

# 시스템 (도시를 세운다면? ~ 자바 프록시)

## 도시를 세운다면?



<https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=ntscafe&logNo=110111135470> (국세청 공식 블로그)

깨끗한 코드를 구현하면 낮은 추상화 수준에서 **관심사**를 분리하기 쉬워진다.

### 추상화란?

공통되는 부분을 일반화하거나, 간추려 내는 것

### 관심사란?

기능이나 모듈

The Art of Separation of Concerns · Aspiring Craftsman

🌐 <https://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/>

# 시스템 제작과 시스템 사용을 분리하라

## 시스템 제작 ≠ 시스템 사용

소프트웨어 시스템은

- 애플리케이션 객체를 제작하고 의존성을 서로 연결하는 **준비 과정**
- 준비 과정 이후에 이어지는 **런타임 로직**

을 분리해야 한다

### 관심사 분리가 중요하다

#### Bad Example

```
public Service getService() {  
    if(service == null)  
        service = new MyServiceImpl(...); // 모든 상황에 적합한 기본값일까?  
    return service;  
}
```


위의 코드는 필드(변수)의 초기화 시점을 그 값이 처음 필요할 때까지 늦추는 기법이며, 필요할 때 생성한다고 보면 된다

- 장점
  - 애플리케이션의 시작 시간이 그만큼 빨라진다!
    - 실제로 필요할 때까지 객체를 생성하지 않으므로 불필요한 부하가 걸리지 않으니까
  - 어떤 경우에도 null을 반환하지 않는다
- 단점
  - **getService()** 메서드가 **MyServiceImpl** 과 **생성자 인수** 에 명시적으로 의존 (너 없으면 안 돼~)
  - 테스트도 문제다
    - 만약 **MyServiceImpl** 가 무거운 객체라면?
      - 테스트 전용 객체가 또 필요 → 이거를 또 service 필드에 할당
      - 일반 런타임 로직에 객체 생성 로직을 섞어서 모든 실행 경로를 테스트 해야 한다
  - 이렇게 되면 **메서드가 작업을 두 가지 이상 수행하는 꼴이 된다**
    - 단일 책임 원칙(SRP, Single Responsibility Principle)를 깰 여지가 충분

- MyServiceImpl이 모든 상황에 적합한 객체일까?
- getService()를 포함한 클래스가 전체 문맥을 알 필요가 있을까?
- 이 시점에서 어떤 객체를 사용할지 알 수 있을까?
- 한 유형이 모든 문맥에 적합할 가능성이 있을까?

지연 초기화는 신중히 사용하자

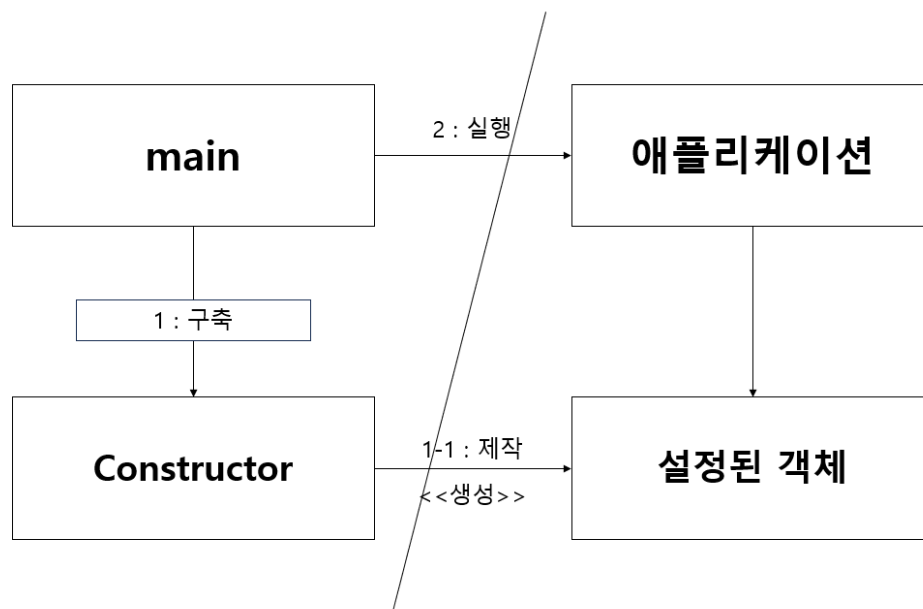
지연초기화

 <https://donghyeon.dev/이펙티브자바/2021/10/22/지연-초기화는-신중히-사용하자/>

## 시스템 생성과 시스템 사용을 분리하는 경우

### Main 분리

- **생성** 과 관련된 것들 - `main` 이나 `main`이 호출하는 모듈 로 옮기고
- 나머지 시스템은 모든 객체가 생성되었고, 모든 의존성이 연결되었다고 가정



main에서 생성을 분리

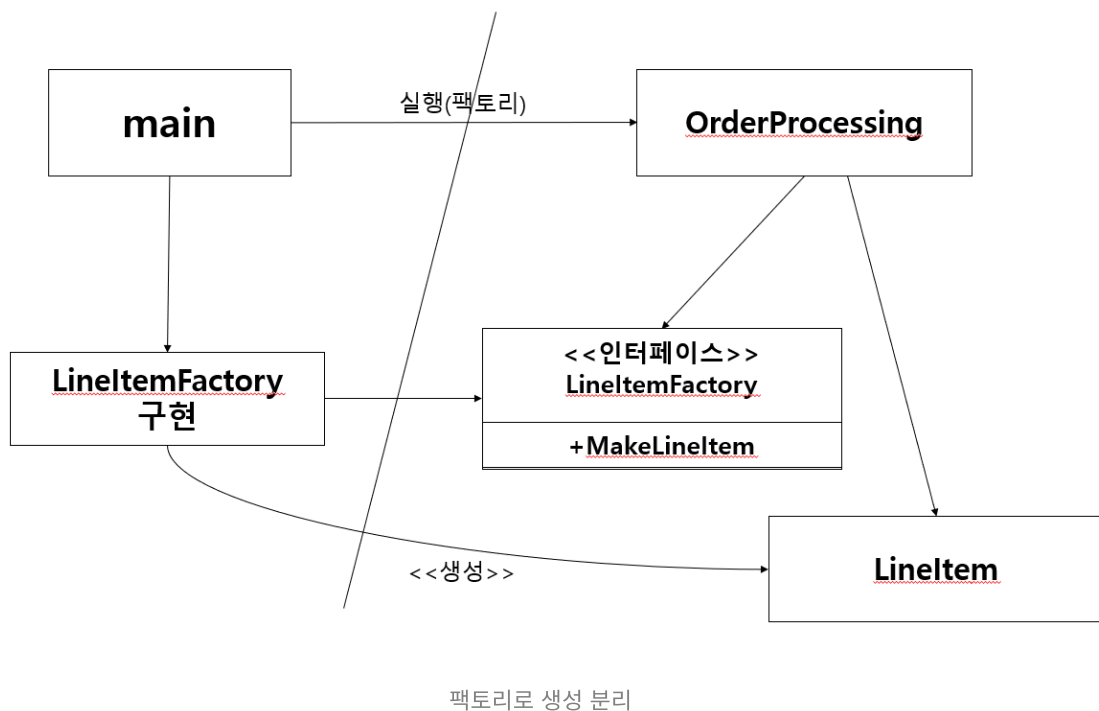
- `main` 함수에서 객체 생성 후 애플리케이션에 넘김

- 객체를 넘겨받은 애플리케이션은 **객체의 생성과정을 모른다**. 그저 객체를 사용할 뿐이다.
  - 적절히 생성되었겠거니~

## 팩토리

때로는 애플리케이션이 **객체가 생성되는 시점을 결정할 필요**도 생김  
 단, **객체의 생성 시점만 결정하지** 코드는 여전히 몰라도 된다

Example : 주문 처리 시스템



- OrderProcessing 애플리케이션은 **팩토리 패턴**을 이용하여 LinItem 인스턴스를 생성(`new LinItemFactory`)해서 Order에 추가
- 모든 의존성이 main에서 OrderProcessing 애플리케이션으로 함함
  - 이는 OrderProcessing 애플리케이션이 LinItem이 생성되는 구체적인 방법을 **여전히 모른다**는 것
  - 하지만 생성되는 시점은 완벽하게 통제
  - 필요 시 OrderProcessing 애플리케이션에서만 사용하는 생성자 인수도 넘길 수 있다

## 의존성 주입

## 의존성 주입

### 제어의 역전을 의존성 관리에 적용한 매커니즘

#### 제어의 역전

- 한 객체가 맡은 보조 책임을 새로운 객체에 떠넘긴다
- 새로운 객체는 **넘겨받은 책임만 맡으므로** 단일 책임 원칙을 지키게 된다
- 떠넘긴 한 객체는 의존성 관리 맥락에서 **의존성 자체를 인스턴스로 만드는 책임은 굳이 지지 않는다**
  - 대신에 이런 책임을 **전담** 매커니즘에 넘겨야 한다
  - **초기 설정**은 시스템 전체에서 필요하므로 **책임질** 매커니즘인 **main** 루틴이나 **특수 컨테이너**를 사용
- 예) JNDI Lookup : 객체는 디렉터리 서버에 이름을 제공하고, 그 이름과 일치하는 서비스를 요청

```
MyService myService = (MyService) (jndiContext.lookup("NameOfMyService"));
```

- 호출하는 객체인 **jndiContext**는 실제로 반환되는 객체의 유형인 **MyService**를 제어하지 않지만, 의존성을 능동적으로 해결

#### 수동적인 클래스는 의존성을 해결하려 하지 않는다

- 대신 **setter 메서드**나 **생성자 인수**를 제공하는 방법으로 의존성을 주입할 수 있다
- DI 컨테이너는 **요청이 들어올 때마다** 필요한 객체의 인스턴스를 만든 후 **setter 메서드**나 **생성자 인수**를 사용해 의존성을 설정한다
- 예) Spring Framework
  - 스프링은 DI Container를 제공하며, 객체 사이의 의존성은 XML 파일에 정의
  - Java에서는 이름으로 특정 객체를 요청


#### 초기화 지연/계산 지연으로 얻는 장점은 포기해야 할까?

- DI를 활용해도 이 지연 방법은 여전히 유용하다
  - 왜? 대다수 **DI Container**는 필요할 때까지는 객체를 생성하지 않고, 계산 지연이나 비슷한 최적화에 쓸 수 있도록 **팩토리를 호출하거나 프록시를 생성하는 방법을 제공한다**

DI를 쉽게 이해하기 위한 예시

## Spring (1) - DI

Spring을 학습한다고 하면 필수적으로 숙지해야 할 개념인 DI에 대해 알아봤습니다.

 <https://vlog.io/@brucehan/Spring-DI에-대하여>

## Spring DI에 대하여

Spring 튜토리얼 1

위 블로그에서는 **바리스타** 예시가 나온다

바리스타가 **커피 메뉴얼**의 변경이 필요할 때  
**본인이 직접 바꾸지 않고 본사에서 메뉴얼을 변경해** 공지한다

실제로도 그렇습니다 예) 뚜레쥬르, 스타벅스, 투썸 플레이스 등

## 확장

예전엔 맞았고, 지금은 틀리다  
매번(한 번에) 옳은 결정으로 인생을 완벽하게 꾸밀 수는 없다. 바뀌어 나가는 것이다.

군락이었던 곳이 → 마을 → 도시로 발전함에 따라 도로를 확장해야 한다

발전된 도시에서 도로를 뚫자니 좀 많이 힘들다  
그렇다고 군락에서 처음부터 그렇게 많이 뚫어도 될까?

**처음부터 완벽하게, 올바르게** 만들 수 있다는 것은 힘들다

새로운 요구사항에 맞춰 **그때그때 조정, 확장** → 애자일 방식의 핵심  
TDD, 리팩토링으로 얻어지는 깨끗한 코드로 시스템을 확장할 수 있다고 믿는다

단, **아키텍처**는 그때마다 조정, 확장이 어렵다  
그래서 **관심사를 적절히 분리**해야 한다

## Bad Example : EJB1, EJB2

- **EJB1** 과 **EJB2** 아키텍처는 관심사를 적절히 분리하지 못함

```
package clean.coder.banking;

import java.util.Collections;
import javax.ejb.*;

/**
 * EJB1 코드
 * 책에서는 EJBLocalObject가 java.ejb.EJBLocalObject로 표현되어 있지만
 * java.ejb 패키지와 관련된 공식문서가 없으므로 javax.ejb 패키지에 있다고 가정함
 */
public interface BankLocal extends EJBLocalObject {
    String getStreetAddr1() throws EJBException;
    String getStreetAddr2() throws EJBException;
    String getCity() throws EJBException;
    String getState() throws EJBException;
    String getZipCode() throws EJBException;

    void setStreetAddr1(String street1) throws EJBException;
    void setStreetAddr2(String street2) throws EJBException;
    void setCity(String city) throws EJBException;
    void setState(String state) throws EJBException;
    void setZipCode(String zip) throws EJBException;

    Collections getAccounts() throws EJBException;
    void setAccounts(Collection accounts) throws EJBException;
    void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

- 영속적으로 저장될 Bank 클래스에 필요한 엔티티 빈은 테이블 행을 표현하는 객체이자 관계형 자료로 메모리에 상주한다
- 클라이언트가 사용할 **프로세스 내의 지역 인터페이스** 나 **다른 JVM에 있는 원격 인터페이스** 를 정의해야 한다
- 위의 예시는 지역 인터페이스를 나타냈는데
  - 여기서 열거하는 속성은 Bank 주소, 은행이 소유하는 계좌다
  - 각 정보는 Account EJB로 처리한다

```
package clean.coder.banking;

import java.util.Collections;
import javax.ejb.*;

// EJB2 코드
public abstract class Bank implements EntityBean {

    // 비즈니스 논리(?)
    public abstract String getStreetAddr1();
    public abstract String getStreetAddr2();
    public abstract String getCity();
    public abstract String getState();
    public abstract String getZipCode();

    public abstract void setStreetAddr1(String street1);
    public abstract void setStreetAddr2(String street2);
    public abstract void setCity(String city);
    public abstract void setState(String state);
    public abstract void setZipCode(String zip);
}
```

```

public abstract Collections getAccounts();
public abstract void setAccounts(Collections accounts);
public void addAccount(AccountDTO accountDTO) {
    InitialContext context = new InitialContext();
    AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
    AccountLocal account = accountHome.create(accountDTO);
    Collections accounts = getAccounts();
    accounts.add(account);
}

// EJB 컨테이너 논리
public abstract void setId(Integer id);
public abstract Integer getId();
public Integer ejbCreate(Integer id) { ... }
public void ejbPostCreate(Integer id) { ... }
// 나머지도 구현해야 하지만 일반적으로 비어있다
public void setEntityContext(EntityContext ctx) {}
public void unsetEntityContext() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbLoad() {}
public void ejbStore() {}
public void ejbRemove() {}
}

```


- 두 번째 예시는 엔티티 빈에 대한 구현 클래스를 나타낸다
- 마지막으로, 영구적인 저장소에다가
  - 객체와 관계형 자료가 매핑되는 방식
  - 원하는 트랜잭션 방식
  - 보안 제약조건 등이 들어가는 XML 배포 기술자를 작성해야 한다
- 엔티티 빈에 대한 구현 클래스 예제에서 비즈니스 논리는 **EJB2 애플리케이션 컨테이너**에 강하게 결합된다
  - 클래스를 생성할 때는 컨테이너에서 파생해야 하며
  - 컨테이너가 요구하는 다양한 생명주기 메서드도 제공해야 한다
- 비즈니스 논리가 **덩치 큰 컨테이너와 밀접하게 결합된** 탓에 단위 테스트가 어렵다
  - 컨테이너를 흉내내거나
  - EJB와 테스트를 실제 서버에 배치해야 한다(...)
- 이러한 EJB2 코드는 프레임워크 밖에서 재사용할 수 없다(~~차라리 다른 프레임워크를 쓰자~~)
  - 이렇게 되면 **빈은 다른 빈을 상속받지 못하므로**, 객체 지향적이지 못하게 될 것이다
- EJB2 빈이 DTO를 정의한다고 했을 때, DTO에는 메서드가 없어 거의 구조체다
  - 동일한 정보를 저장하는 자료 유형이 두 개가 되며
  - 이는 한 객체에서 다른 객체로 자료를 복사하는 반복적인 규격 코드가 필요하다

EJBLocalObject (Java EE 6 )

 <https://docs.oracle.com/javaee/6/api/javax/ejb/EJBLocalObject.html>



EJBLocalObject (Java EE 6 )

 <https://docs.oracle.com/javaee/6/api/javax/ejb/EJBLocalObject.html>

## 횡단 관심사

영속성과 **관심사**는 애플리케이션의 자연스러운 객체 경계를 넘나드는 경향이 있다  
모든 객체가 전반적으로 동일한 방식을 이용하게 만들어야 한다

예를 들어

- 특정 DBMS나 독자적인 파일
- 테이블과 열은 같은 명명관례를 따르거나
- 트랜잭션 의미가 일관적이면 좋다

원론적으로는 모듈화되고 캡슐화된 방식으로 영속성 방식을(오래 계속되도록) 구상할 수 있다

→ 그러나, 현실적으로는 온갖 객체로 흩어져서 쉽지 않다

여기에서 **횡단 관심사** 라는 용어가 나온다

### 횡단 관심사는

핵심적인 기능이 아닌 중간에 삽입되어야 할 기능/관심

AOP(Aspect-Oriented Programming)

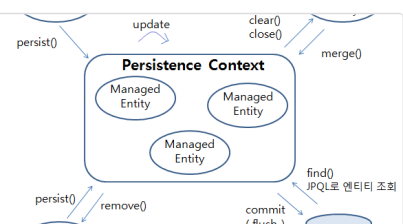
모듈 구성함에 있어 **특정 관심사를 지원하려면** 시스템에서 특정 지점들이 동작하는 방식을 **일관성 있게 바꿔야 한다**

예) 영속성(데이터를 생성한 프로그램의 실행이 종료되더라도 사라지지 않는 데이터의 특성)

#### 영속성(Persistence)이란?

업무를 하다보면 동료가 영속성이란 말을 하는걸 들을 수 있습니다. 혹은 IT서적을 보다 보면 영속성(Persistence)라는 단어가 종종 나오는것을 볼 수 있는데요. 과연 영속성(Persistence)이라는 것이 무엇인지 알아보겠습니다. 알고보면 별거 아닙니다. 아주 쉽

 <https://sugerent.tistory.com/587>



프로그래머 → 영속적으로 저장할 객체와 속성 선언 → 그 영속성 책임을 영속성 프레임워크(JPA, MyBatis, JdbcTemplate)에 위임

→ AOP 프레임워크는 **대상 코드에 영향을 미치지 않는 상태로** 동작 방식 변경

## 자바 프록시

프록시란?

대리자라는 의미이다. 대리자라고 하면 다른 누군가를 대신해 그 역할을 수행하는 존재를 말한다.

[스프링 입문을 위한 자바 객체 지향의 원리와 이해] 6장 : 스프링이 사랑한 디자인 패턴

'객체지향의 4대 특성'은 객체지향을 잘 사용하기 위한 '도구'이다. '객체지향의 5대 원칙'은 이러한 도구를 올바르게 사용하는 원칙으로 볼 수 있다. 그렇다면 디자인패턴은 무엇에 비유할 수 있을까? '디자인 패턴'은 레시피에 비유할 수 있다. 실제 개발 현장에서 비즈니스 요구 사항을 처리하면서

 <https://yunanp.tistory.com/47>



단순한 상황에 적합하며, 개별 객체나 클래스에서 메서드 호출을 감싸는 경우가 좋은 예다

```
package clean.coder;

import java.util.*;

public interface Bank {
    Collection<Account> getAccounts();
    void setAccounts(Collection<Account> accounts);
}
```

추상화를 위한 POJO(Plain Old Java Object) 구현

```
package clean.coder;

import java.util.*;

public class BankImpl implements Bank {
    private List<Account> accounts;

    public Collection<Account> getAccounts() {
        return accounts;
    }

    public void setAccounts(Collection<Account> accounts) {
```

```


        this.accounts = new ArrayList<Account>();
        for (Account account : accounts) {
            this.accounts.add(account);
        }
    }
}

```

reflect : 클래스의 구조를 개발자가 확인할 수 있고, 값을 가져오거나 메소드를 호출

### 자바 Reflection이란?

많은 입문용 자바 서적에서 잘 다루지 않는 Reflection이라는 개념에 대해서 알아보려고 합니다.

 <https://medium.com/msolo021015/자바-reflection이란-ee71caf7eec5>

```

Class c = Class.forName("io.mong.reflect.DemoReflection");
Method[] m = c.getDeclaredMethods();
for(int i=0; i<m.length; i++){
    Method m1 = m[i];
    System.out.println("name: " + m1.getName());
    System.out.println("declare Class: " + m1.getDeclaringClass());

    Class[] gpt = m1.getParameterTypes();

    for(int j=0; j<gpt.length; j++){
        System.out.println("Param: " + gpt[j]);
    }

    Class[] ept = m1.getExceptionTypes();

    try{
        m1.invoke(c.newInstance(), gpt);
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
    }
}

```

```

package clean.coder;

import java.lang.reflect.*;
import java.util.*;

public class BankProxyHandler implements InvocationHandler {
    private Bank bank;

    public BankProxyHandler(Bank bank) {
        this.bank = bank;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        String methodName = method.getName();
        if(methodName.equals("getAccounts")) {
            bank.setAccounts(getAccountsFromDatabase());
            return bank.getAccounts();
        } else if(methodName.equals("setAccounts")) {
            bank.setAccounts((Collection<Account>) args[0]);
            setAccountsToDatabase(bank.getAccounts());
            return null;
        } else {
            ...
        }
    }
}

protected Collection<Account> getAccountsFromDatabase() { ... }
protected void setAccountsToDatabase(Collection<Account> accounts) { ... }
}

```

```

Bank bank = (Bank) Proxy.newProxyInstance(
    Bank.class.getClassLoader(),
    new Class[] { Bank.class },
    new BankProxyHandler(new BankImpl())
);

```

Bank 메서드를 구현하는데 사용하고자 InvocationHandler를 Proxy API에 넘긴다

BankProxyHandler는 Java Reflection API를 사용해 제네릭 메서드에 해당되는 BankImpl 메서드로 감싼다

---

단순한 예제이지만, 코드가 상당히 많으며 복잡하다

→ **코드 양과 크기**는 프록시의 단점

- **프록시를 사용하면 깨끗한 코드를 작성하기 어렵다**
- 시스템 단위로 **실행 지점** 을 명시하는 매커니즘도 제공하지 않는다

AOP 시스템의 진정한 가치는 시스템 동작을 간결하고 모듈화된 방식으로 명시하는 능력