AOP

```
Logger log = Logger.getLogger("order");
    log.info("orderItem started...");
    // do something
    try {
    doSomething(oi);
    } catch(Exception ex) {
    log.info("Error : " + ex.getMessage());
    } finally {
    log.info("orderItem finished...");
    }
    }
```

- 로직을 구현하는 핵심 소스에 많은 수의 로깅 관련 소스가 삽입됨으로써 소스의 가독성과 간결함을 심하게 훼손한다.
- 로깅을 남기는 방법이나 시점, 위치 등을 바꾸려면 거의 모든 소스파일을 변경해야 한다.

orderItem 메소드가 시작하는 상황에 로그를 기록하라.
orderItem 메소드가 끝나는상황에 로그를 기록하라.
orderItem 메소드내에서 Exception이 발생하는 상황에 로그를 기록하라.

AOP

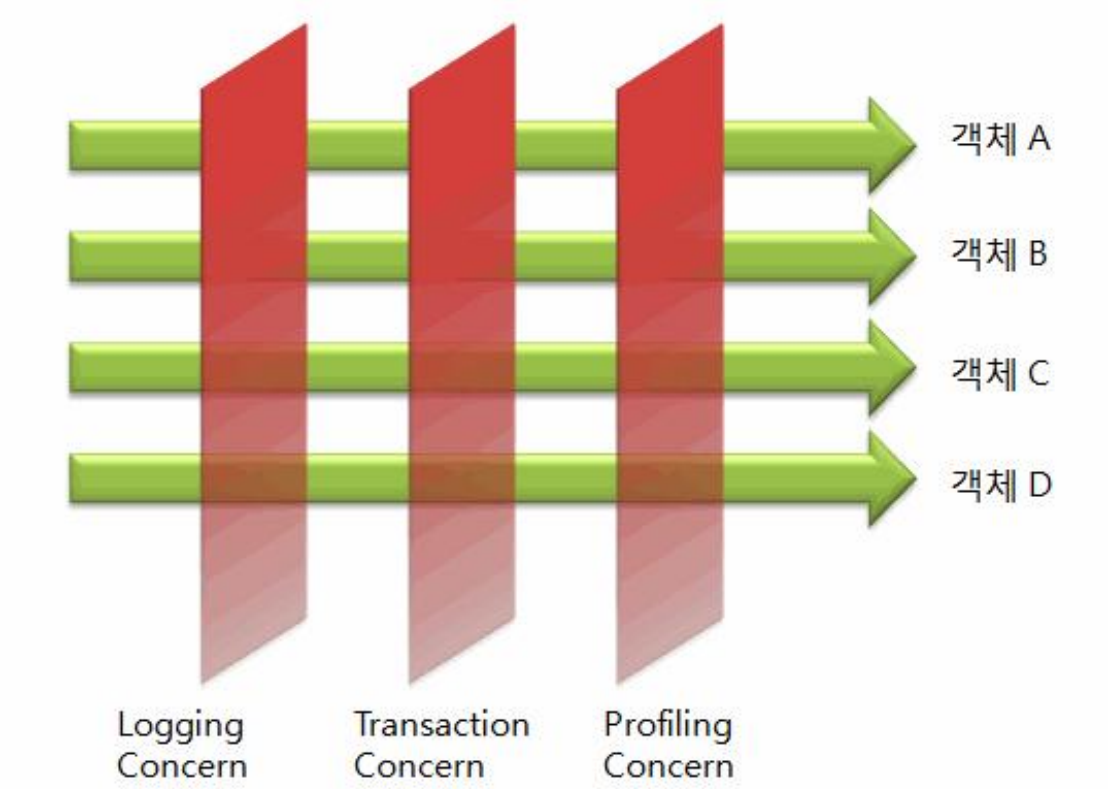
핵심 로직에서는 Logging 관련된 부분이 완전히 제거된다.

```
public class OrderManager {
public void orderItem(OrderItem oi) {
        // do something
        try {
        doSomething(oi);
        } catch(Exception ex) { }
    }

// 핵심 로직에서 발생하는 상황에 대한 로직을 구현한다.
public aspect LoggingAspect {
    // OrderManger내에서 발생하는 모든 Method Call에 대해
    pointcut methodExec() :
        call(* *.*(..)) &&within(OrderManager);

before() : methodExec() {
            Logger log = Logger.getLogger("order");
            log.info(thisJoinPointStaticPart.getSignature() + " started...");
    } //Method Call 시작시수행할 일
...
after() throwing(RuntimeException ex) : methodExec() {
            Logger log = Logger.getLogger("order");
            log.info("Error : " + ex.getMessage());
            } // Exception 발생시
    }
```

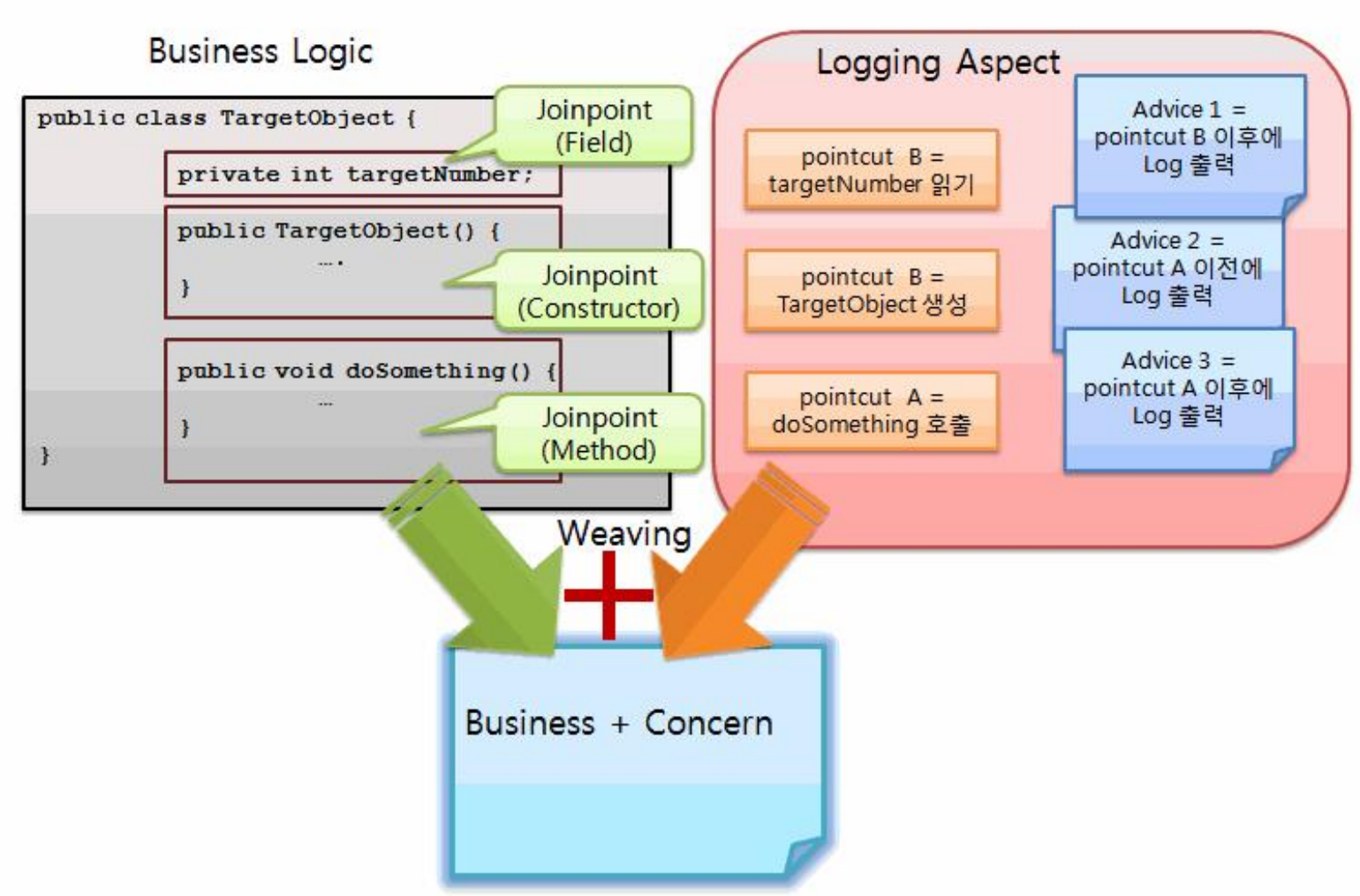
AOP



AspectJ의 용어이다. 하지만 AspectJ가 AOP의 사실상의 표준이기 때문에 무방하다고 간주

joinpoint : 프로그램 수행 과정에서의 특정 지점. 생성자의 호출, 메소드의 호출, 오브젝트 필드에 대한 접근 등의 대표적인 joinpoint들이다.
pointcut : joinpoint와매칭하고자 하는 조건과 그 상황에서의 값들
advice : pointcut에 의해매칭된 joinpoint에서 실행할 작업
aspect : pointcut과 advice의 집합체. 즉 특정 상황(pointcut)과 그 상황에서 수행할 작업(advice)의 집합
weaving : aspect과 핵심 로직을 섞는(weave) 것을 의미

AOP



AspectJ는 Java에서의 AOP 언어의 사실상의 표준이다.
Spring AOP, JBoss AOP 등 새로운 AOP 컴포넌트도 있지만, 주로 AspectJ를 사용한다.

AOP

```java
package sample;

import java.io.IOException;
import java.net.SocketException;

public class ExceptionGenerator {

    public void doException1() throws RuntimeException {
        throw new RuntimeException("Runtime Exception");
    }

    public void doException2() throws java.io.IOException {
        throw new java.io.IOException("IO Exception");
    }

    public void doException3() throws java.net.SocketException {
        throw new java.net.SocketException("Socket Exception");
    }

    public static void main(String[] args) {
        ExceptionGenerator eg = new ExceptionGenerator();
        try { eg.doException1(); } catch (RuntimeException e) { }
        try { eg.doException2(); } catch (IOException e) { }
        try { eg.doException3(); } catch (SocketException e) { }
    }
}
```

- ExceptionGenerator의 로직을 수행하는 과정에서 Exception이 발생하면 이것을 캡쳐해서 기록하고 싶다.
- 이 때 어떤 메소드를 호출하다가 Exception이 발생했는지, Exception의 종류는 무엇인지 등의 정보가 종합적으로 기록하고 싶다.

이 상황을 AOP 없이 처리하려면

```
try { doSomething1() }
    catch(Exception ex) {
logger.log("Error " + ex + "
    occured when executing
    ExceptionGenerator.doSo
    mething1()...");
}
```

AOP

```
1  package sample;
2
3  public aspect ExceptionAspect {
4
5      pointcut callpoint() : call(* *.*(..));
6
7      after() throwing : callpoint() {
8
9          System.out.println("Something Bad Happend...");
10
11     }
12 }
```

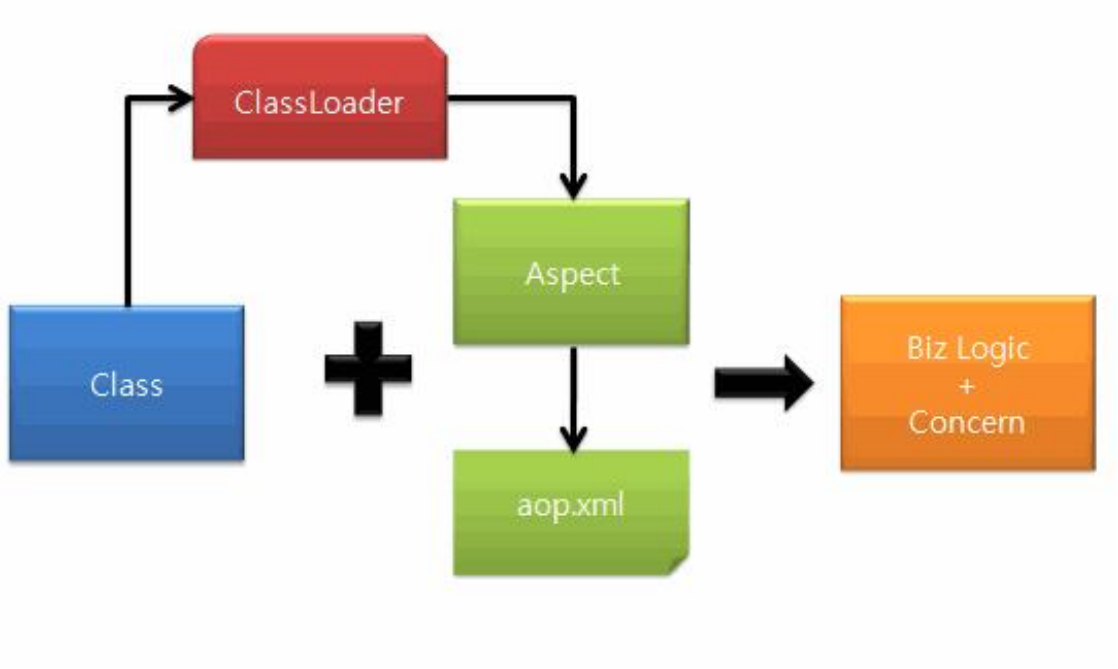
AOP

```
1  package sample;
2
3  public aspect ExceptionAspect {
4
5      pointcut callpoint() : call(* *.*(..));
6
7      after() throwing(Exception ex) : callpoint() {
8
9          System.out.println("[Error]" + ex);
10         System.out.println("\t for calling " + thisJoinPointStaticPart.getSignature())
11
12         StackTraceElement[] ste = ex.getStackTrace();
13         for(int i=0; i<ste.length; i++) {
14             System.out.print("\t");
15             for(int j=0; j<i; j++) System.out.print("\t");
16             System.out.println(ste[i]);
17         }
18
19     }
20 }
```

After advice에서 Exception발생시 Exception 객체를 받는다. 이렇게 받은 객체를 이용해서 필요한 정보를 추출한다.

thisJoinPointStaticPart (또는 thisJoinPoint.getStaticPart())를 이용해 어떤 지점에서 발생한 Exception인지를 알아낸다.

AOP



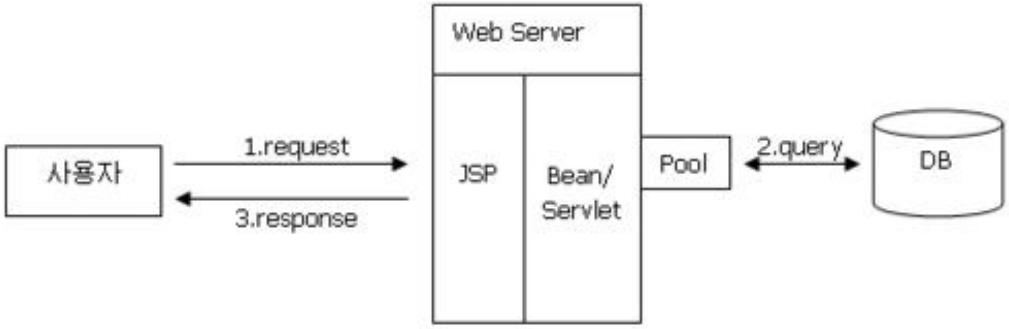
LTW

(**Load Time Weaving**)

MVC

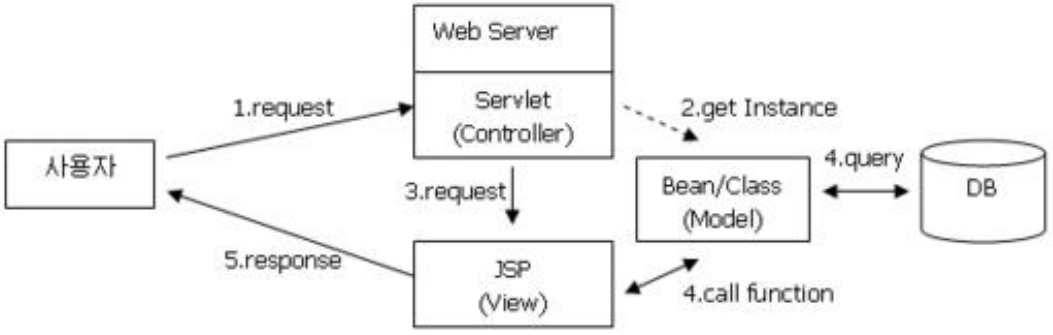
Model 1

- 뷰와 컨트롤러 모두를 하나로 제작

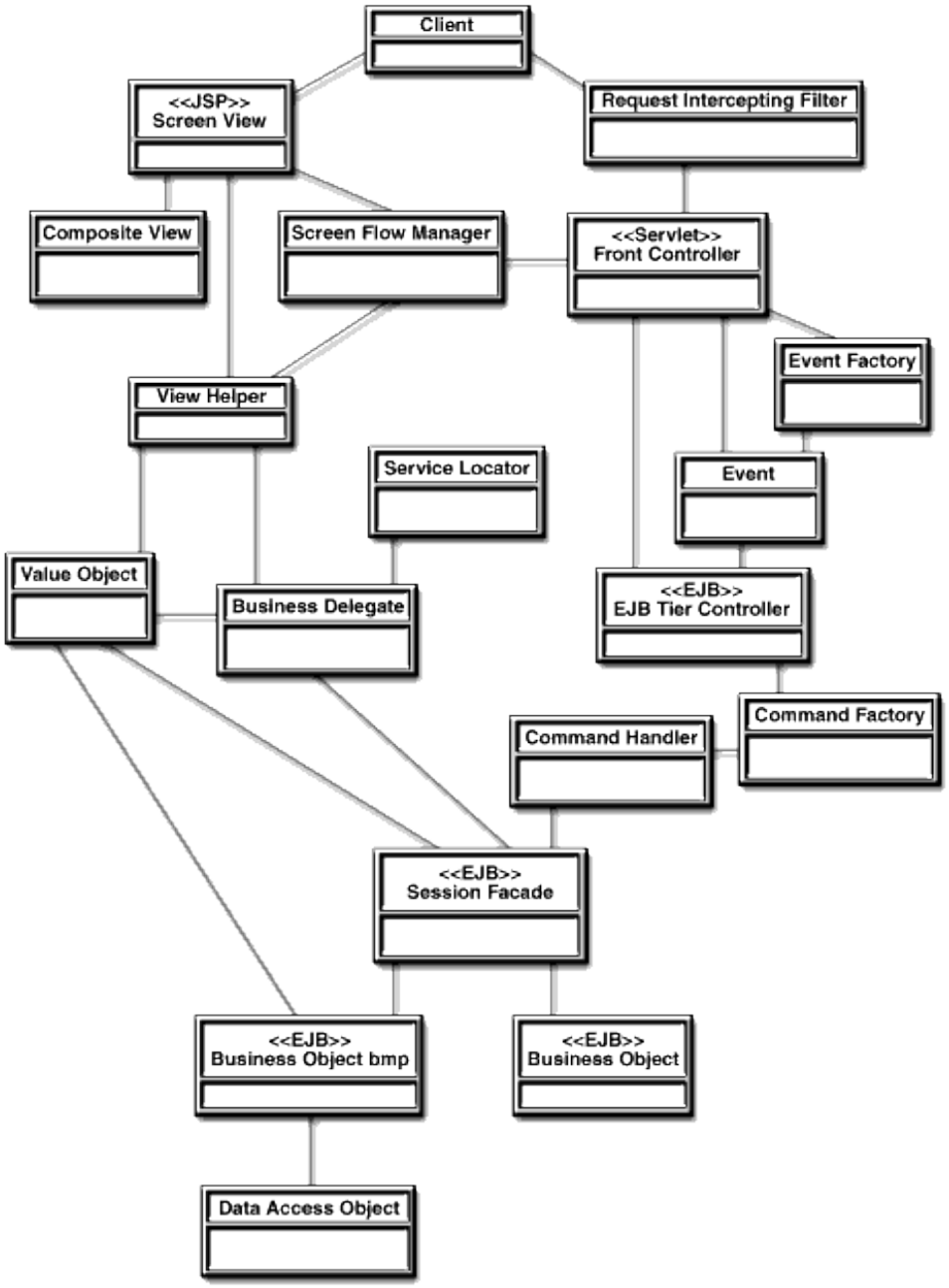


Model 2

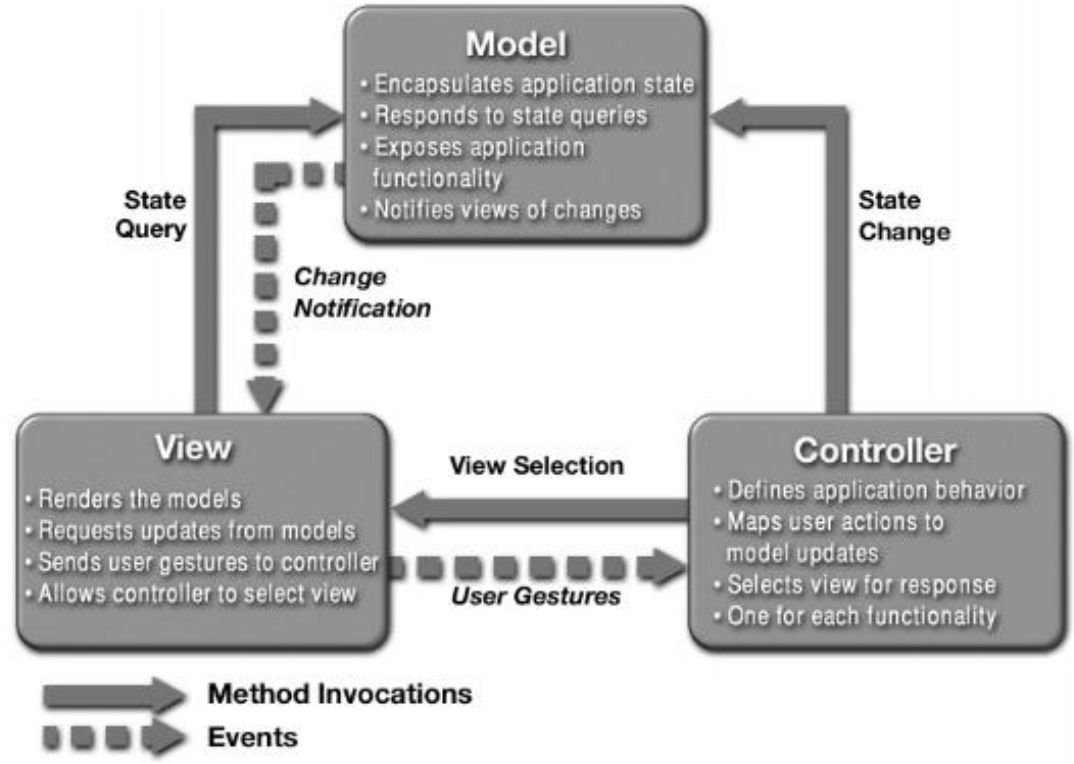
- 컨트롤러를 통해 비즈니스 로직과 뷰가 분리된 형태



MVC


Client
<<JSP>>
Screen View
Request Intercepting Filter
Composite View
Screen Flow Manager
<<Servlet>>
Front Controller
Event Factory
View Helper
Service Locator
Event
Value Object
Business Delegate
<<EJB>>
EJB Tier Controller
Command Factory
Command Handler
<<EJB>>
Session Facade
<<EJB>>
Business Object bmp
<<EJB>>
Business Object
Data Access Object

MVC



Model - The model represents enterprise data and the business rules that govern access to and updates of this data. Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.

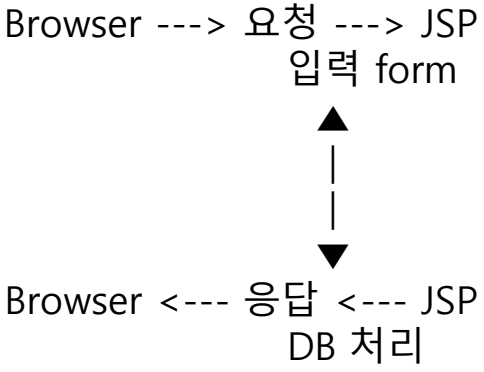
View -The view renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented. It is the view's responsibility to maintain consistency in its presentation when the model changes. This can be achieved by using a push model, where the view registers itself with the model for change notifications, or a pull model, where the view is responsible for calling the model when it needs to retrieve the most current data.

Controller - The controller translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET and POST HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

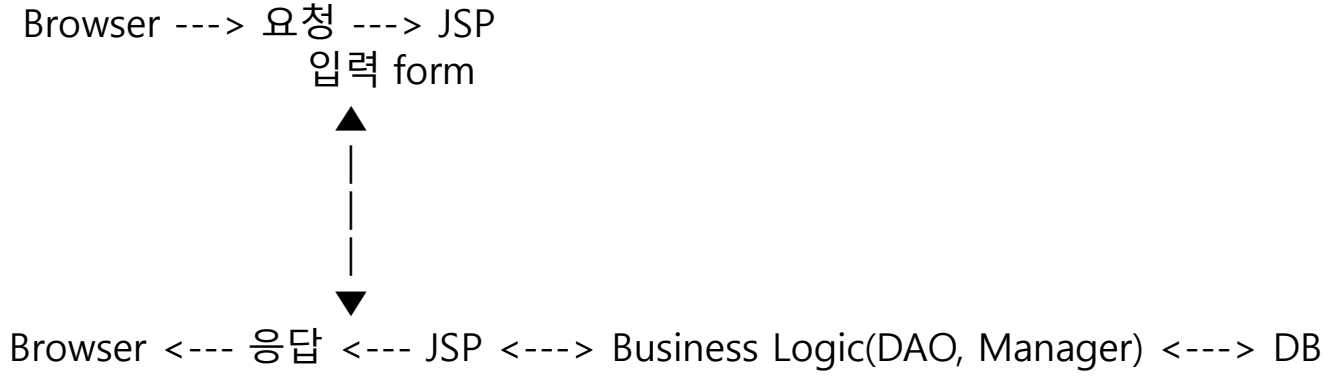
MVC

Model 1의 구조

- Model 1의 구조



- 변형된 Model 1의 구조



MVC

MVC Model2 구조

- request 객체(JSP 내부 객체)의 사용
  . request 객체는 클라이언트로부터 값을 가져오는 목적으로 사용되나 값을 출력하는 목적으로도 사용된다.
  . 저장소 역활을 한다.

예)

값을 가져오는 형태

```
String name = request.getParameter("name");
```

값을 request에 저장하는 형태

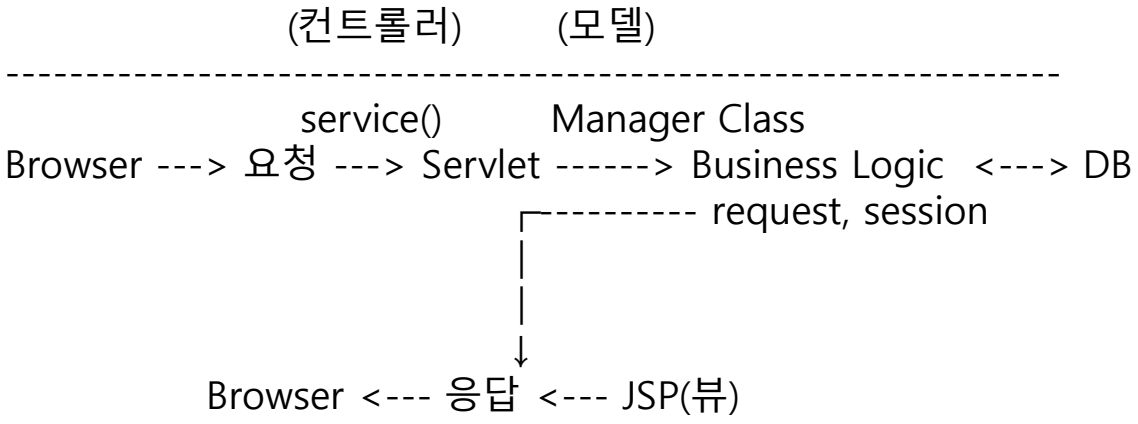
```
MemberDTO dto = new MemberDTO();
request.setAttribute("dto", dto);
```

- 컨트롤러(Controller)
  . 어플리케이션의 흐름 제어나 사용자의 처리 요청을 구현한다.
  . 사용자 인증, 보안 설정 등 전체 어플리케이션에 영향을 미치는 요소를 구현한다.
  . 사용자의 요청에 대해서 알맞은 모델을 사용하고 사용자에게 보여줄 뷰를 선택하기만 하면
  된다.
  . Servlet 으로 제작

- 모델(Model)
  . 비즈니스 로직과 관련된 부분만 처리한다.
  . 데이터베이스 관련 로직을 구현한다.
  . JSP Beans, DAO class, DTO class, Manager class, EJB Component

MVC

- 뷰(View)
  . 사용자에게 알맞은 화면을 보여주는 역활만 수행한다.
  . JSP, EL, JSTL, Custom Tag, XML



MVC

Controller의 처리순서

Browser --------> Servlet(Controller)

1. HTTP 요청 받음, doGet(), doPost() 메소드 호출
2. 클라이언트가 요구하는 기능을 분석(방명록의 경우 글쓰기 등)
3. 요청한 비즈니스 로직을 처리하는 Model 사용, Business Logic Class
4. 결과를 request 또는 session의 setAttribute()메소드를 사용하여 저장
5. 알맞은 뷰 선택 후, 뷰로 포워딩(또는 리다이렉트), jsp 페이지로 이동

↙

JSP

↙

HTTP 응답

↙

Browser

MVC

Command Pattern에 기반한 Controller의 처리순서

- Command Handler
  . Interface, 추상 클래스를 이용하여 구현.
  . JSP Beans class, EJB를 통해서 Command Handler를 구현한다.

Browser --------> Servlet(Controller)

1. HTTP 요청 받음
2. 클라이언트가 요구하는 기능을 분석
3. 로직을 처리할 명령어 핸들어 생성 ---> Command Handler
4. 명령어 핸들러를 이용해 로직 처리 ---> 로직 처리 (Class)

↓

결과 저장

↓

JSP등 뷰 리턴

↓

5. 명령어 핸들러가 리턴한 뷰로 페이지 이동(포워딩, 리다이렉트)

↙

JSP

↙

HTTP 응답

↙

Browser

MVC

수평적인 실행 순서

JSP Model 1

jsp form ---> jsp proc ---> Manager ---> DAO ---> JSP 최종 결과 출력

response

JSP Model 2

jsp form ---> servlet controler ---> Interface Action class ---> Manager ---> DAO -----> JSP 최종 결과 출력

request

request –
↓
response

MVC

MVC Application과 일반 JSP 개발 순서의 비교
- 순수 코딩을 기준으로 한 경우

1. Web Application(JSP Model 1) 개발 순서

① DBMS 설정
② DTO(Data Transfer Object, Domain, Model) Class 제작
③ DAO(Data Access Object, Persistence) Class 제작
④ Manager Class(Process, Business Logic) 제작
⑤ Manager Test
⑥ JSP 생성 연동

2. MVC Web Application(JSP Model 2) 개발 순서

① DBMS 설정
② DTO(Data Transfer Object, Domain, Model) Class 제작
③ DAO(Data Access Object, Persistence) Class 제작
④ Manager Class(Process, Business Logic) 제작
⑤ Manager Test
---
⑥ Servlet Controller 제작
⑦ Action Interface 제작
⑧ Action 구현 Class 제작
⑨ Manager Class 연동
---
⑩ JSP 생성 연동

자바 빈

자바 빈?

자바 빈(Jave Bean)은 아래와 같은 정의를 가지고 있다.

A Java Bean is a reusable software component that can be manipulated visually in a builder tool.
자바 빈은 빌더 툴에서 가시적으로 다루어질 수 있는 재사용 가능한 소프트웨어 부품이다.

간단히 몇 가지 특징을 빌어 정의하자면 데이터 집합의 표현을 위한 클래스 정도로 생각하면 될 것이다.

자바 빈은 크게 아래와 같은 특징들을 지닌다.

- 멤버변수의 값을 할당하거나 추출하는 기능을 가진다.
- 비즈니스 로직과 표현을 위해 코드를 분리하는 역할을 한다.
- 컴포넌트의 재사용성을 높인다.

일반적인 특징입니다.

- 확장자는 *.java 입니다. 컴파일하여 .class형태로 배포합니다.
- 메모리에 생성된 빈즈(DTO)는 다른 자바 클래스(DAO(Data Access Object), Business Logic)에 의해서 사용됩니다.
- 빈즈는 dll과 같은 원리를 가지고 있습니다.
- 빈즈는 sun에서 제시한 작성 규칙이 존재합니다.

자바 빈

자바 빈의 사용

1. 최상단 package 선언을 한다.

2. 실질적인 클래스의 구조를 작성한다.

3. 멤버변수를 작성한다.
   클래스 내부에서만 사용하므로 **private로 선언**하도록 한다.

4. 멤버 메서드 작성
   4-1. set~ 멤버 메서드
   "set + 이름" 형태로 구성되며, 멤버 변수에 값 저장.
   4-2. get~ 멤버 메서드
   "get + 이름" 형태로 구성되며, 멤버 변수의 값 추출.

자바 빈

JavaBean conventions

In order to function as a JavaBean class, an object class must obey certain conventions about method naming, construction, and behavior. These conventions make it possible to have tools that can use, reuse, replace, and connect JavaBeans.

The required conventions are:

The class must have a public default constructor. This allows easy instantiation within editing and activation frameworks.

The class properties must be accessible using *get*, *set*, and other methods (so-called accessor methods and mutator methods), following a standard naming convention. This allows easy automated inspection and updating of bean state within frameworks, many of which include custom editors for various types of properties.

The class should be serializable. This allows applications and frameworks to reliably save, store, and restore the bean's state in a fashion that is independent of the VM and platform.

예제 - PersonBean.java:

```
/**
* Class <code>PersonBean</code>.
*/

public class PersonBean implements java.io.Serializable {
private String name;
private boolean deceased;
/** No-arg constructor (takes no arguments). */
public PersonBean() {
}

/**
* Property <code>name</code> (note capitalization) readable/writable.
*/

public String getName() {
return this.name;
}
/**
* Setter for property <code>name</code>.
* @param name
*/

public void setName(final String name) {
this.name = name;
}
```

예제 - PersonBean.java:

```
/**
* Getter for property "deceased"
* Different syntax for a boolean field (is vs. get)
*/

public boolean isDeceased() {
return this.deceased;

}

/**
* Setter for property <code>deceased</code>.
* @param deceased
*/

public void setDeceased(final boolean deceased) {

this.deceased = deceased;

}

}
```

예제 - TestPersonBean.java

```
/**
* Class <code>TestPersonBean</code>.
*/

public class TestPersonBean {

/**
* Tester method <code>main</code> for class <code>PersonBean</code>.
* @param args
*/

public static void main(String[] args) {

PersonBean person = new PersonBean();

person.setName("Bob");

person.setDeceased(false);

// Output: "Bob [alive]"

System.out.print(person.getName());

System.out.println(person.isDeceased() ? " [deceased]" : " [alive]");

}

}
```

예제 - testPersonBean.jsp

```
<% // Use of PersonBean in a JSP. %>

<jsp:useBean id="person" class="PersonBean" scope="page"/>
<jsp:setProperty name="person" property="*"/>

<html>
<body>
Name: <jsp:getProperty name="person" property="name"/><br/>
Deceased? <jsp:getProperty name="person" property="deceased"/><br/>
<br/>
<form name="beanTest" method="POST" action="testPersonBean.jsp">
Enter a name: <input type="text" name="name" size="50"><br/>

Choose an option:

<select name="deceased">
<option value="false">Alive</option>
<option value="true">Dead</option>
</select>

<input type="submit" value="Test the Bean">
</form>

</body>
</html>
```

참고 : http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html

자바 빈

☒ DTO(Data Transfer Object: 데이터 전송 객체, Value Object) 빈즈
- 폼에서 입력된 데이터들은 하나의 DTO 객체로 변환 될 수 있습니다.
- 하나의 데이터베이스 레코드를 저장하며 레코드와 같은 구조를 가지고 있습니다.
- 하나의 레코드는 빈즈 클래스 객체 하나로 매핑됩니다.
- 데이터베이스 컬럼은 멤버 변수로 매핑됩니다.

1. 필드선언(멤버 변수, 인스턴스 변수)
    . 하나의 컬럼값을 저장
    . 보안성 및 캡슐화, 데이터 은닉의 목적으로 private으로 선언합니다.
    . private String name;

2. setter
    - 필드에 값을 저장하는 기능을 합니다.
    - HTML 폼의 INPUT태그의 값을 빈에 저장하는 역할을 합니다.
    - 메소드명은 set + input태그의 이름 중 첫자를 반드시 대문자로 사용하는 규칙을 적용하여 태그명을 지정합니다. 따라서 HTML에서 input 태그의 이름은 영문 소문자를 사용하며 태그의 이름에 신중을 기해야 합니다.

자바 빈

```
<input type="text" name="id" size="15" value='user1'>
      <input type="text" name="addr" size="15" value='user1'>
public void setId(String id) {
          this.id = id;
      }
public void setAddr(String addr) {
          this.addr = addr;
      }
```

- 모든 setter가 호출됩니다.

```
      <jsp:setProperty name="sungjukBean" property="*"  />
```

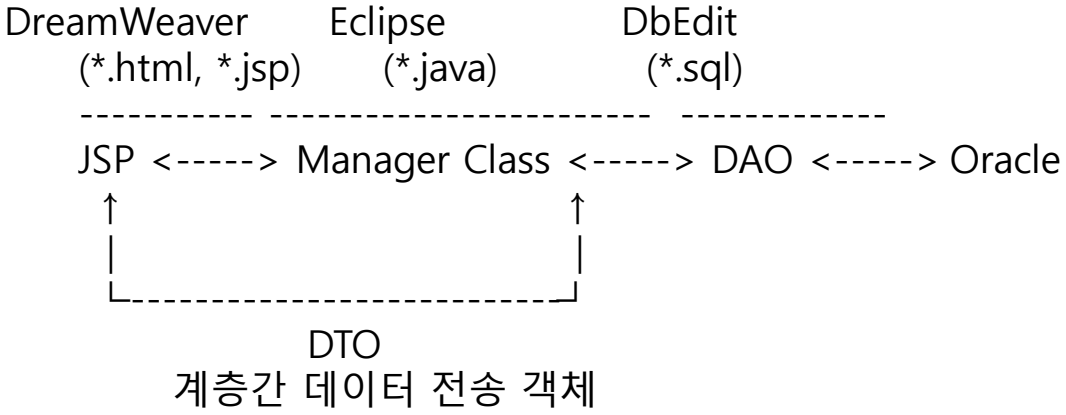
3. getter
   - 인스턴스 변수의 값을 가져오는 기능을 합니다.

```
public String getName(){
    return name;
}
```

자바 빈

☒ DAO(Data Access Object) 빈즈
- DTO 객체를 만들어 편집 및 조작을 합니다.
- DTO를 데이터베이스 서버에 저장하기도 하고 데이터베이스 서버로 부터 레코드를 SELECT 해
DTO 객체로 변경해 가져오기도 합니다.
- Insert, delete, update, select등 데이터 처리를 주 목적으로 합니다.

☒ Manager Class(관리 클래스)
- DTO와 DAO사이에서 연결 및 처리 역할을 합니다.



자바 빈

▩ 빈즈의 사용 Scope(범위)

- Page :
  . 기본값, page를 벗어나면 자동으로 소멸합니다.(중요)

- Request : forward, include에서 사용가능, 약간 사용됨

- Session :
  . 사용자가 로그인해 있는 동안 계속적으로 살아 있음
  . 메모리 소모가 심함으로 필요한 곳에 적절히 사용해야 함
  . 쇼핑카드 구현등 객체를 계속적으로 유지해야하는 경우에 사용됩니다.
  . 사용자가 브러우저를 닫으면 관련 JSP Session 빈은 소멸함(중요)

- Application
  . 웹 사이트 전체, 모든 사용자에게 영향을 미치는 빈
  . 메모리 소모가 심함으로 많이 사용하지 않음, 서버가 운영되는 동안 객체가 살아 있음
  . 모든 사용자가 변수와 객체를 공유하게 됨
  . 서버를 재시작해야 변수들이 재설정됨

IOC

IOC(Inversion of Control)의 의미

제어권의 역전(IOC): 객체의 생성에서부터 생명주기 관리까지 모든 객체에 대한 제어권이 바뀐 것을 의미

자바 기반의 어플리케이션 개발 초기에는 자바 객체를 생성하고 객체간의 의존관계를 연결시키는 등의 제어권이 개발자에게 있었다.
하지만 서블릿, EJB가 등장하면서 개발자들이 가지고 있던 제어권이 서블릿과 EJB를 관리하는 컨테이너에게 넘어가버리게 되었다.

IoC 컨테이너의 분류

DL(Dependency lookup)
- 저장소에 저장되어 있는 빈(Bean)에 접근하기 위하여 개발자들이 컨테이너에서 제공하는 API 를 이용하여 사용하고자 하는 빈(Bean) 을 Lookup 하는 것

DI (Dependency Injection)
- Dependency Injection은 Spring 프레임워크에서 지원하는 IoC의 형태이다.
- DI는 클래스 사이의 의존관계를 빈 설정 정보를 바탕으로 컨테이너가 자동적으로 연결해주는 것을 말한다.
- 개발자들은 빈 설정 파일(저장소 관리 파일)에 의존관계가 필요하다는 정보를 추가하면 된다.

Spring 프레임워크는 Setter Injection, Constructor Injection, Method Injection의 세가지 유형으로 나타난다.

의존성 관리

직접적인 의존성 관리

객체를 물리적으로 직접 생성하여, 의존성의 변경이 어려움.

```
public class BookController{

        ..

        public void rentBook(string isbn){

                BookDao dao = new BookJdbcDao();

                dao.rentBook();

        }

}
```

[편집] 팩토리 패턴/룩업을 사용한 의존성 관리

인터페이스를 통해서 구체 클래스를 감출수 있다.
단, 팩토리 클래스에 대한 의존성은 존재한다.

```
public class BookController{
```

의존성 관리

팩토리 패턴/룩업을 사용한 의존성 관리

인터페이스를 통해서 구체 클래스를 감출수 있다.
단, 팩토리 클래스에 대한 의존성은 존재한다.

```
public class BookController{

        ..

        public void rentBook(string isbn){

                BookDao dao = BookDaoFactory.getDao();

                dao.rentBook();

        }

}
```

의존성 관리

의존성 삽입(DI)을 통한 의존성 관리

의존성이 외부 조립자를 통해 관리되며, 인터페이스를 통해서 구체 클래스를 감춤으로서, 의존성을 언제라도 변경할 수 있다.
의존성을 삽입해주는 조립자만 있으면, 쉽게 확장하고 재사용할 수 있다.

```
public class BookController{

        private BookDao dao;



        public void rentBook(string isbn){

                dao.rentBook();

        }



        public void setDao(BookDao dao){..}

}
```

의존성 관리

스프링의 경량 컨테이너

스프링은 Ioc 기법을 지원하는 경량 컨테이너를 제공한다.
경량 컨테이너라는 것은 특정 플랫폼에 종속되지 않고, 쉽게 테스트 할 수 있음을 의미한다.
자바 코딩으로 객체를 초기화하고, 의존관계를 연결시키는 일들을 스프링 Ioc 컨테이너가 자동으로 해준다.
객체의 생성과 객체의 물리적인 위치를 어플리케이션 로직에서 분리하고
객체간의 의존관계(종속성)를 연결시키는 일들을 컨테이너가 하기 때문에 유연성과 확장성이 높아지게 됨.
엔터프라이즈 서비스를 플랫폼에 독립적으로 운용할 수 있는 바탕이 된다.

스프링 프레임워크를 통해 인스턴스가 생성되어 Inject(삽입)되어 서로 메세지를 주고 받게 된다.
이와 같은 설정은 XML파일(Application Context XML)에 정의된다.

```
<bean id="articleService" class="jcf.showcase.article.service.ArticleService" >

        <property name="articleDao" ref="articleDao"/>

        <property name="fileDao" ref="fileDao"/>

</bean>
```

의존성 관리

스프링의 컨테이너를 통한 빈들간의 협업

```
public class ArticleService {
        private ArticleDao articleDao;
        private IFileDao fileDao;

        public void setArticleDao(ArticleDao dao) {this.articleDao = dao;}
        public void setFileDao(IFileDao fileDao) {this.fileDao = fileDao;}

        public void writeArticle(Article article){
                articleDao.writeArticle(article);
                fileDao.writeFile(article.getAttach());
                ...
        }
}
```

의존성 관리

스프링 프레임워크를 통해 인스턴스가 생성되어 Inject(삽입)되어 서로 메세지를 주고 받게 된다.
이와 같은 설정은 XML파일(Application Context XML)에 정의된다.

```
<bean id="articleService" class="jcf.showcase.article.service.ArticleService" >

        <property name="articleDao" ref="articleDao"/>

        <property name="fileDao" ref="fileDao"/>

</bean>



<bean id="articleDao" class="jcf.showcase.article.dao.ArticleDaoImple" />

<bean id="fileDao" class="jcf.file.dao.FileDaoImple" />
```

위 설정파일은 비지니스 계층과 persistence 계층의 구현 클래스를 담당하고있는 클래스(ArticleDao 와 FileDaoImple)에 대한 생성과 의존관계를 관리한다.
<bean/>엘리먼트를 이용하여 하나의 빈의 생성하고 이렇게 생성된 빈은 setter 메소드를 이용해서 다른 빈과 의존관계 형성이 가능하다.
위의 코드는 ArticleService클래스의 setFileDao() 메소드를 통해 FileDaoImple 인스턴스를 전달하게 된다.

의존성 관리

스프링의 다양한 IoC 방법

생성자를 통한 의존성 관리

```
public class BookController{
        private BookDao Dao;
        public BookController(BookDao dao){
                this.dao = dao;
        }
        public void rentBook(Book book){
                dao.rent(book);
            ..
        }
}
```

```
<bean id="bookController" class="store.ctl.BookController">
        <constructor-arg>
                <ref bean="bookDao">
        </constructor-arg>
</bean>

<bean id="bookDao" class="store.dao.BookJdbcDao" />
```

의존성 관리

생성자를 통한 변수값 설정 방법

```
<bean id="timeController" class="store.ctl.timeController">
        <constructor-arg value="1000" />
</bean>
```

리스트 타입의 값 설정 방법

```
public class PerController{
        private List pets;
        public void setPets(List pets){this.pets = pets}
}

<bean id="petController" class="store.ctl.PetController">
        <property name="pets">
                <list>
                        <value>cat</value>
                        <value>dog</value>
                </list>
        </property>
</bean>
```