SOLID principle

한종대

설계는 잘 하고 있나요?

- 1차 설계 문서대로 개발해보면 굉장히 힘들 겁니다.
  - 좋은 설계를 만드는 데는 경험이 많이 필요하거든요
- ' 좋은 설계 ' 를 만드는 방법이 있나요?
  - 독립성 – 응집성 – 결합도
  - 어떻게 하면 응집성 – 결합도를 좋게 만들 수 있을까요
- 디자인 패턴, 엔터프라이즈 아키텍처, ...

SOLID Principle

- 좋은 '객체지향' 설계를 만들기 위한 기본 원칙들
- **S**ingle Responsibility Principle(단일 책임 원칙)
  - *A class should have only one reason to change.*
- **O**CP (개방폐쇄의 원칙: Open Close Principle)
  - *YOU SHOULD BE ABLE TO EXTEND A CLASSES BEHAVIOR, WITHOUT MODIFYING IT.*
- **L**SP (리스코프 치환 원칙: The Liskov Substitution Principle)
  - *FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.*
- **I**SP (인터페이스 분리의 원칙: Interface Segregation Principle)
  - *CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE.*
- **D**IP (의존성역전의 원칙: Dependency Inversion Principle)
  - *A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.*
  - *B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.*

https://code.tutsplus.com/series/the-solid-principles--cms-634

http://www.nextree.co.kr/p6960/

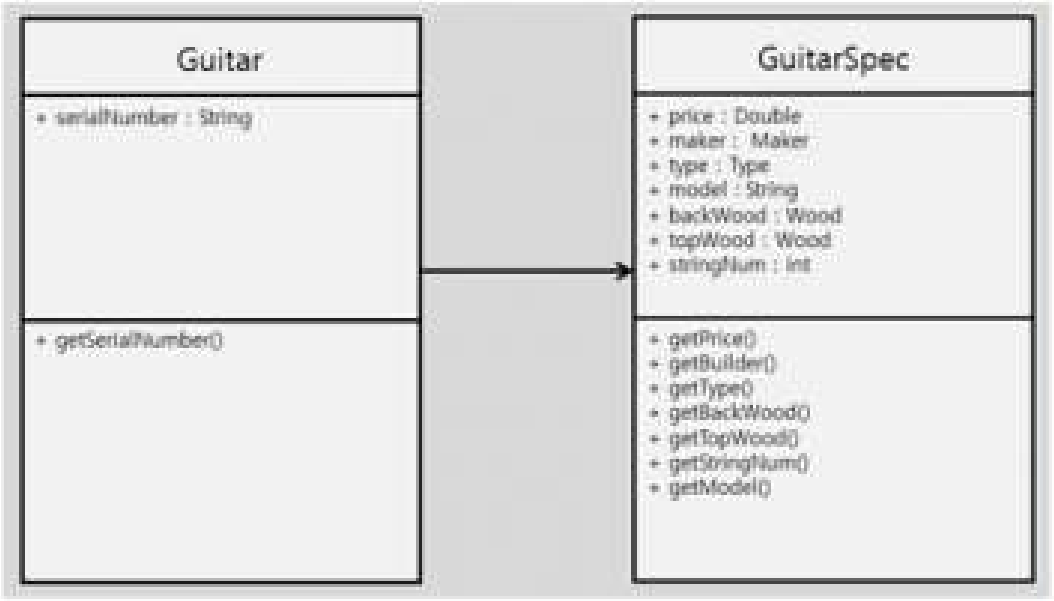
SRP(Single Responsibility Principle) 단일책임 원칙

- 하나의 클래스가 하나의 기능만 가지고 있도록 설계하라 - 독립성
  - ‘하나의 기능 ' 이라는 단어가 너무 추상적인데요
  - ‘하나의 책임’, 혹은 ‘하나의 변경 이유 ' 만을 갖도록 설계하라
    - “The Audience” 찾기
- 가장 기본적인 원칙이지만 가장 직접 구현하기 어려운 원칙

SRP(Single Responsibility Principle) 단일책임 원칙

- 적용방법
  - Extract Class
    - 한 클래스 안에 이의 구조를 변경토록 하는 이유가 둘 이상 존재하는 경우
    - 학생 클래스 : 성적정보, 재학 및 졸업정보, 등록금 납입정보
  - Extract Superclass
    - 클래스를 나누고 보니 유사한 책임을 나눠 맡고 있는 경우
    - 대학생 성적정보, 대학원생 성적정보 클래스 : 성적정보 클래스를 상속하도록
  - Shotgun Surgery
    - 흩어진 메소드들과 필드를 한 클래스로 합침
    - 필요하다면 새 클래스를 만들 수도 있음


```
class Guitar (){

   public Guitar(String serialNumber, double price, Maker
                 maker, Type type, String model, Wood backWood,
                 Wood topWood, int stringNum){
     this.serivalNumber = serialNumber;
     this.price = price;
     this.maker = maker;
     this.type = type;
     this.model = model;
     this.backWood = backWood;
     this.topWood = topWood;
     this.stringNum = stringNum;

   }

  private String serialNumber;
  private double price;
  private Maker maker;
  private Type type;
  private String model;
  private Wood topWood;
  private Wood backWood;
  private int stringNum;

  ....
}
```




```
class Guitar (){

   public Guitar(String serialNumber, GuitarSpec spec){
     this.serivalNumber = serialNumber;
     this.spec = spec;

   }

  private String serialNumber;
  private GuitarSpec spec;

  ....
}

class GuitarSpec(){
   double price;
   Maker maker;
   Type type;
   String model;

   ....
}
```

```php
class Book {

    function getTitle() {
        return "A Great Book";
    }

    function getAuthor() {
        return "John Doe";
    }

    function turnPage() {
        // pointer to next page
    }

    function printCurrentPage() {
        echo "current page content";
    }
}
```

- Audiences:
  - 책 관리자
  - 책 표현 객체
  - 책 파일 저장

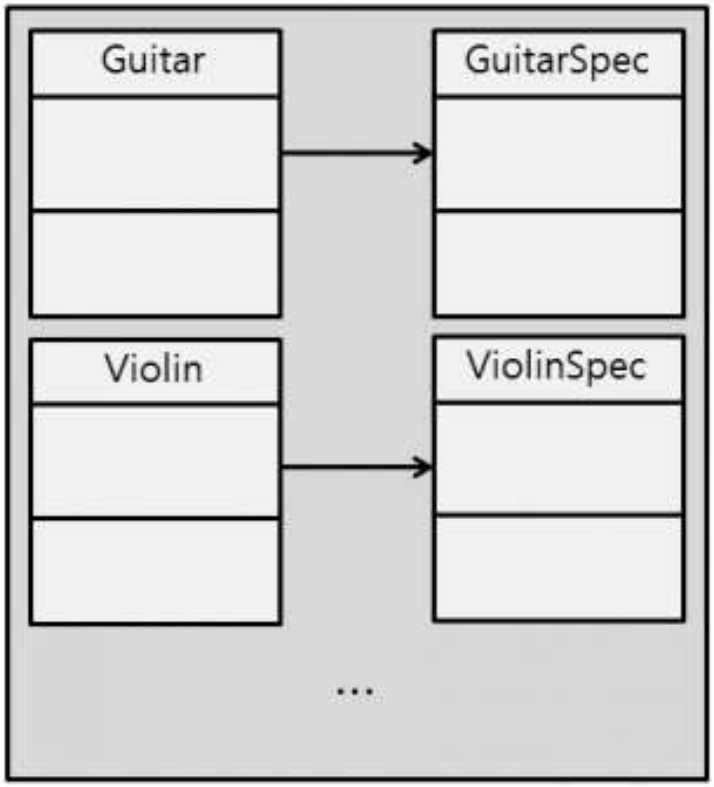
```php
class Book {

    function getTitle() {
        return "A Great Book";
    }

    function getAuthor() {
        return "John Doe";
    }

    function turnPage() {
        // pointer to next page
    }

    function getCurrentPage() {
        return "current page content";
    }

}
```

```php
interface Printer {

    function printPage($page);
}

class PlainTextPrinter implements Printer {

    function printPage($page) {
        echo $page;
    }

}

class HtmlPrinter implements Printer {

    function printPage($page) {
        echo '<div style="single-page">' . $page . '</div>';
    }

}
```

```php
class SimpleFilePersistence {

    function save(Book $book) {
        $filename = '/documents/' . $book->getTitle() . ' - ' . $book->
        file_put_contents($filename, serialize($book));
    }

}
```

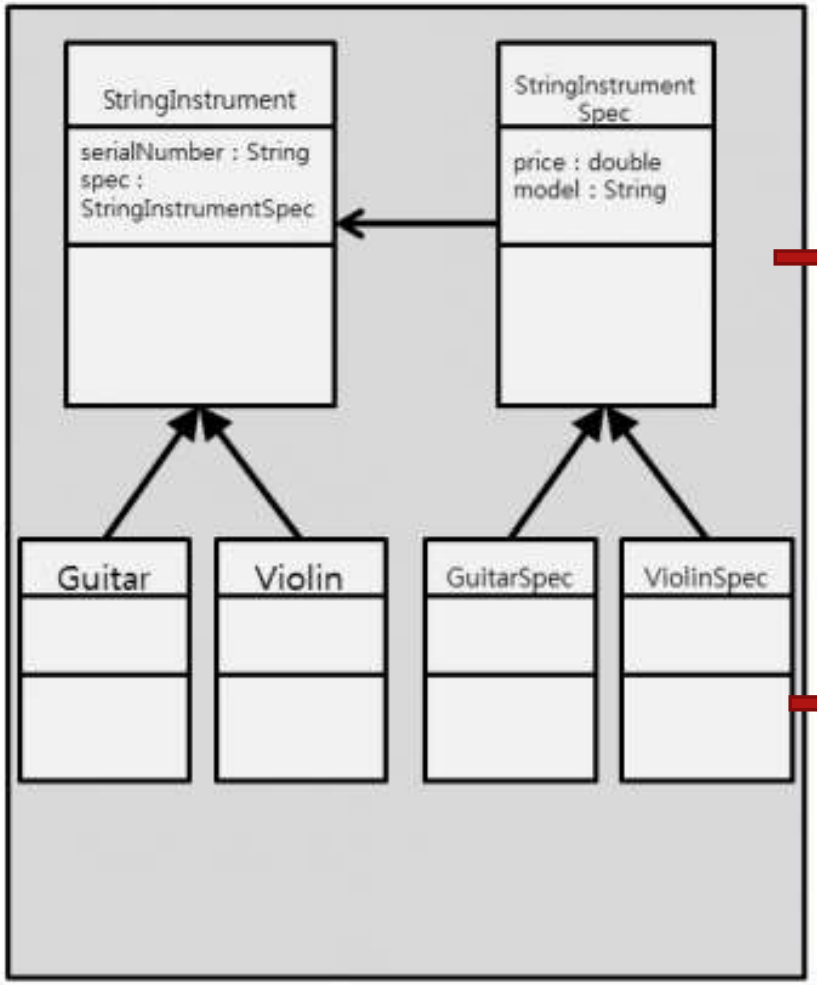
OCP (개방/폐쇄의 원칙: Open Close Principle)

- 무엇에 대해 Open : **확장**
- 무엇에 대해 Close : **변경**
- 코드를 **변경**하지 않고 **확장**할 수 있도록 설계
  - 요구사항 변경 발생 : 기존 코드를 수정하는 대신 새 클래스를 만들어 붙이거나 상속 등을 통해 클래스 재사용
- 변경(확장) 대상과 불변 대상을 명확히 구분 (클래스 나눔)
- 변경 대상과 불변 대상 사이에 인터페이스 정의
- 구상 클래스 대신 인터페이스를 통해 코드 작성
  - 디자인 패턴 가운데 전략 패턴Strategy Pattern




```
class Guitar (){

 public Guitar(String serialNumber, GuitarSpec spec){
     this.serivalNumber = serialNumber;
     this.spec = spec;
   }
 private GuitarSpec spec;
}

class GuitarSpec(){
}

class Violin (){

 public Violin (String serialNumber, ViolinSpec spec){
     this.serivalNumber = serialNumber;
     this.spec = spec;
   }
 private ViolinSpec spec;
}

class ViolinSpec(){
}
                                        ...
```

- 변경 대상은?
- 불변 대상은?

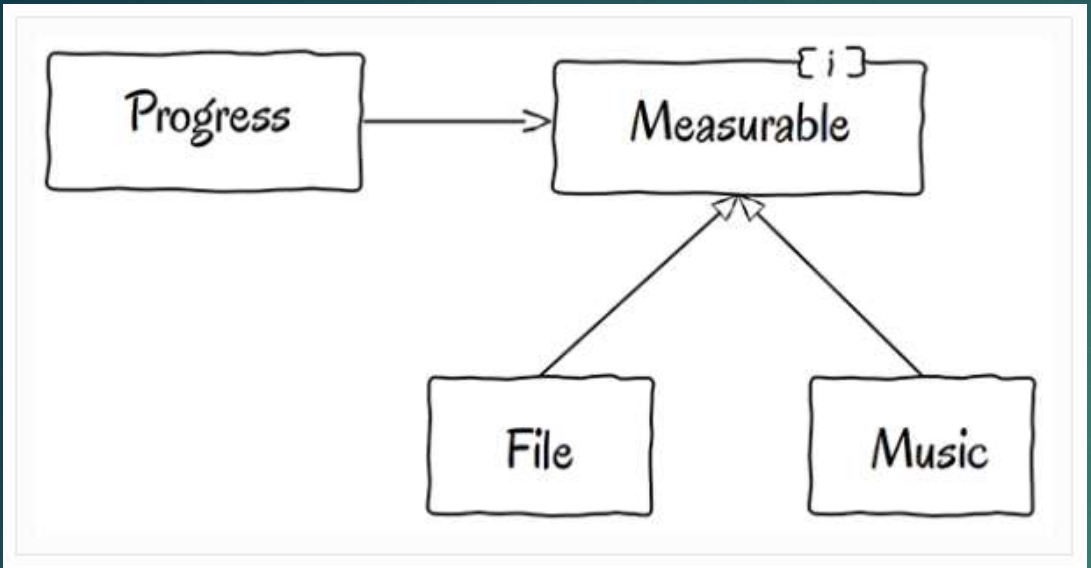



```
class Guitar extends StringInstrument(){

 public Guitar(String serialNumber, GuitarSpec spec){
     this.serivalNumber = serialNumber;
     this.spec = spec;
   }
 private GuitarSpec spec;
}

class GuitarSpec extends StringInstrumentSpec(){
}

class Violin extends StringInstrument(){

 public Violin(String serialNumber, ViolinSpec spec){
     this.serivalNumber = serialNumber;
     this.spec = spec;
   }
 private ViolinSpec spec;
}

class ViolinSpec extends StringInstrumentSpec(){
}
                                    ...
```

```php
class Progress {

    private $file;

    function __construct(File $file) {
        $this->file = $file;
    }

    function getAsPercent() {
        return $this->file->sent * 100 / $this->file->length;
    }

}
```

```php
class File {
    public $length;
    public $sent;
}
```

- OCP가 위배되고 있는가?
- 요구사항 변경 : Progress를 파일 말고 음악에 대해서도 진척도를 출력하게 하려면?
  - 동적 타입 언어, 정적 언어 타입 언어에 따라 다름



불변

```php
interface Measurable {
    function getLength();
    function getSent();
}
```

```php
class File implements Measurable {

    private $length;
    private $sent;

    public $filename;
    public $owner;
```

```php
class Progress {

    private $measurableContent;

    function __construct(Measurable $measurableContent) {
        $this->measurableContent = $measurableContent;
    }

    function getAsPercent() {
        return $this->measurableContent->getSent() * 100 / $this->measurableContent
    }

}
```

LSP (리스코프 치환 원칙: The Liskov Substitution Principle)

- 자식 타입은 부모 타입의 정의를 위반해서는 안된다
- 리스코프 치환 : 자식 타입 클래스는 언제나 부모 타입 클래스로 바꿔끼울 수 있어야 한다
- Programming in Interface
  - 구상 타입을 사용하지 말고 언제나 (가능하다면)인터페이스를 사용해서 프로그래밍 해라
- 적용방법
  - 복수의 객체가 같은 일을 한다면 둘을 하나의 클래스로 묶고 이들을 구분할 수 있는 필드 생성 ex) 직사각형, 정사각형
  - 같은 연산을 약간씩 다르게 한다면 공통 인터페이스를 만들고 이를 구현 ex) 사각형, 원
  - 두 개체가 공통 연산 외에 약간의 차이를 가진다면, 상속을 통해 구현 ex) 가득 찬 원, 빈 원

Override 주의보

- 부모클래스의 메소드를 Override를 이용해 재정의할 때, 그 양상이 완전히 달라진다면(LSP가 깨질 정도라면):
  - 인터페이스를 이용해야 했거나
  - 사실은 부모-자식 사이가 아니었거나

좋은 예

```php
class Vehicle {

    function startEngine() {
        // Default engine start functionality
    }

    function accelerate() {
        // Default acceleration functionality
    }
}
```

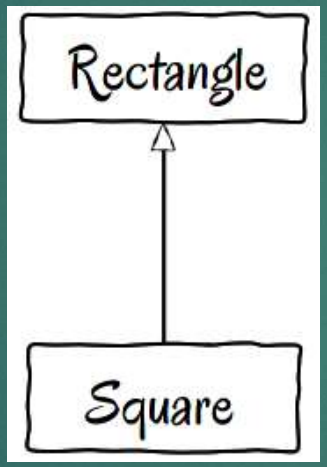
```php
class Driver {
    function go(Vehicle $v) {
        $v->startEngine();
        $v->accelerate();
    }
}
```

사용

```php
class Car extends Vehicle {

    function startEngine() {
        $this->engageIgnition();
        parent::startEngine();
    }

    private function engageIgnition() {
        // Ignition procedure
    }

}

class ElectricBus extends Vehicle {

    function accelerate() {
        $this->increaseVoltage();
        $this->connectIndividualEngines();
    }

    private function increaseVoltage() {
        // Electric logic
    }

    private function connectIndividualEngines() {
        // Connection logic
    }

}
```

나쁜 예

```php
class Rectangle {

    private $topLeft;
    private $width;
    private $height;

    public function setHeight($height) {
        $this->height = $height;
    }

    public function getHeight() {
        return $this->height;
    }

    public function setWidth($width) {
        $this->width = $width;
    }

    public function getWidth() {
        return $this->width;
    }

}
```




```php
class Square extends Rectangle {

    public function setHeight($value) {
        $this->width = $value;
        $this->height = $value;
    }

    public function setWidth($value) {
        $this->width = $value;
        $this->height = $value;
    }
}
```

IS-A 관계인가 ? **Yes!...but**

IS-A관계가 성립한다고 프로그램에서 까지 그런 것은 아님

나쁜 예

```php
class Rectangle {

    private $topLeft;
    private $width;
    private $height;

    public function setHeight($height) {
        $this->height = $height;
    }

    public function getHeight() {
        return $this->height;
    }

    public function setWidth($width) {
        $this->width = $width;
    }

    public function getWidth() {
        return $this->width;
    }

}
```

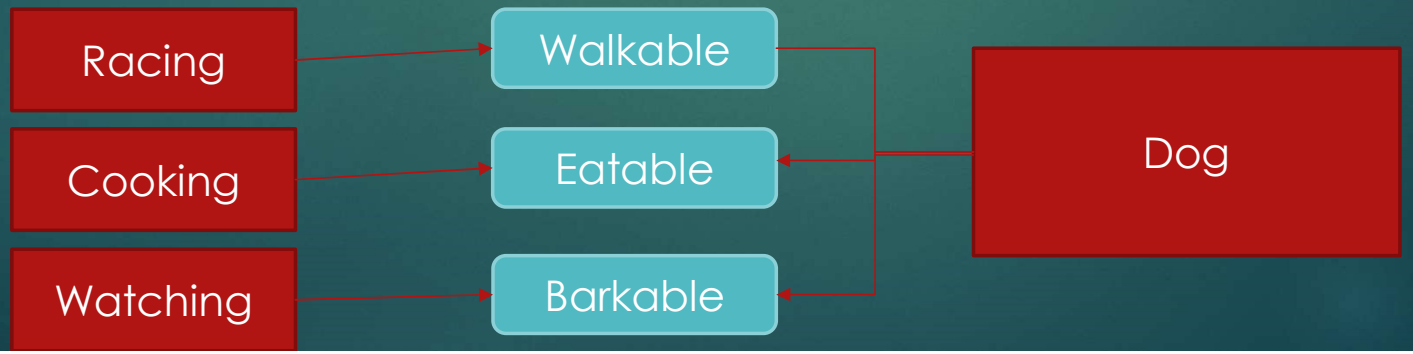
```php
class Square extends Rectangle {

    public function setHeight($value) {
        $this->width = $value;
        $this->height = $value;
    }

    public function setWidth($value) {
        $this->width = $value;
        $this->height = $value;
    }
}
```

```php
class Client {

    function areaVerifier(Rectangle $r) {
        $r->setWidth(5);
        $r->setHeight(4);

        if($r->area() != 20) {
            throw new Exception('Bad area!');
        }

        return true;
    }

}
```

어떻게 고쳐야 할까?

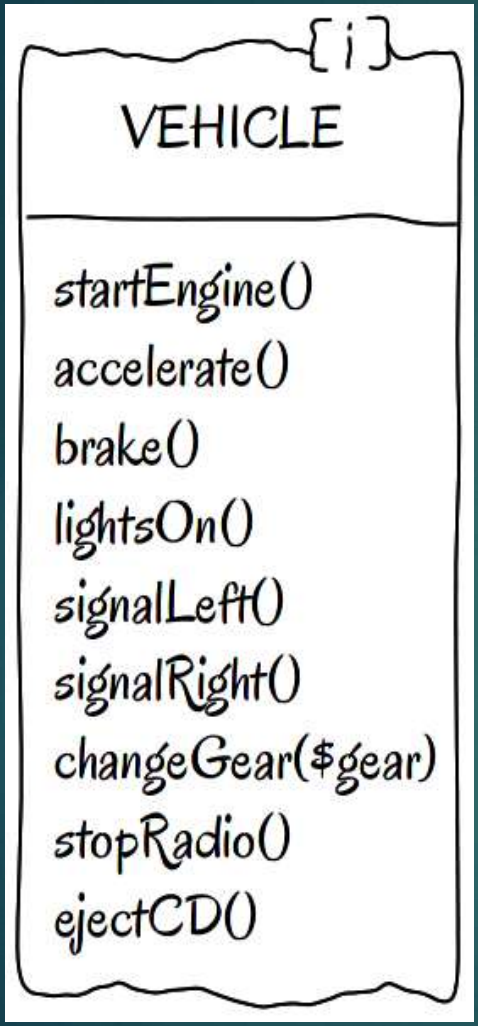
ISP (인터페이스 분리의 원칙: Interface Segregation Principle)

- 자신이 사용하지 않을 인터페이스는 구현하지 말아라
  - 응집성과 연관
  - 개수가 아니라 크기 이야기
- 하나의 일반적인 인터페이스보다는, 여러 개의 구체적인 인터페이스가 낫다
  - 인터페이스의 단일 책임 이야기
  - Animal 인터페이스보다는 Barkable, Walkable, Eatable 인터페이스가 낫다

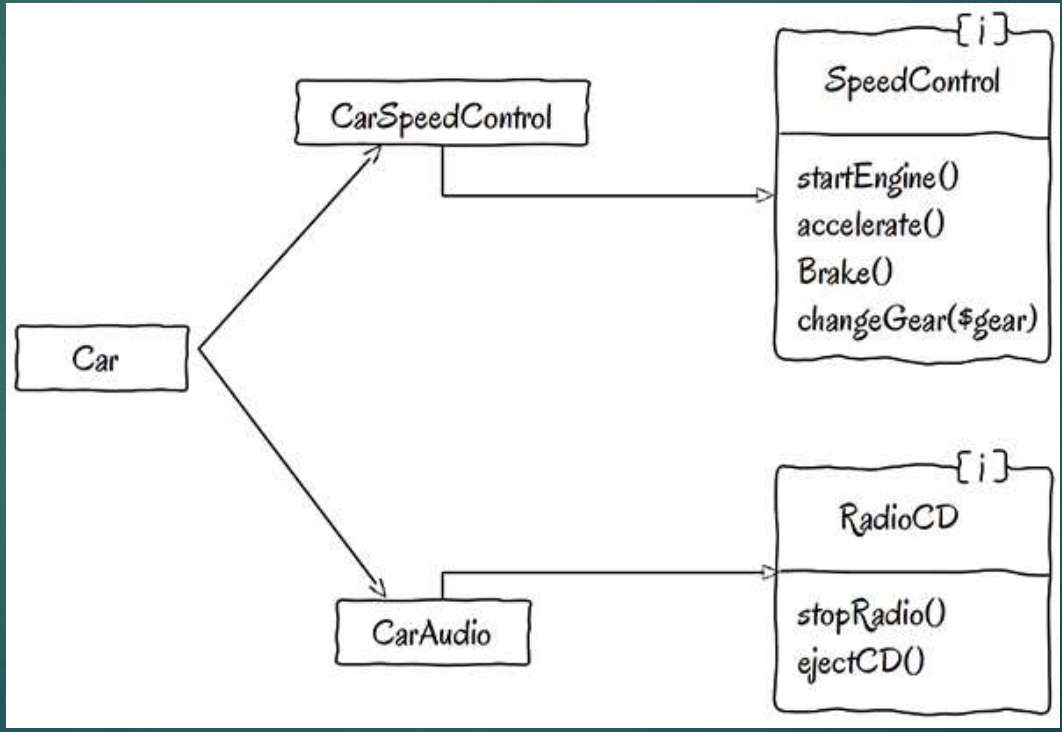


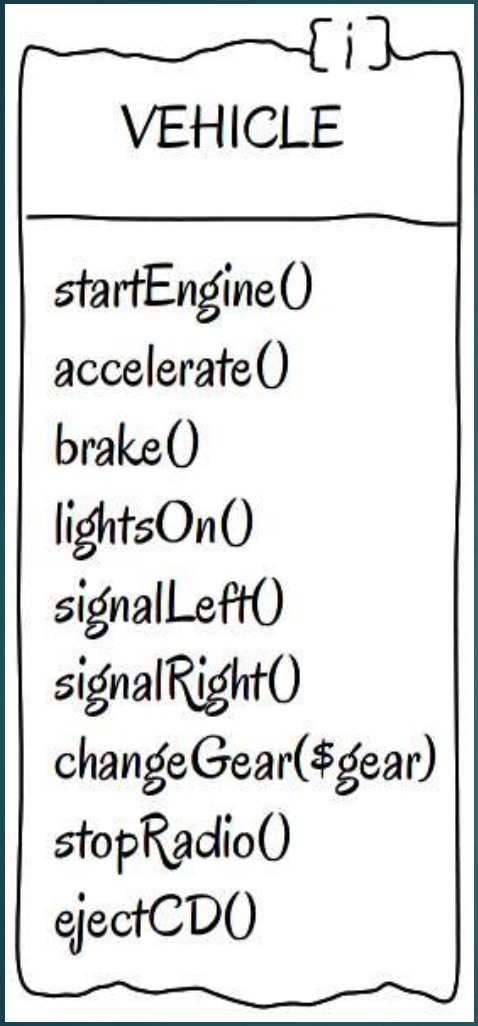
ISP (인터페이스 분리의 원칙: Interface Segregation Principle)

- 적용 방법
  - 클래스 상속을 통한 인터페이스 분리
  - 상속 대신 위임을 사용
    - 퍼사드 패턴Façade Pattern

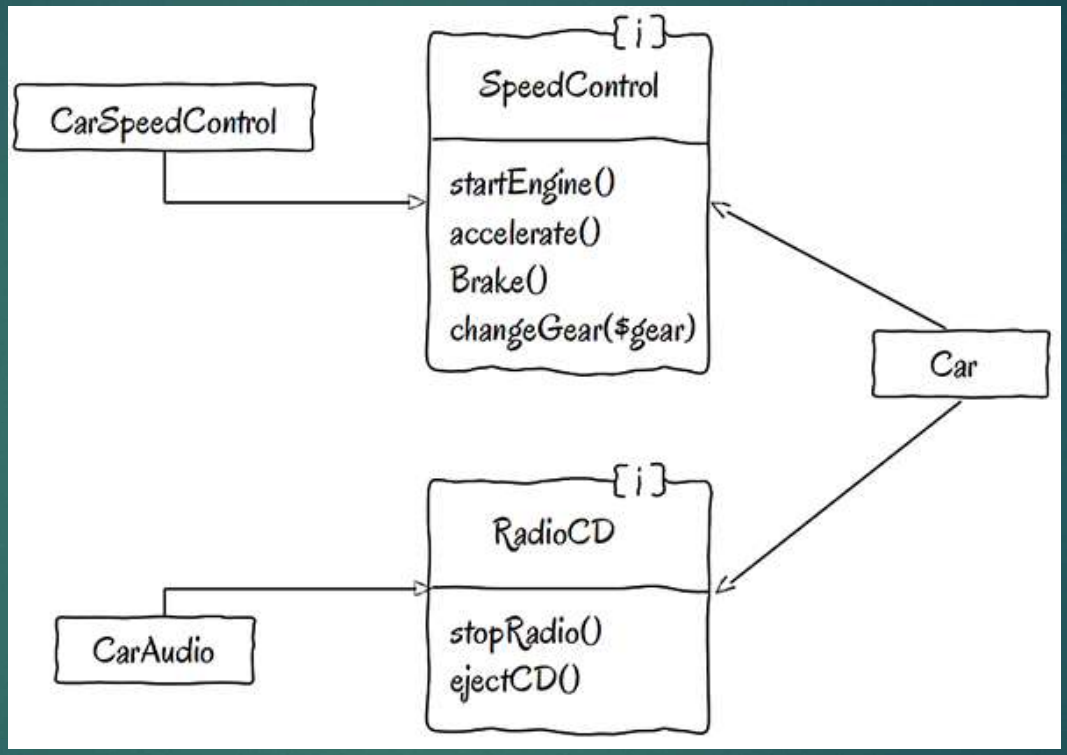


- 1. Bus/Taxi 같은 클래스를 통짜로 작성하여 Vehicle의 모든 메소드 구현
- 2. SpeedControl, CarAudio같은 작은 클래스를 이용해 Vehicle의 책임을 분할

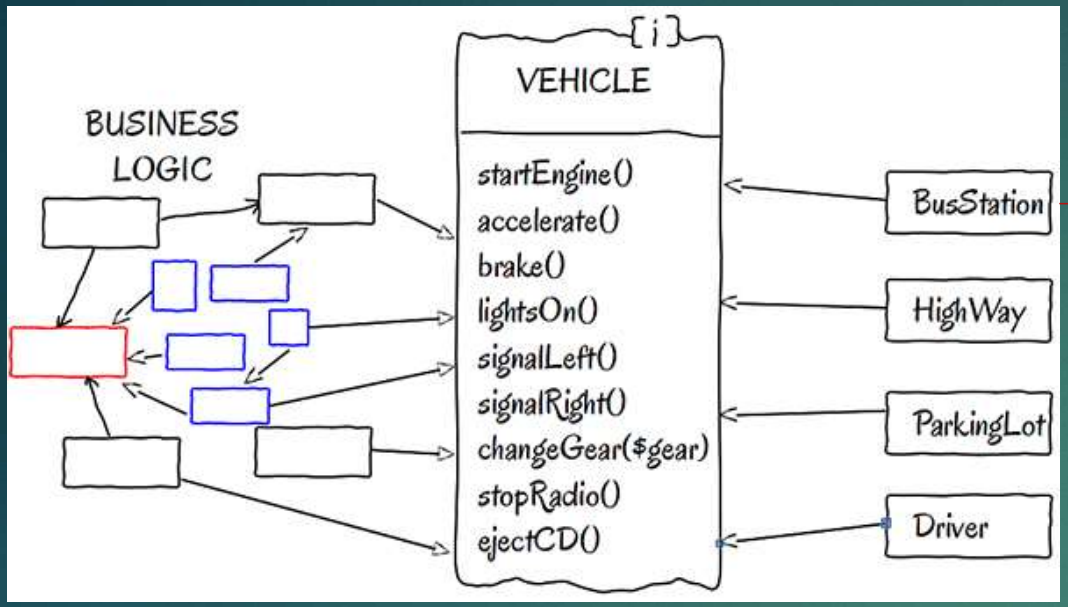


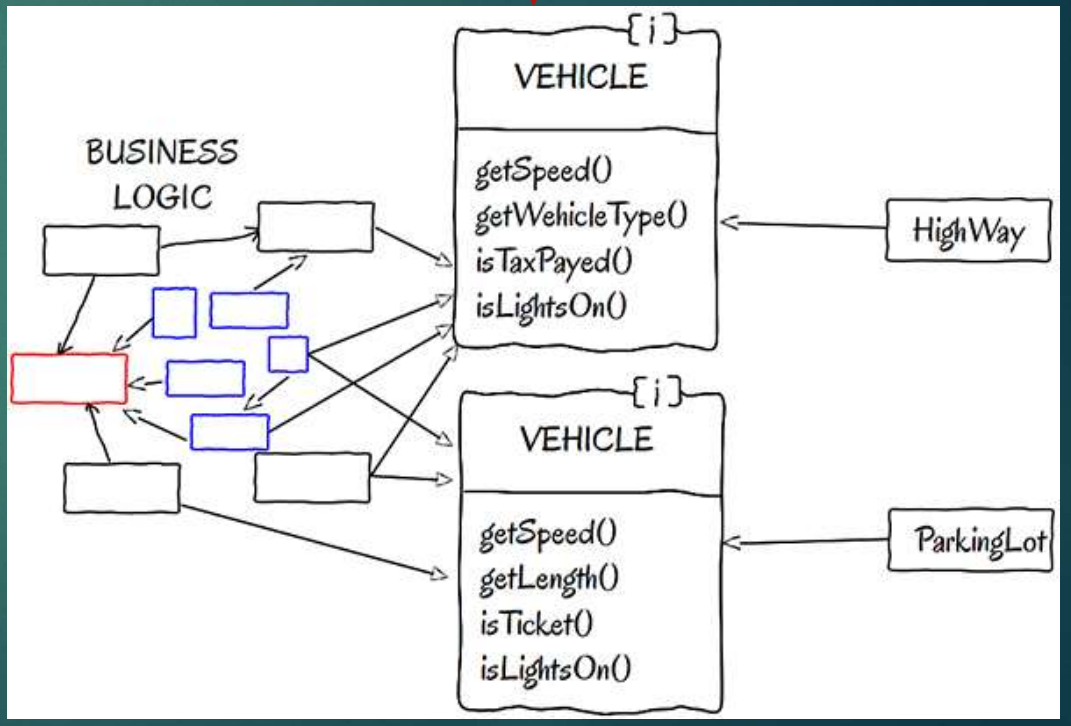


- **더 나은 방법**: vehicle을 더 작은 인터페이스로 분할



- 그러나 이미 Vehicle에 의존하고 있는 다른 클라이언트들은?





The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.

ParkingLot 클래스가 ejectCD 메소드를 호출할 일이 있을까?

```
public class SimpleTableDemo ... implements TableModelListener {
    ...
    public SimpleTableDemo() {
        ...
        table.getModel().addTableModelListener(this);
        ...
    }
    //인터페이스를 통해 노출할 기능을 구현합니다.
    public void tableChanged(TableModelEvent e) {
        int row = e.getFirstRow();
        int column = e.getColumn();
        TableModel model = (TableModel)e.getSource();
        String columnName = model.getColumnName(column);
        Object data = model.getValueAt(row, column);

        ...// Do something with the data...
    }
    ...
}
```

- SimpleTableDemo를 사용해 전체 기능 공개 가능
- 이벤트핸들러에게는 TableModelListner 인터페이스만으로 필요한 메소드 노출 (tableChanged만)

DIP (의존성역전의 원칙: Dependency Inversion Principle)

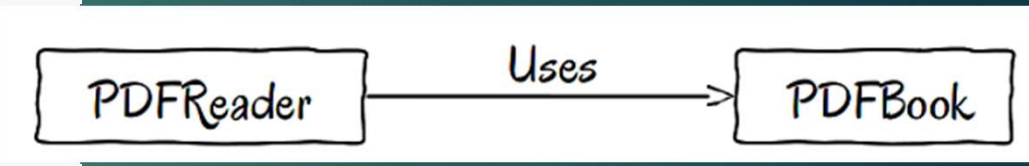
- 할리우드 원칙, Inversion of Control, Dependency Injection이라고도 함
- Caller가 Callee에 의존해야 할까, Callee가 Caller에 의존해야 할까?
- 비유 : 할리우드 배우(Callee)와 캐스팅 디렉터(Caller)가 있을때..
  - 디렉터가 교체되면 배우가 변경되어야 하는가?
  - 배우가 바뀌면 디렉터가 교체되어야 하는가?

"Don't call us, we'll call you"

- 하위 모듈이 상위 모듈의 변경을 요구해서는 안 되지만 구현하다 보면 자주 발생하는 일
  - Ex) Audio 모듈이 Speaker 모듈의 speak() 메소드를 사용하고 있었는데, Speaker 모듈이 Headphone 모듈로 바뀌려면 -> Audio의 변경 발생

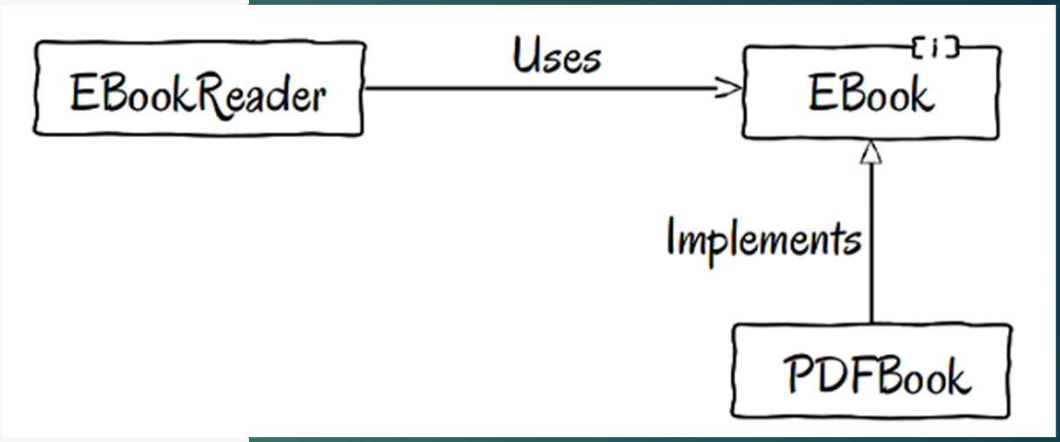
```php
class Test extends PHPUnit_Framework_TestCase {

    function testItCanReadAPDFBook() {
        $b = new PDFBook();
        $r = new PDFReader($b);

        $this->assertRegExp('/pdf book/', $r->read());
    }

}

class PDFReader {

    private $book;

    function __construct(PDFBook $book) {
        $this->book = $book;
    }

    function read() {
        return $this->book->read();
    }

}

class PDFBook {

    function read() {
        return "reading a pdf book.";
    }
}
```

PDFBook->read() 메소드가 변경된다면?



```php
interface EBook {
    function read();
}

class EBookReader {

    private $book;

    function __construct(EBook $book) {
        $this->book = $book;
    }

    function read() {
        return $this->book->read();
    }

}

class PDFBook implements EBook {

    function read() {
        return "reading a pdf book.";
    }
}

class MobiBook implements EBook {

    function read() {
        return "reading a mobi book.";
    }
}
```

추상화 계층 삽입



좋은 설계 훈련하기

- 평소 설계할 때 좋은 설계를 생각하는 훈련을 하지 않으면 평생 절대 늘지 않음
  - 나쁜 설계라도 코드가 돌아는 가니까...
- 나중에 큰 코드 유지보수하다 큰 코 다침
  - 4만라인짜리 클래스 고쳐보실래요?