

# 185p. ~ 191p.

## ▼ 변경하기 쉬운 클래스



대다수 시스템은 지속적인 변경이 가해진다. 그리고 뭔가 변경할 때마다 시스템이 의도대로 동작하지 않을 위험이 따른다. (185p)

```
public class Sql {  
    public Sql(String table, Column[] columns)  
    public String create()  
    public String insert(Object[] fields)  
    public String selectAll()  
    public String findByKey(String keyColumn, String keyValue)  
    public String select(Column column, String pattern)  
    public String select(Criteria criteria)  
    public String preparedInsert()  
    private String columnList(Column[] columns)  
    private String valuesList(Object[] fields, final Column[] columns)  
    private String selectWithCriteria(String criteria)  
    private String placeholderList(Column[] columns)  
}
```

위에 코드에서 변경이 필요한 순간

1. update와 같은 새로운 SQL문 지원
2. 기존 SQL문 수정(ex. select 문에 내장된 select 문 지원)

변경할 이유가 2가지이므로 위에 Sql클래스는 SRP를 위반한다.

SRP(Single Responsibility Principle) : 클래스나 모듈을 변경할 이유가 단 하나뿐이어야 한다는 원칙

```
abstract public class Sql {  
    public Sql(String table, Column[] columns)  
    abstract public String generate();  
}
```

```

public class CreateSql extends Sql {
    public CreateSql(String table, Column[] columns)
    @Override public String generate()
}

public class SelectSql extends Sql {
    public SelectSql(String table, Column[] columns)
    @Override public String generate()
}

public class InsertSql extends Sql {
    public InsertSql(String table, Column[] columns, Object[] fields)
    @Override public String generate()
    private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
    public SelectWithCriteriaSql(
        String table, Column[] columns, Criteria criteria)
    @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
    public SelectWithMatchSql(
        String table, Column[] columns, Column column, String pattern)
    @Override public String generate()
}

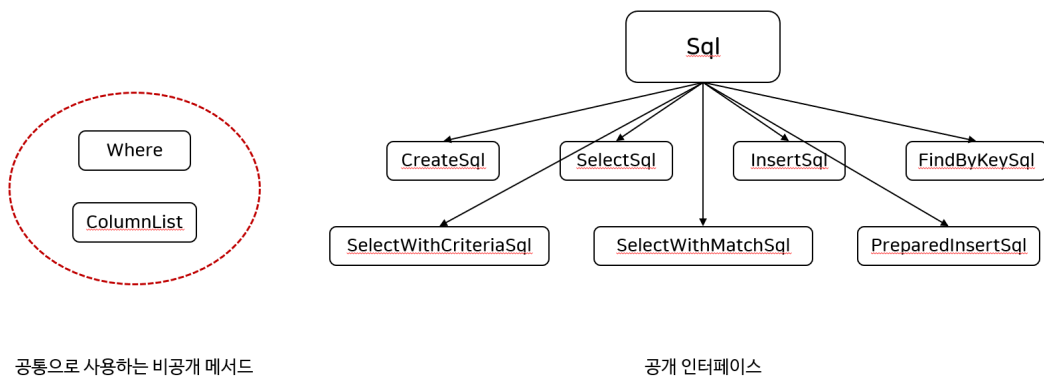
public class FindByKeySql extends Sql
    public FindByKeySql(
        String table, Column[] columns, String keyColumn, String keyValue)
    @Override public String generate()
}

public class PreparedInsertSql extends Sql {
    public PreparedInsertSql(String table, Column[] columns)
    @Override public String generate()
    private String placeholderList(Column[] columns)
}

public class Where {
    public Where(String criteria)
    public String generate()
}

public class ColumnList {
    public ColumnList(Column[] columns)
    public String generate()
}

```



위와 같은 구조의 장점

1. 코드가 순식간에 이해됨
2. 함수 하나를 수정했다고 다른 함수가 망가질 위험도 사라짐
3. 테스트 관점에서 모든 논리를 구석구석 증명하기도 쉬워짐
4. update문을 추가할 때 기존 클래스를 변경할 필요가 전혀 없음
5. SRP와 OCP도 지원

위와 같은 구조는 세상의 모든 장점만 취한다!

OCP(Open-Closed Principle) : 확장에 개방적이고 수정에 폐쇄적이어야 한다는 원칙

### **변경으로부터 격리**

상세한 구현에 의존하는 코드는 테스트가 어렵다.

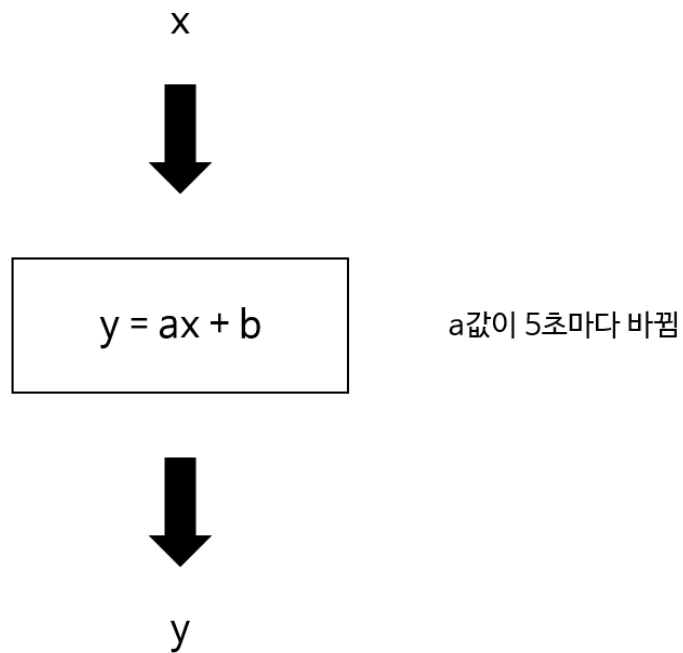
Portfolio 클래스를 만든다고 가정하자.

Portfolio 클래스는 외부 TokyoStockExchange API를 사용해 포트폴리오 값을 계산한다.

따라서 우리 테스트 코드는 시세 변화에 영향을 받는다.

5분마다 값이 달라지는 API로 테스트 코드를 짜기란 쉽지 않다.

보통 테스트 코드는 일정한 입력에 대한 올바른 출력을 정의하고 확인한다.



같은 x값이라도 시간에 따라 y값이 다르게 나옴

이때 y값이 제대로 나온 값인지 확인하려면 새로 계산해야 됨

이러한 문제는 어떻게 해결할 수 있을까?

### 1. Stock Exchange 인터페이스 생성

```
public interface StockExchange {  
    Money currentPrice(String symbol);  
}
```

### 2. StockExchange 인터페이스를 구현하는 TokyoStockExchange 클래스를 구현

### 3. Portfolio 생성자를 수정해 StockExchange 참조자를 인수로 받음

```
public Portfolio {  
    private StockExchange exchange;  
    public Portfolio(StockExchange exchange) {  
        this.exchange = exchange;  
    }  
}
```

```

    }
    // ...
}

```

#### 4. 테스트용 클래스는 StockExchange 인터페이스를 구현하며 고정된 주가를 반환

```

public class PortfolioTest {
    private FixedStockExchangeStub exchange;
    private Portfolio portfolio;

    @Before
    protected void setUp() throws Exception {
        exchange = new FixedStockExchangeStub();
        exchange.fix("MSFT", 100);
        portfolio = new Portfolio(exchange);
    }

    @Test public void GivenFiveMSFTTotalShouldBe500() throws Exception {
        portfolio.add(5, "MSFT");
        Assert.assertEquals(500, portfolio.value());
    }
}

```

위와 같은 방법으로 시스템의 결합도를 낮추면 유연성과 재사용성도 더욱 높아진다.

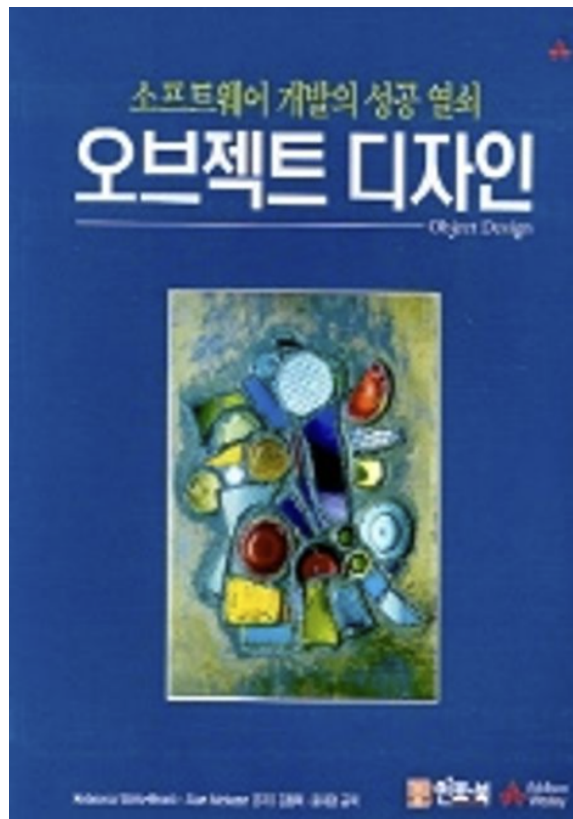
결합도가 낮다 = 각 시스템 요소가 다른 요소로부터 그리고 변경으로부터 잘 격리 되어 있다

결합도를 최소로 줄임 → DIP를 따르는 클래스가 생성됨


DIP(Dependency Inversion Principle) : 클래스가 상세한 구현이 아니라 추상화에 의존해야 한다는 원칙

#### ▼ 참고문헌

[RDD]: Object Design: Roles, Responsibilities, and Collaborations, Rebecca WirfsBrock et al., Addison-Wesley, 2002.



영풍문고 - 서점다운 서점

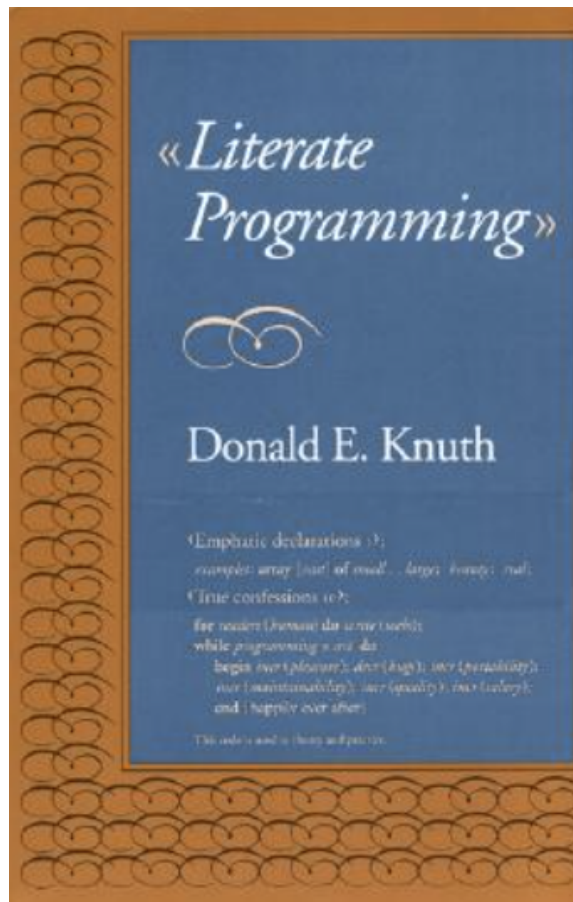
 <https://www.ypbooks.co.kr/book.yp?bookcd=1501301270>

[PPP]: Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.



<https://product.kyobobook.co.kr/detail/S000001875106>

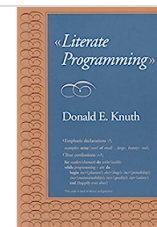
[Knuth92]: Literate Programming, Donald E. Knuth, Center for the Study of language and Information, Leland Stanford Junior University, 1992.



### Literate Programming (Volume 27) (Lecture Notes)

Literate Programming (Volume 27) (Lecture Notes) [Knuth, Donald E.] on Amazon.com. \*FREE\* shipping on qualifying offers. Literate Programming (Volume 27) (Lecture Notes)

[a https://www.amazon.com/Literate-Programming-Lecture-Notes-Donald/dp/0937073806](https://www.amazon.com/Literate-Programming-Lecture-Notes-Donald/dp/0937073806)



★★★★★ 19