Chapter 11

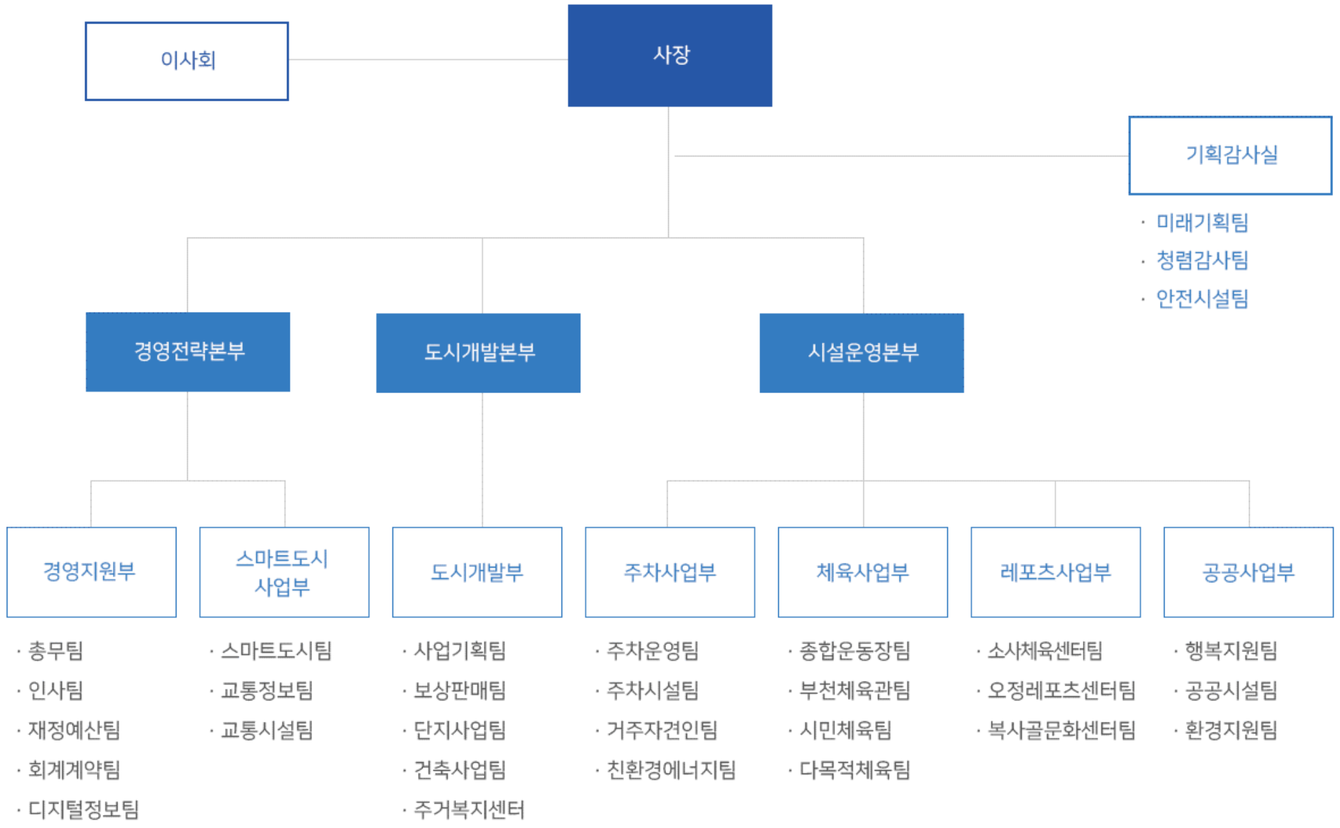
관심사

분리

관심사 분리의 방법과 그 이점

코드를 도시처럼

큰 그림을 그리는 사람과
작은 사항에 집중하는 사람은 분리되어야 한다.



ref. 나의 고향...부천도시공사 조직도

낮은 추상화 수준에서 깨끗한 코드는 관심사 분리를 쉽게 해준다.
높은 추상화 수준인 시스템을 깨끗하게 유지하려면?

우선 제작과 사용을 분리해라!

생성과 사용이 분리되지 않은 경우

```java
public Service getService() {
    if (service == null)
        service = new MyServiceImpl();
    return service;
}
```

초기화 지연 기법 (Lazy Initialization)
생성과 사용이 함께 동작한다.

장점

1. 실제로 필요할 때 까지 객체 생성 X
   -> 불필요한 부하 X
2. 어떤 경우에도 null 반환 X

단점

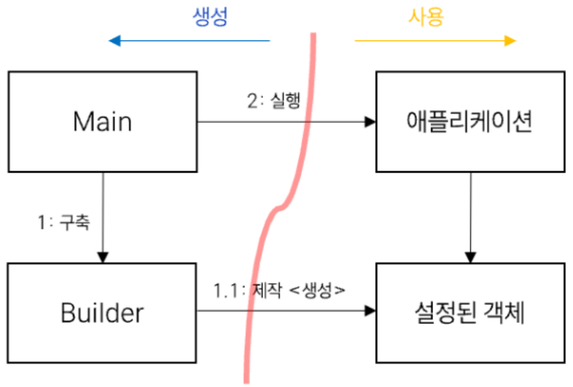
1. getService 메서드가 MyServiceImpl과 생성자 인수에 명시적으로 의존
2. 일반 런타임 로직과 생성 로직이 섞임
   -> 2개의 책임
   -> 단일 책임 원칙 위반
3. MyServiceImpl 객체가 모든 상황에 적합한 지 판단 불가

손 쉬운 기법을 남용하면
체계적이고 탄탄한 시스템은
깨지게 된다!

생성과 사용의 분리 방법 3가지

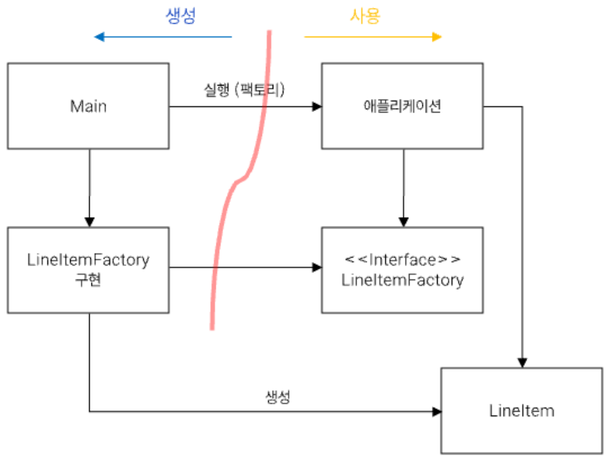
1

Main 분리



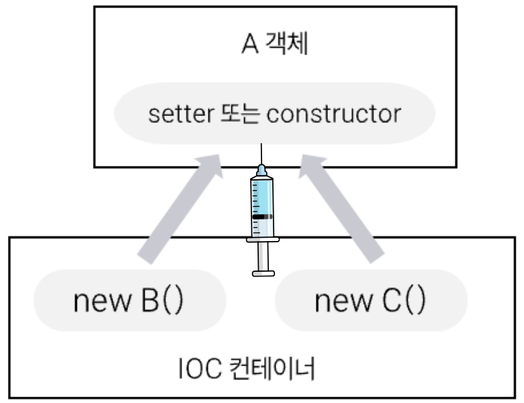
2

팩토리



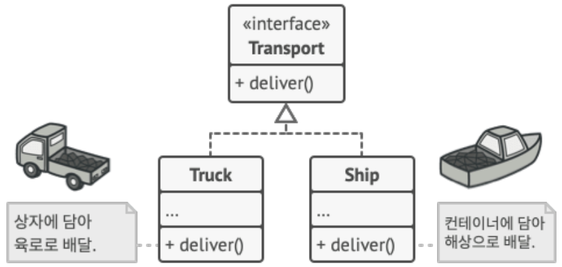
3

의존성 주입



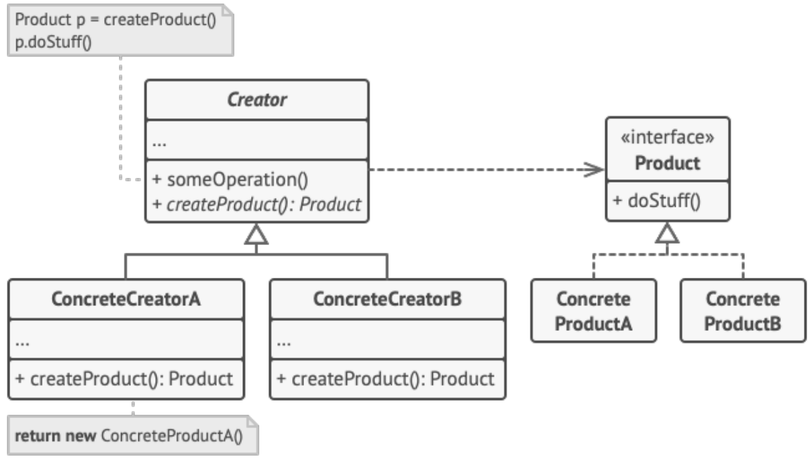
팩토리 메서드 패턴과 추상 팩토리 패턴..?!

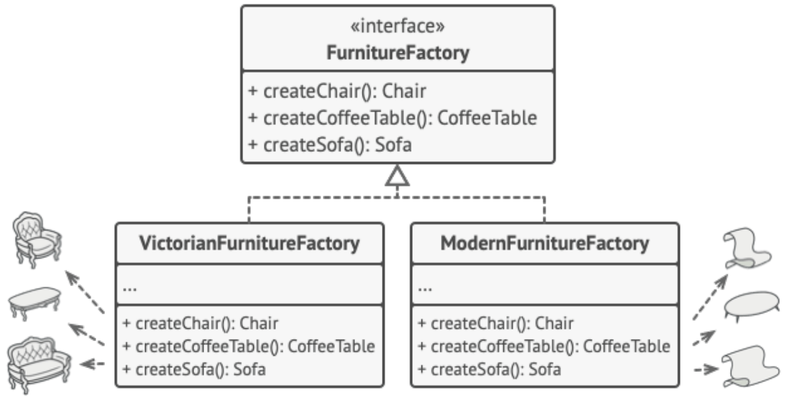
팩토리 메서드 패턴 vs 추상 팩토리 패턴



팩토리 메서드

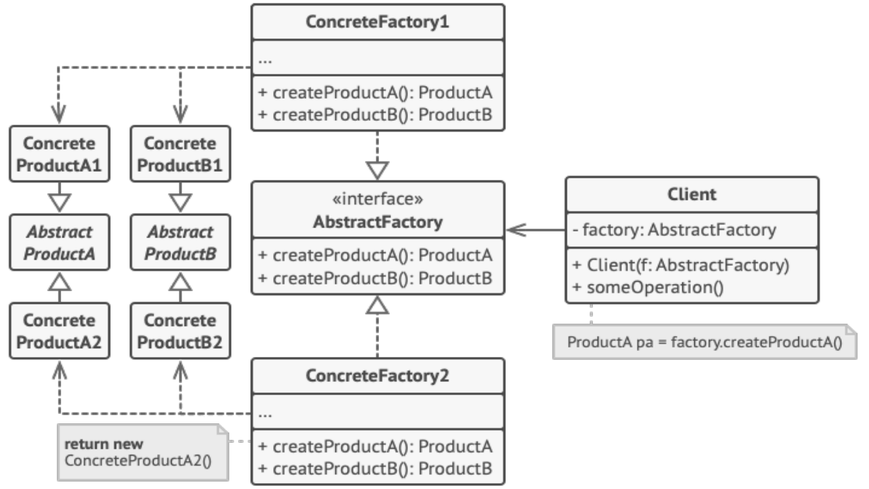
객체 생성을 팩토리 클래스로 위임
-> 한 팩토리가 한 종류의 객체 생성





추상 팩토리

서로 관련이 있는 객체들(제품군)을 묶어서 여러 종류의 객체를 생성하기 위해 사용

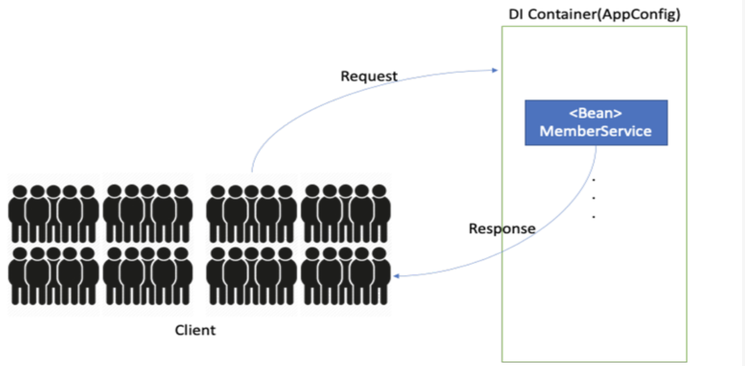


ref. https://refactoring.guru/ko

싱글톤 패턴과 의존성 주입

싱글톤 패턴을 적용하지 않는다면?

수많은 사용자들로 인해 발생하는 요청
↓
멀티스레드로 생성된 수많은 스레드가
처리하는 과정마다 필요한 객체를 생성
↓
시스템적 성능 저하 및 굉장한 메모리 낭비가 발생



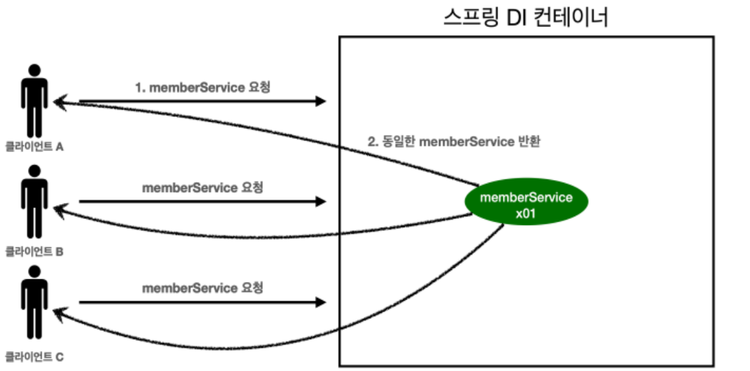
ref. https://catsbi.oopy.io/6c4846a1-130d-4aba-94ea-e630cc15056d

하지만 싱글톤 패턴의 단점은??

스프링 컨테이너로 해결
개발자가 싱글톤 패턴을 적용하지 않아도 객체 인스턴스를 싱글톤으로 관리한다.
싱글톤 패턴의 단점은 없애면서 장점만 얻을 수 있다.

의존성 주입 (DI)

제어 역전(Inversion of Control, IoC) 기법을 의존성 관리에 적용
↓
객체는 의존성 자체를 인스턴스로 만드는 책임을 지지 않고
이러한 책임을 전담하는 매커니즘에 넘김으로써 제어를 역전
↓
중간에 의존성 주입자(컨테이너)가 간접적으로 의존성을 주입하는
방식을 통해 모듈간의 결합을 느슨하게 함 (낮은 결합도)



ref. 김영한님 스프링 - 기본편 강의 자료

횡단 관심사

관심사가 적절히 분리되지 못한 예

```java
public interface BankLocal extends java.ejb.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;
    Collection getAccounts() throws EJBException;
    void setAccounts(Collection accounts) throws EJBException;
    void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

인터페이스

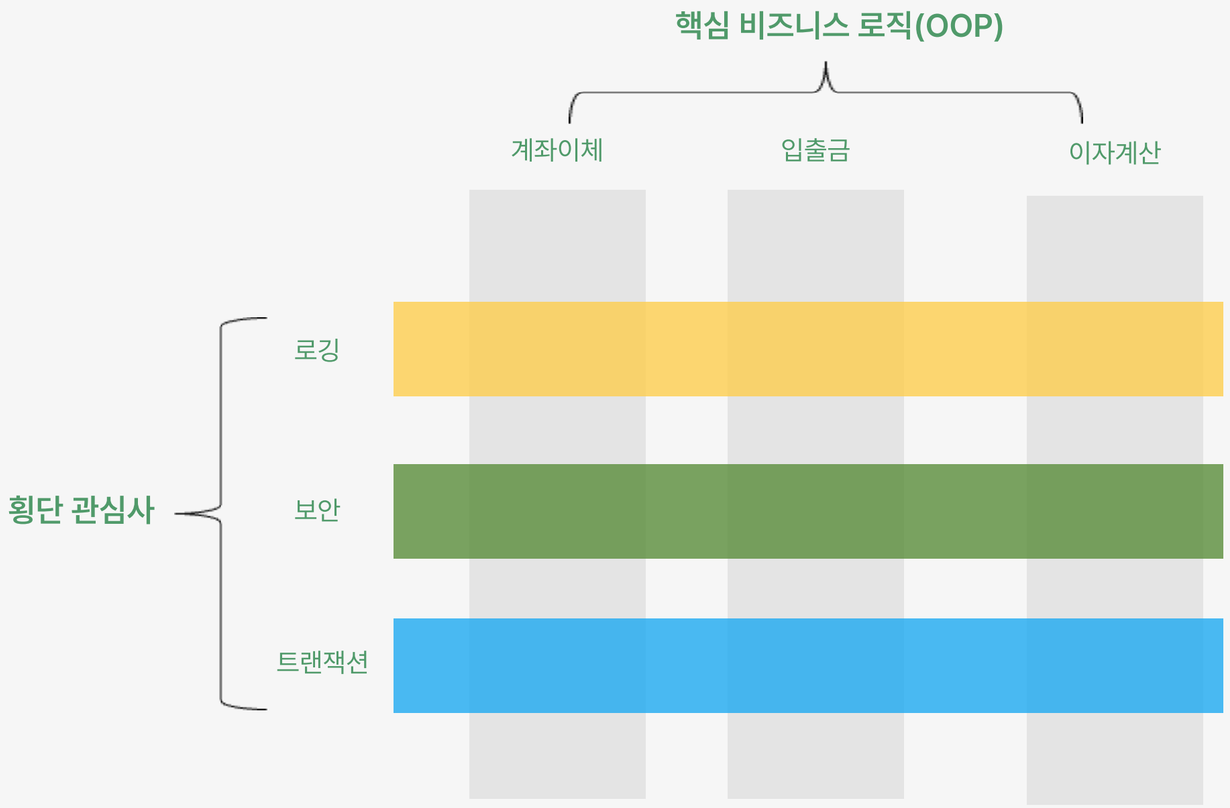
```java
public abstract class Bank implements javax.ejb.EntityBean {
    // 비즈니스 로직
    public abstract String getStreetAddr1();
    public abstract String getStreetAddr2();
    public abstract String getCity();
    public abstract String getState();
    public abstract String getZipCode();
    public 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 Collection getAccounts();
    public abstract void setAccounts(Collection accounts);
    public void addAccount(AccountDTO accountDTO) {
        InitialContext contet = new InitialContext();
        AccountHomeLocal accountHome = context.lookup("AcccountHomeLocal");
        AccountLocal account = accountHome.create(accountDTO);
        Collection accounts = getAccounts();
        accounts.add(account);
    }
    // EJB 컨테이너 로직
    public abstract void setId(Integer id);
    public abstract Integer getId();
    public Integer ejbCreate(Integer id) { ... }
    public void ejbPostCreate(Integer id) { ... }
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}
}
```

구현 클래스

덩치 큰 EJB 컨테이너와
강하게 결합되어 테스트하기
힘들고,
컨테이너가 요구하는
사항들을 충족해야 한다.

횡단 관심사

관심사를 적절히 분리해 관리하면 소프트웨어 아키텍처는 점진적으로 발전할 수 있다.



AOP

관점 지향 프로그래밍으로 OOP로 개발된 모듈에
중복으로 들어가는 트랜잭션, 로깅, 보안 코드를 횡단 관심사
로 분류해서 관점(aspect)으로 관리하는 방법

EJB 아키텍처가 이미 트랜잭션, 보안 등 일부 영역에서
관심사를 완벽하게 분리
→ 관점 지향 프로그래밍 (AOP) 예견

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

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);
        }
    }
}

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 (mehtodName.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()));
```

관점 분리 메커니즘 - 자바 프록시

단순한 상황에 적합하다.
개별 객체나 클래스에서 메서드 호출을 감싸는 경우가 좋은 예다.

But,

1. 코드의 양이 상당히 증가 → 깨끗한 코드 작성이 어려움
2. 시스템 단위로 '실행 지점'을 명시하는 메커니즘 제공 X

관점 분리 메커니즘
- 순수 자바 AOP 프레임워크

```xml
<beans>
    ...
    <bean id="appDataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="com.mysql.jdbc.Driver"
        p:url="jdbc:mysql://localhost:3306/mydb"
        p:username="me"/>

    <bean id="bankDataAccessObject"
        class="com.example.banking.persistence.BankDataAccessObject"
        p:dataSource-ref="appDataSource"/>

    <bean id="bank"
        class="com.example.banking.model.Bank"
        p:dataAccessObject-ref="bankDataAccessObject"/>
    ...
</beans>
```

```java
XmlBeanFactory bf = new XmlBeanFactory(new ClassPathResource("app.xml", getClass()));
Bank bank = (Bank) bf.getBean("bank");
```

순수 자바 AOP= AOP 언어인 AspectJ를 사용하지 않는 방법
순수 자바 AOP 프레임워크에는 Spring AOP, JBoss AOP 등이 있다.
위 프레임워크에서는 내부적으로 프록시를 사용한다.

스프링 AOP 프레임워크는 빈 생성 정보만 xml에 정의해놓으면
DI 컨테이너가 생성과 주입을 전부 처리해준다.

↓

1. EJB2 시스템이 가지던 강한 결합 문제를 모두 해결
2. 프록시나 관점 논리보다 단순

POJO와 Spring의 등장

스프링 삼각형

EJB 기술에 지나친 종속

-> 마틴 파울러의 Plain Old Java Object 발표
(환경과 기술에 종속되지 않는 순수한 자바 객체)

기술적 복잡함을 핵심 로직에서 제거

"스프링의 정수(essence)는 엔터프라이즈 서비스 기능을 POJO에 제공하는 것"
= 엔터프라이즈 서비스 기술과 POJO 애플리케이션 로직을 담은 코드를 분리

"분리됐지만 반드시 필요한 엔터프라이즈 서비스 기술을 POJO 방식으로 개발된 애플리케이션 핵심 로직을 담은 코드에 제공한다"는 것이 **스프링의 가장 강력한 특징과 목표**

```java
@Entity
@Table(name = "BANKS")
public class Bank implements Serializable {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    @Embeddable
    public class Address {
        protected String streetAddr1;
        protected String streetAddr2;
        protected String city;
        protected String state;
        protected String zipCode;
    }

    @Embedded
    private Address address;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy = "bank")
    private Collection<Account> accounts = new ArrayList<Account>();

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void addAccount(Account account) {
        account.setBank(this);
        accounts.add(account);
    }

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

    public void setAccounts(Collection<Account> accounts) {
        this.accounts = accounts;
    }
}
```

EJB 개선

매력적인 아키텍쳐의 스프링 프레임워크를 보고
EJB3을 같이 완전히 뜯어고치게 된다.

↓

1. 다른 라이브러리나 프레임워크에 종속되지 않은
POJO 객체만 남는다.

2. 테스트를 하기 쉬워졌다.

결론

깨끗한 시스템 구조 = POJO + 관심사 분리

Why?

01 분리가 잘 되어있다면, 확장이 쉬워진다.

02 지엽적인 관리와 결정이 가능해진다. 가장 적합한 책임자(모듈화된 컴포넌트)에게 전달할 수 있다.

03 테스트 주도 아키텍처 구축이 가능해진다.

⋮