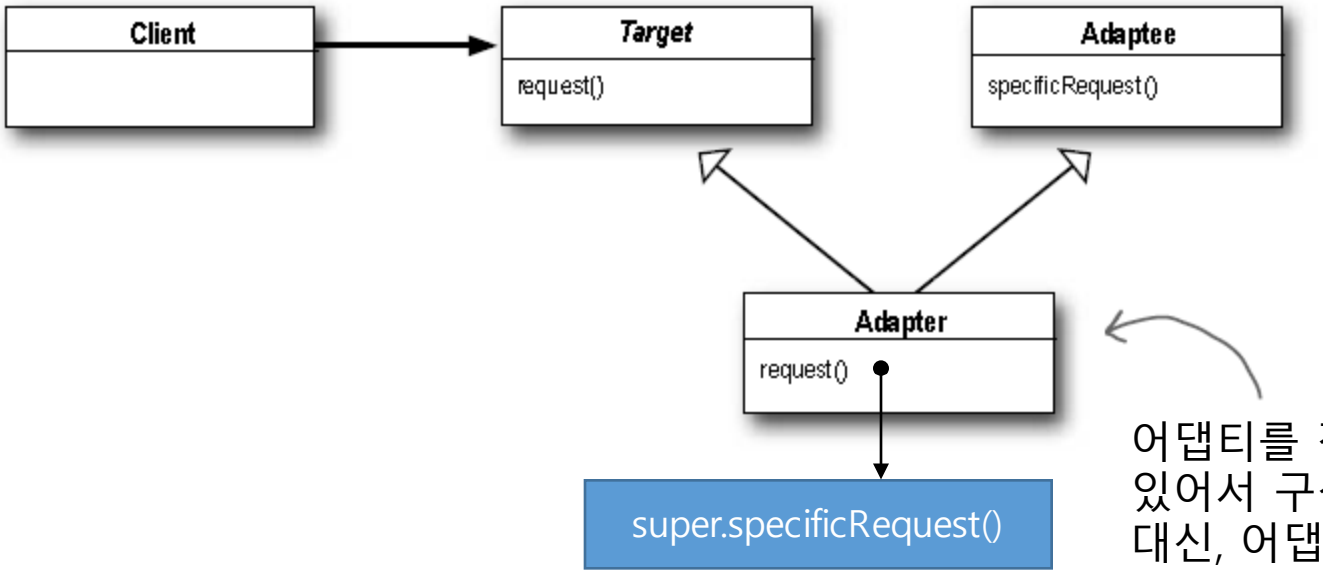
```
public class VenderAdapter implements Vender {
        Vender2 vender;     // 어댑티를 구성

        public VenderAdapter (Vender2 vender) {
                this.vender = vender;
        }

        public void request() {  // 타켓 인터페이스 구현
                vender.specificRequest(); // 어댑티로 요청 위임
        }
}
```

Simple Example (애댑터 패턴 적용)

```
public class Test {
    public static void main(String[] args) {
        Vender vender = new Vender();
        MyApp app = new MyApp(vender);
        app.run();
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Vender vender = new VenderAdapter(new Vender2());
        MyApp app = new MyApp(vender);
        app.run();
    }
}
```

클래스 어댑터



어댑티를 적응시키는 데 있어서 구성을 사용하는 대신, 어댑터를 어댑티와 타켓 클래스 모두의 서브 클래스로 만듭니다.

Example Code (클래스 애댑터 패턴)

```
public interface Vender {
        void request();              // 타켓 인터페이스 정의
}
```

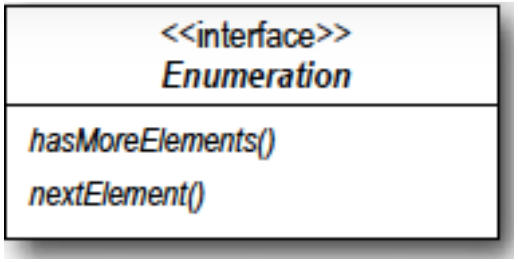
```
public class VenderClassAdapter extends Vender2
                                    implements Vender {

        public void request() {
                super.specificRequest();
        }

}
```

객체 어댑터 vs. 클래스 어댑터

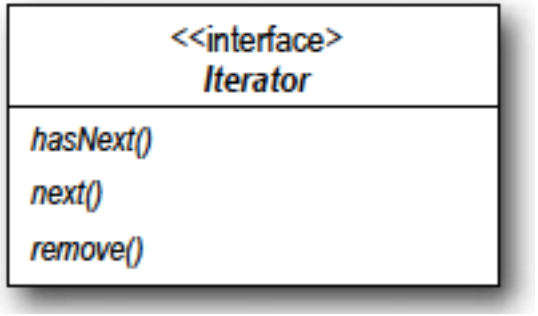
- 객체 어댑터
  - 객체 구성 사용
  - 객체 어댑터는 어댑티 클래스 뿐만 아니라 그 서브 클래스에 대해서도 어댑터 역할을 수행함
  - 서브 클래스의 레퍼런스만 가지고 있으면 어댑티의 행위가 오버라이드되더라도 이를 쉽게 이용할 수 있음
- 클래스 어댑터
  - 상속 사용
  - 클래스 어댑터는 특정 어댑티 클래스에만 어댑터 역할을 수행하므로, 어댑티의 서브클래스에 대해서는 어댑터 역할을 수행할 수 없음.
  - 클래스 어댑터는 필요한 경우에 어댑티의 행위의 일부를 직접 오버라이드 할 수 있음
  - 어댑티로의 추가적인 객체 레퍼런스 없이 어댑티로 접근 가능

어댑터 실전 예제

- Enumeration
  - 초기 컬렉션 (Vector, Stack, Hashtable 등)에서 사용
  - 컬렉션의 모든 항목에 접근하기 위한 인터페이스



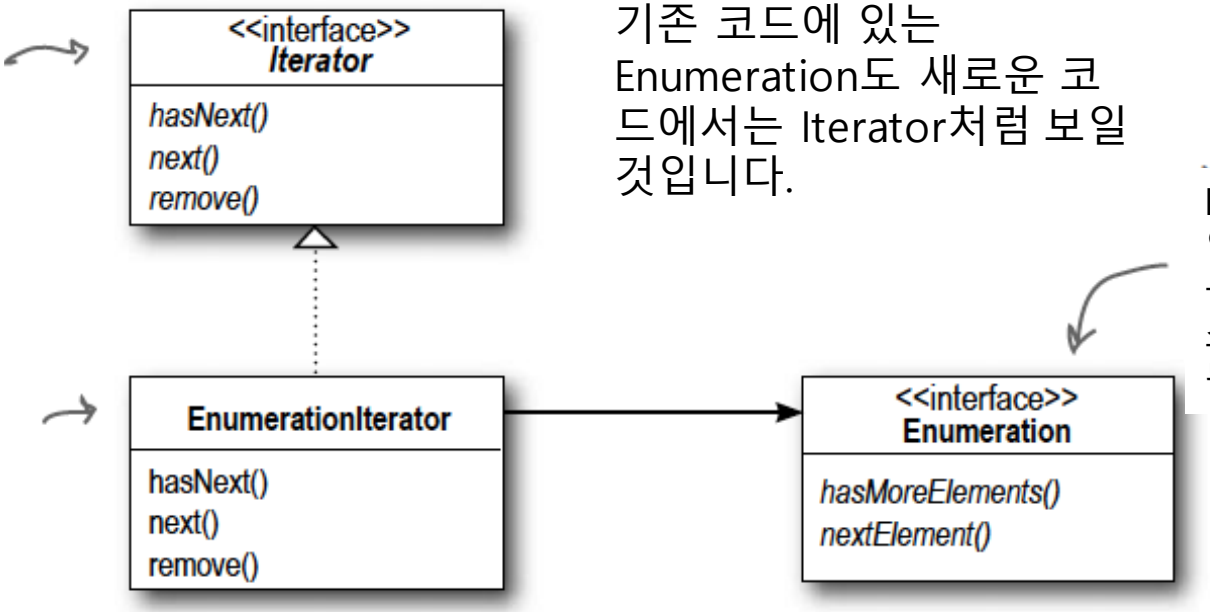
- Iterator
  - 새로운 버전의 컬렉션 클래스에서 항목에 접근할 때 사용하는 인터페이스



- 지금은...
  - 새로 만드는 코드에서는 Iterator만 사용할 계획
  - 하지만 Enumeration 인터페이스를 사용하는 기존 코드를 사용해야 하는 경우가 종종 있음.

어댑터 디자인

새로운 코드에서는 Iterator만 사용하게 됩니다. 물론 그 뒤에는 Enumeration이 숨어 있을 수도 있죠.



기존 코드에 있는 Enumeration도 새로운 코드에서는 Iterator처럼 보일 것입니다.

Enumeration 인터페이스를 구현하는 클래스는 어댑티가 됩니다.

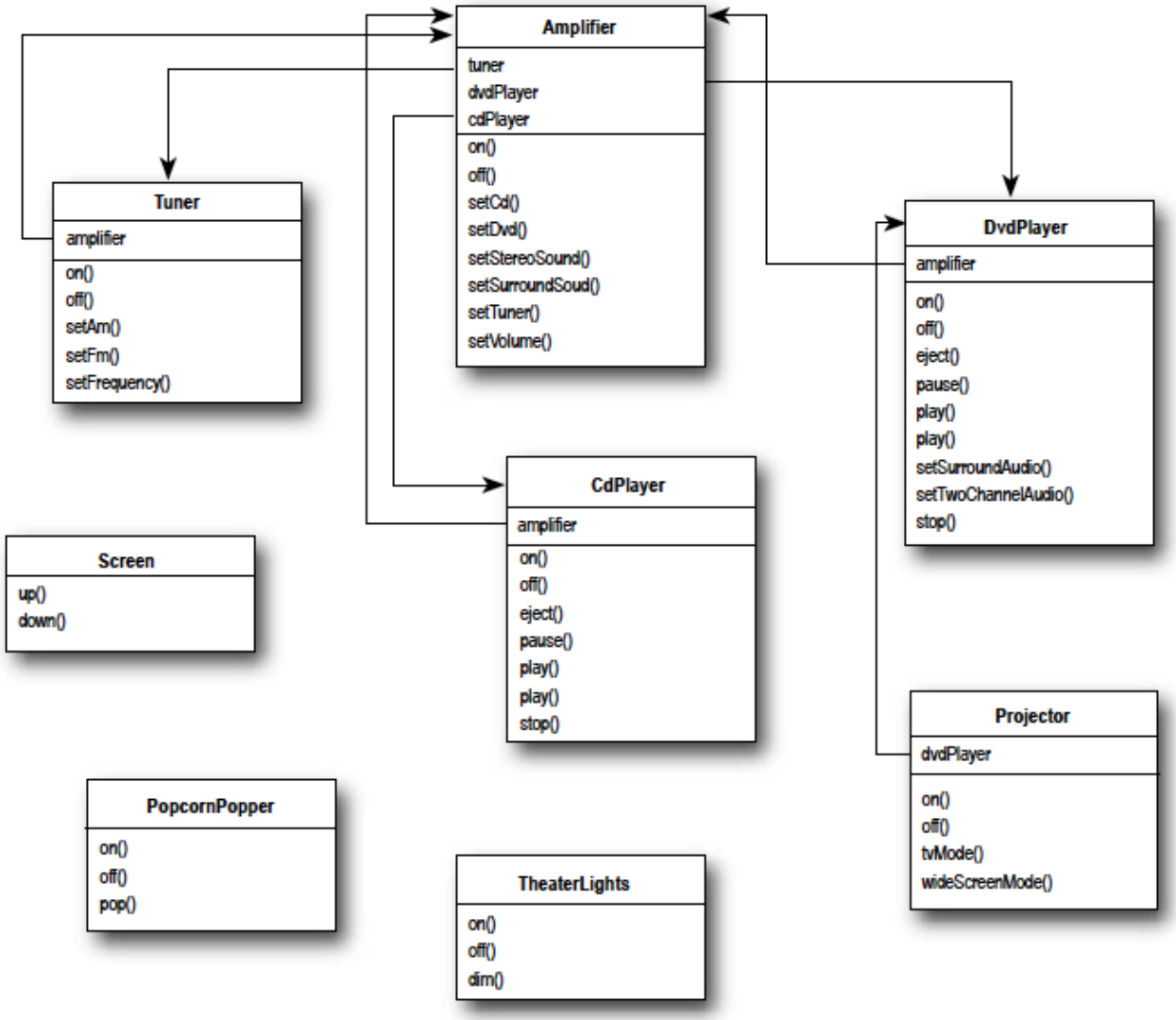
EnumerationIterator가 어댑터입니다.

EnumerationIterator 어댑터 코드

```
public class EnumerationIterator implements Iterator<Object> {
        Enumeration<?> enumeration;

        public EnumerationIterator(Enumeration<?> enumeration) {
                this.enumeration = enumeration;
        }

        public boolean hasNext() {
                return enumeration.hasMoreElements();
        }

        public Object next() {
                return enumeration.nextElement();
        }

        public void remove() {
                throw new UnsupportedOperationException();
        }
}
```

홈 씨어터

클래스가 많고, 클래스들은 서로 복잡하게 얽혀 있어, 제대로 사용하려면 꽤 많은 인터페이스를 배우고 쓸 수 있어야 합니다.



영화보기 위해 필요한 작업들...

1. 팝콘 기계를 견다
2. 팝콘 튀기기 시작
3. 전등을 어둡게 조절
4. 스크린을 내린다
5. 프로젝터를 켠다
6. 프로젝트를 와이드 스크린 모드로 전환한다.
7. 앰프를 켠다
8. 앱프 입력을 DVD로 전환한다.
9. 앰프를 서라운드 음향 모드로 전환한다.
10. 앰프 볼륨을 중간(5)로 설정한다.
11. DVD 플레이어를 켠다
12. DVD를 서라운드 오디오로 설정한다.
13. DVD를 재생한다.

아직 끝난 게 아닙니다...

- 영화가 끝나면 어떤 식으로 꺼야 할까요?
  방금 했던 일을 전부 역순으로 해야 되지 않을 까요?
- CD나 라디오를 들을 때도 이렇게 복잡할까요?
- 시스템을 업그레이드하면 또 다른 작동 방법을 배워야 하지 않을까요?

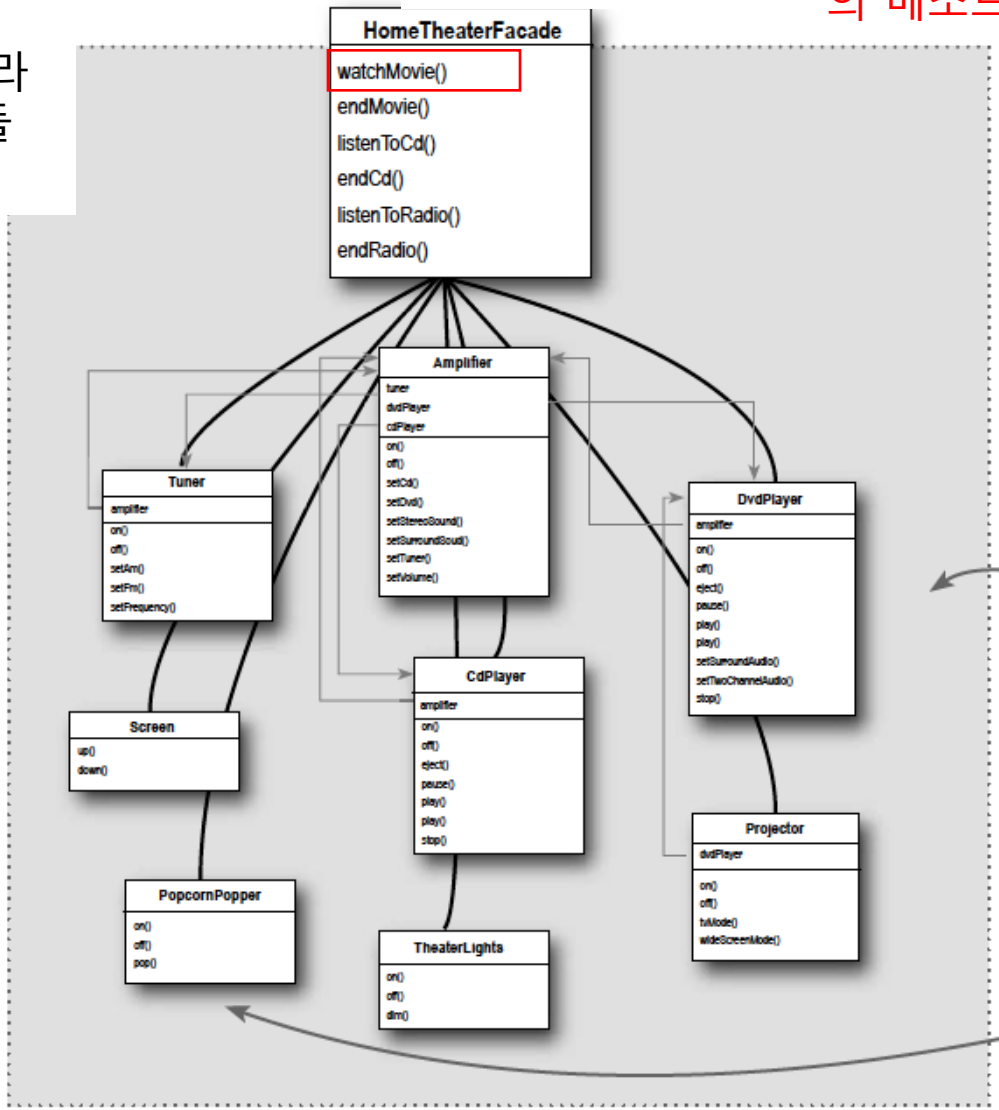
퍼사드 패턴을 쓰면 훨씬 쓰기 쉬운 인터페이스를 제공하는 퍼사드 클래스를 구현함으로써, 복잡한 시스템을 훨씬 쉽게 사용할 수 있습니다.

홈 씨어터 시스템용 퍼사드를 만들어 봅시다.

watchMovie() 같이 몇 가지 간단한 메소드만 들어있는 HomeTheaterFacade라는 클래스를 새로 만들어야 합니다.

퍼사드

퍼사드 클래스에서는 홈 씨어터 구성요소들을 하나의 서브시스템으로 간주하고 watchMovie() 메소드에서는 서브시스템의 메소드들을 직접 호출



play()

퍼사드를 쓰더라도 서브시스템에는 여전히 직접 접근 가능

퍼사드를 써서 단순화 시킨 서브 시스템

on()

HomeTheaterFacade

```
public class HomeTheaterFacade {
        Amplifier amp;
        Tuner tuner;
        DvdPlayer dvd;
        CdPlayer cd;
        Projector projector;
        TheaterLights lights;
        Screen screen;
        PopcornPopper popper;

        public HomeTheaterFacade(Amplifier amp,
                                 Tuner tuner,
                                 DvdPlayer dvd,
                                 CdPlayer cd,
                                 Projector projector,
                                 Screen screen,
                                 TheaterLights lights,
                                 PopcornPopper popper) {
        … }
```

https://github.com/kwanulee/DesignPattern/blob/master/facade/hometheater/src/hansung/designpatterns/facade/hometheater/HomeTheaterFacade.java

HomeTheaterFacade

```
public void watchMovie(String movie) {
        System.out.println("Get ready to watch a movie...");
        popper.on();
        popper.pop();
        lights.dim(10);
        screen.down();
        projector.on();
        projector.wideScreenMode();
        amp.on();
        amp.setDvd(dvd);
        amp.setSurroundSound();
        amp.setVolume(5);
        dvd.on();
        dvd.setSurroundAudio();
        dvd.play(movie);
}

// 기타 메소드
```

https://github.com/kwanulee/DesignPattern/blob/master/facade/hometheater/src/hansung/designpatterns/facade/hometheater/HomeTheaterFacade.java

HomeTheaterTestDrive

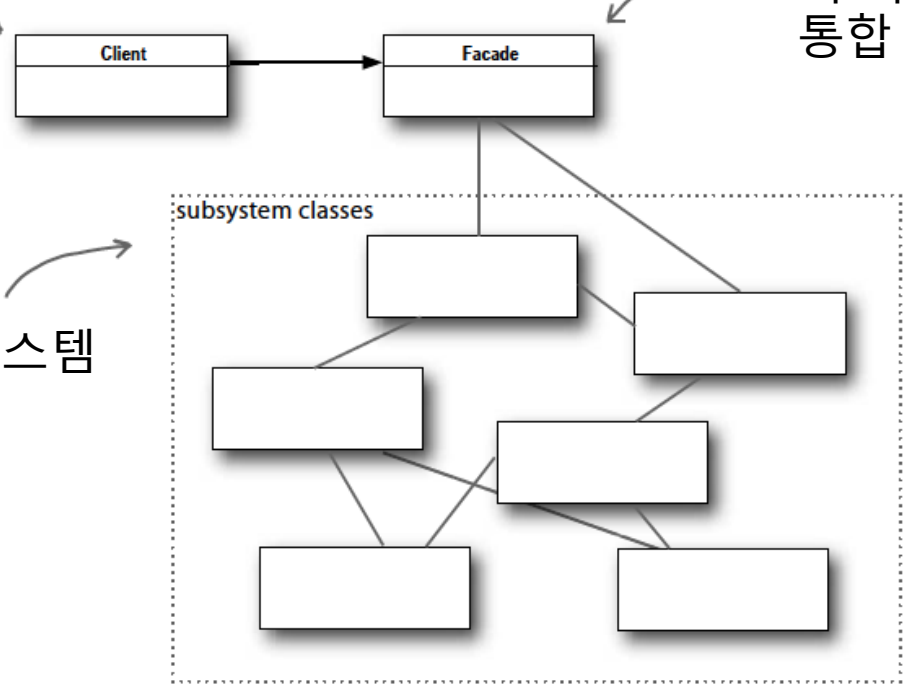
```
public class HomeTheaterTestDrive {
        public static void main(String[] args) {
                Amplifier amp = new Amplifier("Top-O-Line Amplifier");
                Tuner tuner = new Tuner("Top-O-Line AM/FM Tuner", amp);
                DvdPlayer dvd = new DvdPlayer("Top-O-Line DVD Player", amp);
                CdPlayer cd = new CdPlayer("Top-O-Line CD Player", amp);
                Projector projector = new Projector("Top-O-Line Projector", dvd);
                TheaterLights lights = new TheaterLights("Theater Ceiling Lights");
                Screen screen = new Screen("Theater Screen");
                PopcornPopper popper = new PopcornPopper("Popcorn Popper");

                HomeTheaterFacade homeTheater =
                                new HomeTheaterFacade(amp, tuner, dvd, cd,
                                                projector, screen, lights, popper);

                homeTheater.watchMovie("Raiders of the Lost Ark");
                homeTheater.endMovie();
        }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/facade/hometheater/src/hansung/designpatterns/facade/hometheater/HomeTheaterTestDrive.java

퍼사드 패턴

- 정의
  - 어떤 서브시스템의 일련의 인터페이스에 대한 통합된 인터페이스를 제공합니다.
  - 퍼사드에서 고수준 인터페이스를 정의하기 때문에 서브시스템을 더 쉽게 사용할 수 있습니다.

퍼사드 덕분에 일하기가 수월해져서 행복해하고 있는 클라이언트



더 사용하기 쉬운 통합 인터페이스

복잡한 서브시스템

최소 지식 원칙 (Principle of Least Knowledge) – 일명 Law of Demeter

정말 친한 친구하고만 얘기하라.
모르는 사람하고는 얘기하지 마라.

어떤 객체든 그 객체와 상호작용하는 클래스의 개수를 줄임으로써, 시스템의 한 부분이 변경되었을 때, 영향 받는 클래스의 수를 줄임

질문

- 이 코드는 몇 개의 클래스하고 연결되어 있을 까요?

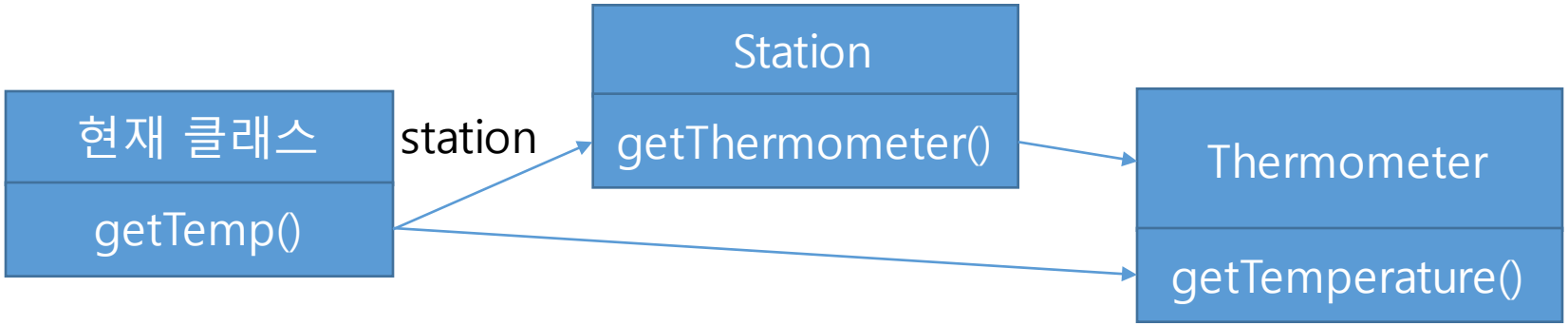
```
public float getTemp() {
        return station.getThermometer().getTemperature();
}
```

질문

- 이 코드는 몇 개의 클래스하고 연결되어 있을 까요?

```
public float getTemp() {
        return station.getThermometer().getTemperature();
}
```

```
public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return thermometer.getTemperature();
}
```



질문

- 어떻게 하면 최소 지식 원칙을 적용하여 의존하는 클래스의 수를 줄일 수 있을 까요?

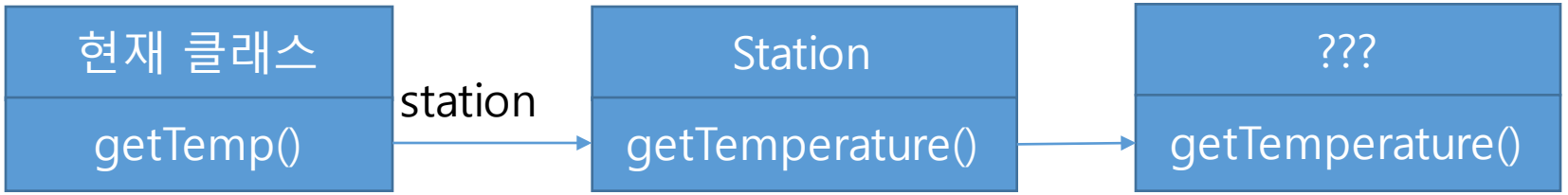
```
public float getTemp() {
        return station.getThermometer().getTemperature();
}
```

질문

- 어떻게 하면 최소 지식 원칙을 적용하여 의존하는 클래스의 수를 줄일 수 있을 까요?

```
public float getTemp() {
        return station.getThermometer().getTemperature();
}
```

```
public float getTemp() {
        return station.getTemperature();
}
```



최소 지식 원칙의 장단점

- 장점
  - 의존하는 클래스의 수를 줄임으로써, 설계의 복잡도를 줄일 수 있고, 변경에 대한 영향이 줄어들어 유지보수성 및 적응성이 높아짐
- 단점
  - 요청을 단순히 전달하는 래퍼 (Wrapper) 메소드가 많아짐으로 인해, 수행 속도의 오버헤드를 야기 시킴

핵심 정리

- 기존 클래스를 사용하려고 하는데 인터페이스가 맞지 않으면 어댑터를 사용
- 큰 인터페이스, 또는 여러 인터페이스를 단순화시키거나 통합시켜야 되는 경우에는 퍼사드를 사용
- 어댑터는 인터페이스를 클라이언트에서 원하는 인터페이스로 변환
- 퍼사드는 클라이언트를 복잡한 서브시스템과 분리시켜주는 역할