EJB 기반 애플리케이션에서 데이터베이스의 효율적 액세스를 위한 IDAO의 설계 및 구현

최 성 만[†]·이 정 열[††]·유 철 중[†††]·장 옥 배[††††]

요 약

컴포넌트 기반 애플리케이션의 개발 및 배치를 위한 명세를 제공하는 EJB(Enterprise JavaBeans)는 엔터프라이즈 환경에서 가장 복잡한 트랜잭션 관리, 퍼시스턴스, 동시성 제어 등을 자동적으로 관리해주는 J2EE 환경의 핵심으로서 분산개발을 가능하게 한다. 본 논문에서는 EJB 기반 레거시 시스템에서 DAO의 트랜잭션 로직 복잡성과 시스템의 성능저하 문제를 해결하고자 한다. 따라서, 본 논문에서는 Iterator 패턴을 적용한 IDAO를 설계 및 구현하였다. IDAO는 컨테이너 관리 트랜잭션을 통해 데이터베이스 커넥션에 따른 트랜잭션 로직의 복잡성과 시스템의 과부하 감소 및 시스템의 성능저하를 감소시키는 효과를 얻었다.

Design and Implementation of IDAO for Efficient Access of Database in EJB Based Application

Seong-Man Choi[†] · Jeong-Yeal Lee[††] · Cheol-Jung Yoo[†††] · Ok-Bae Chang[††††]

ABSTRACT

EJB, providing specification for development and deployment of component based application, permits distributed development as central element of J2EE environment that manages automatically transaction management, persistence, concurrency control that are the most complicated work in enterprise environment. In this paper, we wish to resolve DAO's transaction logic complexity and performance reduction of system in the EJB based legacy system. Therefore, this paper describes the design and implementation of IDAO that applies Iterator pattern. IDAO gets effect that reduces complexity of transaction logic, system overload by database connection, and reduction of performance through container managed transaction.

키워드 : EJB(Enterprise JavaBeans), DAO(Data Access Object), Iterator pattern, IDAO(Integrated Data Access Object)

1. 서 론

큰 규모의 복잡한 문제를 일정한 기준에 의해 작은 조각으로 분할하고 적당한 크기로 분할된 문제를 먼저 해결한 후 보다 단순한 기초로부터 정교한 솔루션을 구축하는 것이 현재 각광을 받고 있는 컴포넌트 시스템이다[1]. 이러한 컴포넌트 개발 방법은 객체지향의 원리에 따라 업무 기능과 관련 데이터를 하나의 단위로 처리하므로 비즈니스 환경 및 기술 변화를 수용할 수 없는 현 기술의 한계와 업무통합 또는 분산환경, 이식성, 경량화 요구, 개발자 개인의 능력과 비자동화된 개발 절차에 따른 문제점 등의 한계에 따라 필요성이 제기되었다. 컴포넌트 개발 방법의 주된 목적은 한 번 설계한 것을 다른 상황에서도 반복적으로 사용하기를 원하는 재사용성에 있으며, 이렇게 함으로써 개발 생산성의 획기적인 향상을 도모하고 지속적인 품질의 개선을 가져올 수 있다. 또한, 계속 변화되는 상황에서 중앙 통제가 힘든 상황에 처해질 경우 쉽게 대체가 가능하고 설계와 구현의 재사용을 통하여 생산성을 향상시키고 효율적으로 테스트된 코드를 사용함으로써 신뢰성을 증가시키는 특징을 가지고 있다[2, 3]. 본 논문에서는 엔터프라이즈 애플리케이션을 위한 Java 진영에서 새롭게 내놓은 차세대 컴포넌트 기술 표준인 EJB (Enterprise JavaBeans) 기반 레거시 시스템에서 데이터베이스의 액세스를 캡슐화하는 DAO(Data Access Object)에 대해서 알아본다. 또한, 레거시 시스템에서 발생된 문제점인 DAO 트랜잭션 로직의 복잡성과 불필요한 DAO 생성 및 시스템의 과부하를 해결하고자 디자인 패턴의 행위 패턴에 속하는 Iterator 패턴을 적용하여 데이터베이스의 효율적 액세스를 위한 IDAO를 제안하고자 한다. 데이터베이스의 효율적 액세스를 위한 IDAO는 컨테이너 관리 트랜잭션을 통해

† 준 회 원 : 전북대학교 대학원 전산통계학과
†† 준 회 원 : 전북대학교 대학원 전산통계학과, 정인대학 사무정보계열 교수
††† 종신회원 : 전북대학교 자연과학대학 컴퓨터과학과 교수, 전북대학교 영상정보신기술연구소 연구원
†††† 정 회 원 : 전북대학교 자연과학대학 컴퓨터과학과 교수, 전북대학교 영상정보신기술연구소 연구원

논문접수 : 2001년 9월 19일, 심사완료 : 2001년 11월 26일

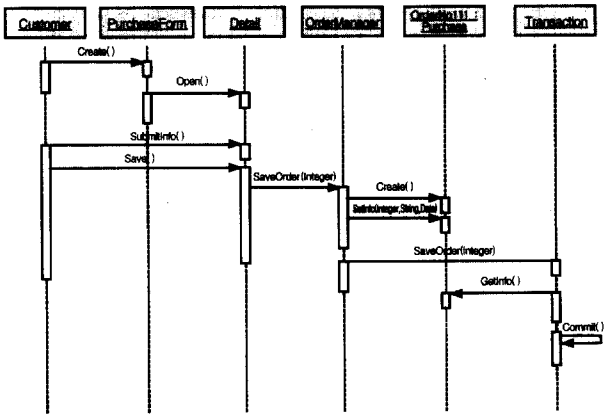
트랜잭션 조작에 관한 복잡성을 줄여주기 때문에 시스템의 과부하 감소와 시스템의 성능저하를 감소시키는 효과가 있었다. 본 논문의 구성은 2장에서는 레거시 시스템에서 DAO에 대해서 알아보고, DAO에서 발생되는 문제점에 대해서 알아본다. 3장에서는 Iterator 패턴의 적용과정 및 Iterator 패턴을 적용한 IDAO 설계에 대하여 설명한다. 4장에서는 Iterator 패턴을 적용한 IDAO의 구현과정을 IDAO 구현 프로세스 워크플로우를 통해 알아보고 실제 Iterator 패턴을 적용한 IDAO의 구현에 대해서 설명한다. 마지막 5장에서는 결론 및 향후 연구과제를 제시한다.

2. 레거시 시스템에서의 DAO 개요 및 문제점

본 장에서는 전자상거래 웹 사이트인 멀티 계층 웹 애플리케이션의 전체적인 개발 프로세스에서 고객이 상품을 구매하는 과정에서 해당 데이터의 상세한 정보를 데이터베이스에서 직접 가져오는 DAO에 대해서 분석해보고 문제점에 대해서 알아본다.

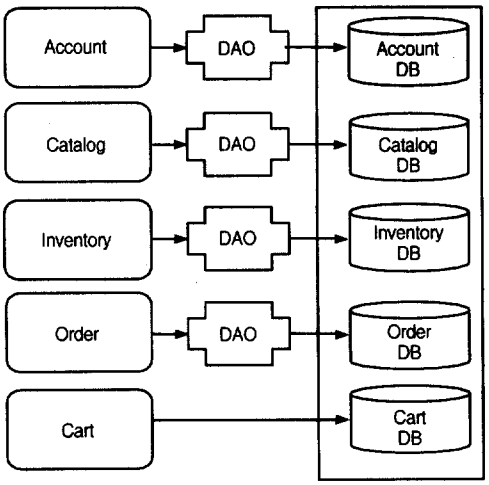
2.1 레거시 시스템에서의 DAO 개요

먼저 전자상거래에서 고객이 웹 브라우저를 통해 쇼핑몰의 상품 서버에 접속을 한 후, 상품을 구매하는 과정을 시퀀스 다이어그램으로 나타내면 (그림 1)과 같다.



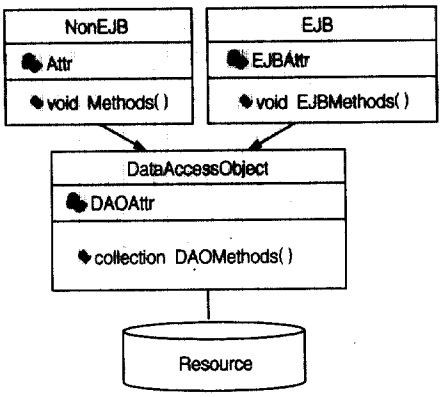
(그림 1) 상품 구매과정의 시퀀스 다이어그램

(그림 1)에서는 고객(Customer)이 웹 브라우저를 통해 상품 서버에 접속한 후, 쇼핑몰에서 제공하는 상품의 정보(PurchaseForm)를 열람한다. 매장, 상품, 특정 모델의 순서로 열람할 수 있으며, 사용자가 상품 정보를 열람하는 도중에 구매의향이 있는 상품들을 선택하면(OrderManager) 즉시 상품에 대한 대금을 지불하지 않고 일단 쇼핑백에 저장을 한다. 추후 한 번에 계산을 함으로써 매번 상품들을 선택할 때마다 해당 금액을 지불해야하는 불편함을 예방하고자 함이다. 쇼핑백을 열람하면 선택한 상품들의 가격 및 내역을 확인할 수 있으며, 금액을 지불하기 전에 상품 구매를 취소할 수 있다. 여러 가지 배송 방법 중에서 가장 자신에게 적합한 하나의 배송 방법을 선택한 후 모든 상품들에 대한 금액을 지불함으로써 거래가 완료되는 과정(transaction)을 보여주고 있다. (그림 1)에서 살펴본 상품 구매의 전체과정 중에서 사용자가 웹 브라우저를 통해 쇼핑몰 상품 서버에 접속한 후 해당되는 상품을 구매하는 과정에 필요한 각각의 해당 DAO(Data Access Object)에 대해서 살펴본다. 먼저, 사용자의 인증 단계로서 사용자 ID와 패스워드를 입력한다. 그러면 Account DAO가 Accout DB에 저장된 사용자 ID와 패스워드의 일치성 여부를 확인한 후, 저장되어진 정보와 입력한 정보가 일치하면 제품 카탈로그를 볼 수 있는 화면을 보여준다. 그렇지 않은 경우에는 다시금 사용자의 인증을 확인하는 단계를 수행한다. 제품 카탈로그 목록 보기에서는 제품 카탈로그에 있는 Catalog DAO가 Catalog DB에 접근하여 해당되는 데이터를 가져온다. 또한 선택된 제품에 대한 재고 상태를 알아보기 위해서 저장소에서는 Inventory DAO가 Inventory DB에 접근하여 해당되는 제품의 재고상황을 파악하여, 파악한 내용을 사용자에게 전달해주면 사용자는 해당되는 제품을 주문할 수 있다. 이때 Order DAO가 Order DB에 접근하여 주문 정보를 저장한다. Cart에 저장된 주문정보를 확인하고자 할 때에는 DB 내용에 대한 직접적인 조작은 없다. 따라서, 이때에는 정보를 조회만 하기 때문에 무상태 세션빈을 이용하여 Cart DB에 저장된 정보를 가져와서 보여주기만 하면 되는 과정을 (그림 2)에서 보여주고 있다.



(그림 2) 레거시 시스템에서의 DAO 구조도

따라서 위에서 언급한 각각의 해당 DAO(Data Access Object)에 대해서 살펴보도록 한다[4]. 먼저 DAO는 XML 데이터를 표현하고 데이터베이스 벤더의 독립적인 목적을 위한 객체이다. DAO는 데이터베이스 접근을 캡슐화하는데 이용되며, 대규모 배치에 적합하고, 독립적인 리소스 벤더와 독립적인 리소스 구현 및 빈 관리 퍼시스턴스에서 컨테이너 관리 퍼시스턴스까지의 이동을 용이하게 해주는 이점을 가진다. (그림 3)은 이러한 DAO 아키텍처를 보여주고 있다.



(그림 3) DAO 아키텍처

```
public class CatalogDAO {
   private Connection con ;
   public CatalogDAO(Connection con) {
        this.con = con ;
   }
   public Category getCategory(String categoryId)
   throws SQLException {
    String qstr = "select catid, name, descn
    from " + DatabaseNames.CATEGORY_TABLE
    + " where catid = '" +
        categoryId + "'" ;
         Statement stmt = con.createStatement( ) ;
         ResultSet rs = stmt.executeQuery(qstr) ;
         Category cat = null ;
         while (rs.next( )) {
              int i = 1 ;
              String catid = rs.getString(i++).trim( ) ;
              String name = rs.getString(i++) ;
              String descn = rs.getString(i++) ;
              cat = new Category(catid, name, descn) ;
         }
         rs.close( ) ;
         stmt.close( ) ;
         return cat ;
   }
    ..... 중략 .....

public class CatalogImpl implements CatalogModel {
   public Category getCategory(String categoryId) {
        Connection con = getDBConnection( ) ;
        try {
             CatalogDAO dao = new CatalogDAO(con) ;
             return dao.getCategory(categoryId) ;
        } catch (SQLException se) {
             throw new GeneralFailureException(se) ;
        } finally {
             try {
                  con.close( ) ;
             } catch (Exception ex) {
             throw new GeneralFailureException(ex) ;
    ..... 중략 .....
```

(그림 4) CatalogDAO 소스

(그림 3)을 통해 알 수 있듯이 DAO의 참여자(participants)는 NonEJB, EJB, DataAccessObject, Resource가 있다. NonEJB 클래스는 EJB-required( )를 제외한 비즈니스 메소드를 인스턴스화시켜 DAO를 이용하며, EJB 클래스는 EJB-required( )를 담당하며 비즈니스 메소드가 호출될 수 있는 환경을 제공한다. DataAccessObject 클래스는 리소스에 대한 오퍼레이션 추상화와 특정한 타입의 데이터 처리와 접근하기 위한 표준 API를 제공한다. Resource는 임의적인 API 방법에 의한 검색과 영속적인 리소스 데이터를 ConcreteDataObject에 제공한다. (그림 2)에서 언급한 레거시 시스템에서의 DAO 구조도에 해당되는 내용의 한 예로 CatalogDAO 소스에 대해서 살펴본다. (그림 4)의 CatalogDAO 클래스는 CatalogEJB에 의해 생성된 모든 SQL 문을 캡슐화하여 CatalogEJB에서 필요로 하는 객체로 데이터베이스에 저장된 관계 데이터를 매핑시킨다. Catalog 엔티티 빈은 CatalogImpl에서 상속받은 CatalogEJB에 의해 제공되는데, 사용자가 사용자 ID와 패스워드를 정확하게 입력한 후 사용자에게 최초로 보여주는 화면 목록으로 사용자에게 제품을 선택할 수 있도록 목록을 보여준다. 사용자에게 제품에 대한 목록을 보여주기 위해서는 Catalog 엔티티 빈에 대한 구현으로 CatalogDAO( )를 통해 Catalog 데이터베이스에 접근하여 제품 정보에 대한 각각의 필요한 메소드를 호출한다. 먼저, getCategory( )를 통해 categoryid, name, descn에 대한 정보, getProduct( )를 통해 제품 목록에 대한 제품의 id, 가격목록, 개당 가격 등과 같은 정보, getItem( )를 통해 사용자에 의해 선택된 제품의 id, 가격목록, 개당 가격의 정보를 얻어온다. 또한 searchProducts( )를 통해 선택된 제품의 정보를 반복 수행하여 제품의 현재 상태인 판매가능 및 품절 등에 대한 정보를 알려준다. getQueryStr( )에서는 사용자의 편의를 도모하기 위해 아이콘에서 입력한 모든 판매구분의 정보인 제품 판매, 제품 반품, 제품 반환, all 등에 관한 정보를 보여준다.

2.2 DAO의 문제점

본 장에서는 웹 애플리케이션으로 작성된 전자상거래 사이트에서 사용자에 의해 제품을 구매하는 전체 과정 중에서 일부인 Catalog 엔티티 빈에 해당하는 내용을 언급해봄으로써 전체적인 컴포넌트의 구성을 예측할 수 있다. 전체적인 컴포넌트의 구성은 여러 개로 나누어진 컴포넌트마다 서로 다른 각각의 DAO를 통해 데이터베이스에 접근한다. 이로 인한 문제점으로는 첫째, DAO가 해당되는 데이터를 조작시 서로 다른 트랜잭션에서 조작이 이루어지기 때문에 트랜잭션 로직을 복잡하게 하여 시스템에 과부하를 초래하게 된다. 둘째, 각 빈과 연결된 DAO는 빈이 인스턴스화 될 때 동시에 인스턴스화되므로 불필요한 DAO를 추가적으로 생성하게 된다. 셋째, 불필요한 DAO의 생성으로 인해 많은 메모리를 차지하여 시스템의 성능을 감소시키게 된다. 이러한 문제점을 해결하기 위해 3장에서는 설계의 재사용성을 높여주고 설계시간의 단축 및 시스템의 안정성을 향상시키는 Iterator 패턴을 적용하는 과정과 Iterator 패턴을 적용한 IDAO를 설계한다.

3. Iterator 패턴을 적용한 IDAO의 설계

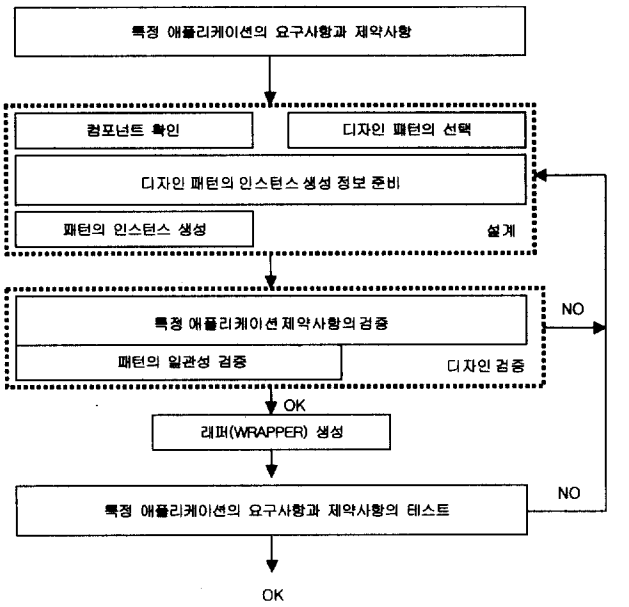
3.1 Iterator 패턴의 적용과정

디자인 패턴이란 미래의 비슷한 상황에서 다시 적용될 수 있는 과거에 잘 정의된 설계에 대한 정보를 기록하는 것이다[5]. 디자인 패턴을 EJB 컴포넌트 설계에 적용하면 설계 시간의 단축 및 개발자간의 의사소통 시간을 줄일 수 있다는 장점과 효율적이고 유용성이 좋은 디자인 패턴을 이용함으로써 시스템의 안정성과 성능을 높일 수 있는 장점을 가진다[6]. Gamma는 디자인 패턴을 생성, 구조, 행위 패턴으로 분류하였으며, 이를 다시 패턴이 하는 일이 무엇인지를 반영하는 목적과 패턴이 객체 또는 클래스에 처음으로 적용될 때의 명확한 범위에 따라 구분을 해보면 <표 1>과 같이 표현할 수 있다[7].

〈표 1〉 디자인 패턴의 유형

목적 / 범위	생성 패턴	구조 패턴	행위 패턴
클래스	Factory Method	Adapter(class)	Interpreter Template Method
객 체	Abstract Factory Builder Prototype Singleton	Adapter(object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command ***Iterator*** Mediator Observer State Strategy Visitor

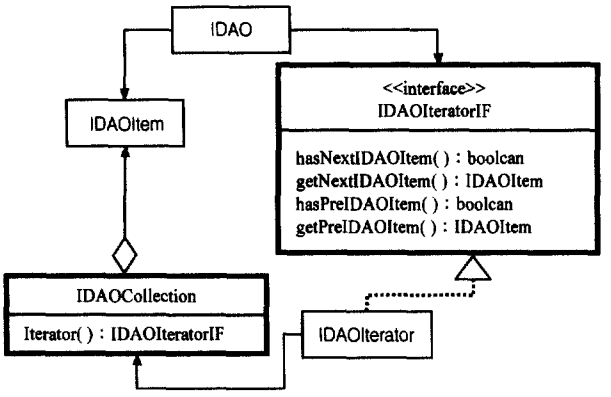
본 논문에서 제안한 IDAO는 디자인 패턴의 유형 중에서 행위 패턴에 속하는 Iterator 패턴을 적용한다. Iterator 패턴은 디자인 패턴 중에서 가장 많이 이용되며, 가장 간단한 패턴중의 하나로서 콜렉션에서 객체를 순차적으로 접근하기 위해서 메소드를 선언한 후 인터페이스를 정의한다. 그러한 인터페이스를 통해 콜렉션을 접근하는 클래스는 그 인터페이스를 구현하는 클래스와 독립적으로 존재한다. Iterator 패턴을 이용하면 크게 다음과 같은 이점을 가진다. 먼저, Iterator 패턴을 적용시 가장 중요한 결과로 프로그램이 수정될 동안 데이터를 통해 반복 수행된다. 둘째로 열거형 클래스는 컨테이너 클래스의 기본적인 데이터 구조의 제한된 접근을 필요로 할 때 데이터를 통해 전달된다. 클래스를 포함한 같은 모듈 내에서 정의된 클래스는 클래스 변수가 포함한 데이터를 액세스해야 한다. 셋째로 디자인 패턴에서 Iterator 패턴은 크게 두 가지 타입인 외부 Iterator와 내부 Iterator를 갖는다. 지금까지 기술한 일반적인 내용이 바로 외부 Iterator 패턴이었다. 내부 Iterator 패턴은 사용자로부터 어떤 특별한 요구없이 직접적으로 각각의 요소에 대한 오퍼레이션을 수행하는 전체적인 콜렉션을 통해 데이터를 전달하는 메소드이다. 내부 Interator 패턴은 Java에서 비교적 흔하게 사용되지 않는다. 그러나 0과 1 사이의 상태값을 표준화시키는 메소드나 특별한 경우에 모든 문자열을 전환시킬 때 사용된다. 일반적으로 외부 Iterator 패턴은 호출 프로그램이 직접적으로 각각의 요소를 접근하며, 오퍼레이션의 수행을 결정해주기 때문에 더 많은 컨트롤을 필요로 한다. 결과적으로 Iterator 패턴을 적용하면 데이터 집합에 대해 수행하는 방법을 나타내지 않고 데이터 요소의 집합을 통해서 정의된 방법을 제공하는 이점을 가지고 있다[8]. 본 논문에서 제안한 IDAO는 Iterator 패턴을 적용하였는데, (그림 5)는 이러한 Iterator 패턴의 컴포넌트화 과정을 보여준다[9].



(그림 5) Iterator 패턴의 컴포넌트화 과정

(그림 5)에서는 특정 애플리케이션의 요구사항과 제약사항에 의하여 컴포넌트를 확인하며, 디자인 패턴은 컴포넌트와의 상호작용을 위한 조건으로서 적합한 패턴을 선택한다. 디자인 패턴의 인스턴스 생성 정보 준비 단계에서는 디자인 패턴 요소와 디자인 요소간의 연관 관계 및 선택된 디자인 패턴간의 관계를 포함한다. 디자인 패턴 요소는 디자인 패턴에서 포함하고 있는 구성요소와 관련되고 컴포넌트에 의해 발생되는 이벤트와 컴포넌트에 의해 제공되는 서비스를 포함한다. 패턴의 인스턴스 생성은 소프트웨어 설계를 명세하는 부분에서 디자인 패턴을 추상 솔루션으로 변환하는데 이용한다. 설계 검증 단계에서는 특정 애플리케이션 제약사항의 검증과 패턴의 일관성을 검증하는 단계로 구분된다. 먼저 특정 애플리케이션 제약사항의 검증 단계는 구조와 상호작용의 일관성 검증과 컴포넌트에 의해 제공된 서비스를 검증한다. 패턴의 일관성 검증 단계는 디자인 패턴의 인스턴스 생성이 초기의 디자인 패턴과 같은 제약조건을 가지는 것을 검증한다. 래퍼(wrapper) 생성 단계에서는 컴포넌트의 데코레이터(decorator)로서 동작한다. 모든 컴포넌트 사이에서의 상호작용은 래퍼를 통해 이루어진다. (그림 5)에서 언급한 Iterator 패턴의 컴포넌트화 과정을 통해서 본 논문에서 제안한 IDAO에 적용시켜본 결과를 (그림 6)에서 보여주고 있다. (그림 6)의

다이어그램은 iDao로 구성된 사용자 인터페이스 클래스들이 IDAO의 조합 클래스로 표현된다. IDAO의 인스턴스는 IDAOCollection 객체에 의해 캡슐화된 집합에서 IDAOItem 객체들을 표시하기를 요청한다. IDAO 객체는 IDAOCollection 객체를 직접적으로 접근하지 못한다. 대신에 IDAO에게는 IDAOIteratorIF 인터페이스를 구현하는 객체를 제공한다. IDAOIteratorIF 인터페이스는 결과적으로 발생되는 IDAOItem 객체들의 내용을 가져오는 메소드를 정의한다.



(그림 6) IDAO에 적용된 Iterator 패턴의 구조

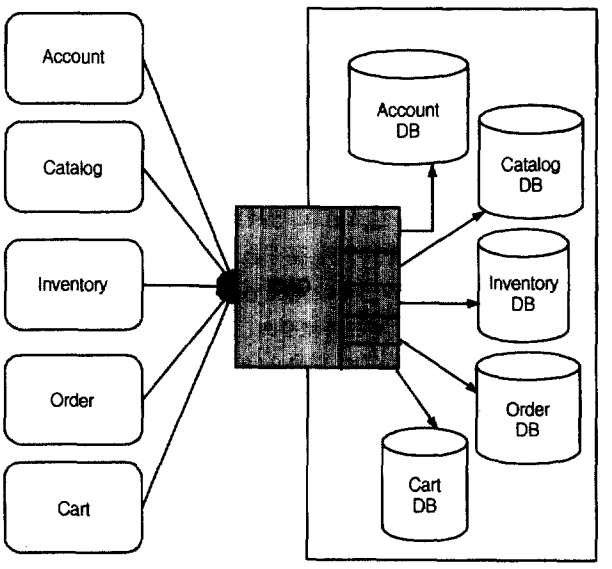
```
public interface IDAOIteratorIF {
  public boolean hasNextIDAOItem( );
  public IDAOItem getNextIDAOItem( );
  public boolean hasPreIDAOItem( );
  public IDAOItem getPreIDAOItem( );
  }
public class IDAOCollection {
  public IDAOIteratorIF iterator( ) {
    return new IDAOIterator( );
  } // iterator( )
  public class IDAOIterator implements IDAOIteratorIF {
    public boolean hasNextIDAOItem( ) {
    ......
    } // hasNextIDAOItem( )
    public IDAOItem getNextIDAOItem( ) {
    ......
    } // getNextIDAOItem( )
    public boolean hasPreIDAOItem( ) {
    ......
    } // hasPreIDAOItem( )
    public IDAOItem getPreIDAOItem( ) {
    ......
    } // getPreIDAOItem( )
  } // class IDAOIterator
  ......
} // class IDAOCollection
```

(그림 7) IDAO에 적용된 Iterator 패턴

(그림 7)은 실제로 (그림 6)을 기반으로 IDAO에 적용된 Iterator 패턴의 적용된 소스를 보여주고 있다. (그림 7)에서 적용된 코드를 살펴보면 설계를 구현하기 위한 스켈레톤 코드를 발견할 수 있다. IDAOCollection 클래스의 스켈레톤 리스트를 살펴보면, 다른 클래스들이 IDAOCollection 객체의 내용에 관한 반복되는 객체를 얻는데 이용되는 iterator( )를 포함하고 있다. 또한 IDAOCollection 클래스는 iterator 메소드의 인스턴스 전용(private) 클래스를 포함한다. 위와 같이 Iterator 패턴을 사용하여 여러 종류의 데이터를 순차적으로 접근할 수 있는 단일 인터페이스인 IDAOIteratorIF를 제공하여 DB 커넥션 수를 하나로 만들고 데이터의 종류에 따라서 불필요하게 생성되는 객체의 인스턴스를 줄일 수 있다. 즉, 기존의 경우 데이터 접근방식이 동일함에도 불구하고 접근하고자 하는 데이터 종류에 따라 AccountDAO, CatalogDAO, InventoryDAO, OrderDAO 등으로 여러 객체를 이용하여 접근되던 과정을 IDAO로 단순화시켰다. 또한, 이렇게 Iterator 패턴이 적용된 IDAO를 사용함으로써 각각의 데이터 접근 객체(DAO)의 메소드에 따라 트랜잭션 과부하가 발생되는 원인을 IDAO의 메소드에서만 트랜잭션을 할당하도록 함으로써 트랜잭션 처리에 대한 수를 줄일 수 있었다.

3.2 Iterator 패턴을 적용한 IDAO 설계

(그림 5)의 디자인 패턴의 컴포넌트화 과정을 통해서 본 논문에서는 Iterator 패턴을 적용한 IDAO를 (그림 8)에서 보여주고 있다.



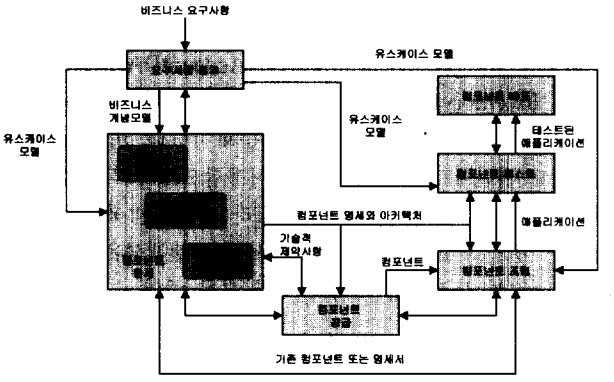
(그림 8) Iterator 패턴을 적용한 IDAO

(그림 8)은 DAO의 복잡한 트랜잭션 로직과 불필요한 DAO 생성 및 시스템의 과부하를 해결하기 위해 개선한 IDAO의 구조도이다. 각각의 컴포넌트에서 IDAO 패턴에 접근하면 각 데이터베이스의 특징을 가진 IDAO에서는 해당 데이터베이스에 실시간(real-time)으로 접근한다. 또한, IDAO 패턴은 다수의 DAO가 작업하는 내용을 트랜잭션으로 관리하는 것보다 통합된 IDAO 패턴을 이용하는 것이 컨테이너 관리 트랜잭션에서 트랜잭션 조작에 관한 복잡성을 줄여주기 때문에 결과적으로 시스템의 과부하를 감소시킬 수 있다. 또한, 시스템의 성능과 확장성 및 효율성을 증대하는 효과를 기대할 수 있다.

4. Iterator 패턴을 적용한 IDAO 구현

4.1 IDAO의 구현 프로세스 워크플로우

본 논문에서 제안한 EJB 기반 IDAO를 구현하는데 적용한 전체적인 구현 프로세스 워크플로우를 (그림 9)에서 보여주고 있다[10].



(그림 9) IDAO의 구현 프로세스 워크플로우

박스로 표현된 것은 의미있는 산출물을 생산해내는 일련의 활동인 워크플로우를 나타낸다. 전체 구현 프로세스 워크플로우에서 먼저 요구사항 정의 워크플로우는 사용자에 의한 요구사항이 전달되면 유스케이스 모델과 비즈니스 개념 모델을 전단계 결과물로서 입력받아 명세 워크플로우에 전달한다. 전달받은 명세 워크플로우는 레거시 시스템, 패키지, 데이터베이스, 특정 아키텍처 또는 툴의 사용과 같은 기술적인 제약사항 정보를 이용하여 컴포넌트 명세와 컴포넌트 아키텍처를 생성한다. 명세 워크플로우의 결과물들은 컴포넌트 구매나 개발에 대한 결정을 하기 위해 컴포넌트 공급 워크플로우에서 이용된다. 또한 명세 워크플로우의 결과물들은 조립 워크플로우에서 정확한 컴포넌트 통합을 위한 지침을 제공하며, 테스트 워크플로우에서 테스트 시나리오를 생성하는데 필요한 입력물로 이용된다. 공급 워크플로우에서는 컴포넌트를 개발하거나, 제 3의 컴포넌트 공급자를 통해 구매 또는 레거시 컴포넌트나 소프트웨어를 재사용, 통합, 정제해서 필요한 컴포넌트를 사용할 수 있도록 해준다. 또한 공급 워크플로우에서는 컴포넌트 조립 전에 컴포넌트에 대한 단위 테스트를 수행한다. 컴포넌트 조립 워크플로우에서는 사용자 요구사항에 적합한 애플리케이션을 만들기 위해 컴포넌트를 레거시 시스템이나 사용자 인터페이스들과 적절히 통합한다. 컴포넌트 테스트 워크플로우에서는 사용자 요구사항에 적합한 데이터를 전달하기 전에 컴포넌트, 컴포넌트의 그룹, 서브 시스템 또는 전체 시스템과 같은 단위 테스팅, 모듈 테스팅, 컴포넌트 테스팅 등의 많은 유형의 테스팅을 수행한다. 컴포넌트 배포 워크플로우에서는 컴포넌트 배포를 위해 준비 사항을 빠짐없이 수행한 후 배포 마법사의 도움을 받아 서버의 위치 정보 등을 알려주면 개발자 디스크립터 작성 등과 같은 배포에 관련된 모든 작업을 자동적으로 수행한다.

4.2 Iterator 패턴을 적용한 IDAO 구현

실제적으로 본 논문에서 Iterator 패턴을 적용한 IDAO의 소스 일부분을 살펴보면 (그림 10)과 같다. (그림 10)은 Iterator 패턴을 적용하여 J2EE 플랫폼 상에서 구현되는 컴포넌트 중 데이터에 접근하는 코드를 통합시킴으로써 엔티티 빈의 성능을 향상시키는 소스이다.

```
package IDAOClass

public class DAOClass {
  public DAOClass( ) {
  }
  public IDAO getInput(Object obj) {
  }
  public IDAO getOutput(Object obj) {
  }
  public void setInput(Object obj) {
  }
  public void setOutput(Object obj) {
  }
}
```

```
package IDAOClass ;

import java.rmi.RemoteException ;
import javax.rmi.PortableRemoteObject ;
import javax.naming.InnitialContext ;
import javax.ejb.CreateException ;

public class IDAOFactory extends IDAOIterator {
  IDAO iDao = null ;
  DAOClass daoclass = null ;

  public static Connection getConnetion( ) throws javax.naming.
  NamingException {
    InitialContext initial = new InitialContext( ) ;
    Object objref = initial.lookup(JNDINames.CONNECTION_EJBHOME) ;
    return (Connection)
      PortableRemoteObject.narrow(objref,
        Connection.class) ;
  }
  // Connection를 얻어오는 과정으로 Connection의 최소화, 시스템의
    과부하를 감소시킴, 메모리를 적게 차지하여 시스템의 속도를 증가
    시킴
public static IDAO getDAOClass(DAOClass daoclass)
{
   iDAO = daoclass.getInput(daoclass) ;
   return (IDAO) iDAO ;
 }
public static void create(IDAO iDAO) {
   iDAO.create(iDAO.GetInput(daoclass)) ;
 }
public static void store(IDAO iDAO) {
   iDAO.store(DAO.GetInput(daoclass)) ;
 }
public static void remove(IDAO iDAO) {
   iDAO.remote(iDAO.GetInput(daoclass)) ;
 }
}
// DAO의 역할을 기술하는 부분으로 트랜잭션을 최소화함
```

(그림 10) Iterator 패턴을 적용한 IDAO의 소스

즉, 엔티티 빈이 자주 데이터베이스에 접근하는 코드를 내포함으로써 엔티티 빈의 생명주기 동안 데이터베이스의 커넥션이 자주 발생되어지는 것을 최대한으로 억제하였다. 트랜잭션의 경우 DAO의 데이터베이스 커넥션 부분을 getConnection ( ) 메소드로 단일화하여 트랜잭션의 처리단위를 감소시켰다. 또한, 이 커넥션 관리를 웹 애플리케이션 서버에 위임함으로써 커넥션 풀링(pooling)이 발생하도록 하여 데이터베이스의 커넥션 수를 감소시켜 시스템의 성능저하를 감소시키고 과부하도 감소시켰다. 또한 이 IDAO에서 적용된 클래스는 입력

객체를 setInput( ), getInput( ) 등의 메소드를 이용하여 필터 형식으로 만들었기 때문에 여러 DAO를 거쳐 수행해야 할 일을 단 하나의 메소드만을 통해 처리할 수 있도록 하여 불필요한 DAO 생성을 하지 않게 하여 시스템의 확장성 및 효율성을 향상시켰다.

4.3 IDAO의 의의

본 논문에서는 레거시 시스템에서 데이터베이스 접근을 캡슐화하는데 이용되는 DAO 트랜잭션 로직의 복잡성과 불필요한 DAO 생성 및 시스템 과부하의 문제점을 해결하고자 IDAO를 제안하였다. 제안한 IDAO는 콜렉션에서 객체를 순차적으로 접근하기 위해 메소드를 선언하는 Iterator 패턴을 적용하였다. Iterator 패턴을 적용한 결과 인터페이스를 통해 단지 콜렉션을 접근하는 클래스가 그 인터페이스를 구현하는 클래스와 독립적으로 존재하기 때문에 데이터 집합에 대해 수행되는 방법을 나타내지 않고 데이터 요소의 집합을 통해서 정의된 방법을 제공한다. 따라서, IDAO는 <표 2>에서와 같이 레거시 시스템과의 성능측정 요소를 비교할 때 시스템의 과부하정도, 트랜잭션 복잡성, 메소드 효율성 등은 DB 커넥션의 수와 트랜잭션 처리 수, DAO 개수 등에 의해 좌우됨을 알 수 있다.

〈표 2〉 레거시 시스템과의 성능측정 요소

패턴 / 비교관점	DAO	IDAO
DB 커넥션의 수	DAO 객체 수 만큼	한 개
트랜잭션 처리 수	DAO 객체의 메소드 수 만큼	IDAO의 메소드 수 만큼
DAO 개수	DAO 객체의 수 만큼	한 개

〈표 3〉 DAO와 IDAO의 성능비교

비교관점 / 패턴	DAO	IDAO
시스템 과부하	높음	다소 낮음
트랜잭션 로직의 복잡성	크다	작다
호출시간	길다	짧다
메모리 효율성	나쁘다	좋다
불필요한 DAO 생성	존재함	존재하지 않음
기능향상 및 확장성	지원하지 않음	지원함

<표 2>를 기반으로 레거시 시스템의 DAO와 본 논문에서 제안한 IDAO의 성능을 비교해보면 <표 3>과 같이 성능비교표를 만들어 볼 수 있다. <표 3>에서 보는 바와 같이 트랜잭션에 참가하는 메소드의 수를 줄이고 여러 개의 DAO 객체를 두지 않음으로써 불필요하게 중복되는 로직을 줄였다. 또한, DB 커넥션 수를 IDAO에 하나만 있게 함으로써 DAO에서의 문제점인 트랜잭션 조작에 관한 복잡성과 불필요한 DAO 생성 및 시스템의 과부하를 최소화시킴으로써 시스템의 성능을 향상시켰다.

5. 결론 및 향후 연구과제

본 논문에서는 EJB 기반 레거시 시스템에서 데이터베이스의 액세스를 캡슐화하는 DAO에서 발생된 DAO 트랜잭션 로직의 복잡성과 불필요한 DAO 생성 및 시스템의 과부하를 감소시키고자 디자인 패턴의 행위 패턴에 속하는 Iterator 패턴을 적용한 IDAO를 제안하였다. 논문에서 제안한 IDAO는 데이터베이스의 효율적 액세스를 위해 컨테이너 관리 트랜잭션을 통해 트랜잭션에 관한 복잡성을 줄여 시스템의 과부하 문제와 시스템의 성능저하를 감소시키는 효과를 얻었다. 향후 연구과제로는 JMS나 Connector 아키텍처 등의 새로운 Java 기술이 채택된 e-비즈니스 시스템 상에서 적용될 수 있는 IDAO를 설계할 필요가 있고 또한 BEA의 WebLogic과 같은 특정 프로덕트와 관련된 최적화된 IDAO의 설계도 필요하다.

참 고 문 헌

[1] Clemens Szyperski, "Component Software : Beyond Object-Oriented Programming," Addison Wesley Lognman, Inc., 1998.

[2] Peter Herzum, Oliver Sims, "The Business Component Approach : Business Object Design and Implementation II," OOPSLA '98 Workshop Proceedings, UK : Spring-Verlag, 1998.

[3] Desmond F. D'Souza, Alan Cameron Wills, "Objects, Components and Frameworks with UML : The Catalysis Approach," Addison Wesley Longman, Inc., 1999.

[4] Deepak Alur, John Crupi, Dan Malks, "Core J2EE Patterns : Best Practices and Design Strategies," Prentice Hall PTR, 2001.

[5] Mark Grand, "Patterns in Java, Volumel : A Catalog of Reusable Design Patterns Illustrated with UML," John Wiley & Sons, Inc., 1998.

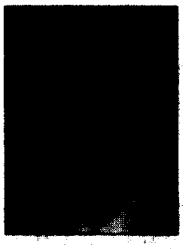
[6] Karl Rege, "Design Patterns for Component-Oriented Software Development," in Proceedings of EUROMICRO, pp. 220-228, 1999.

[7] Erich Gamma, Richard Helm, Raplh Johnson, John Vlissides, "Design Patterns : Elements of Reusable Object-Oriented Software," Addison Wesley Longman, Inc., 1995.

[8] Cooper, James William, "Java Design Patterns : A Tutorial," Addison Wesley & Sons, Inc., 2000.

[9] Stephen S. Yau, Ning Dong, "Integration in Component-Based Software Development Using Design Patterns," in Proceedings of COMPSAC, pp.369-374, 2000.

[10] John Cheesman and John Daniels, "UML Components - A Simple Process for Specifying Component-Based Software," Addison Wesley Longman, Inc., 2001.



최 성 만

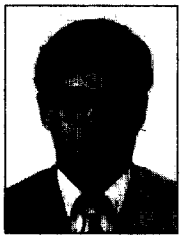
e-mail : sm3099@cs.chonbuk.ac.kr

1999년 전주대학교 전자계산학과 졸업 (학사)

2000년~현재 전북대학교 대학원 전산통계학과 석사과정

1997년~1999년 정인대학 사무자동화과 조교

관심분야 : 소프트웨어공학, 컴포넌트 기반 소프트웨어 개발 방법론, UML 기반 모델링, 디자인 패턴, 에이전트공학



이 정 열

e-mail : ljy8383@hanmail.net

1984년 전북대학교 전산통계학과 졸업 (학사)

1986년 동국대학교 경영대학원 전자계산학과 졸업(석사)

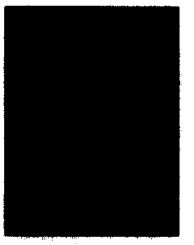
1994년 강원대학교 대학원 전자계산학과 졸업(석사)

1996년~현재 전북대학교 대학원 전산통계학과 박사과정

1984년~1993년 농림수산부 통계관실(전산실) 근무

1995년~현재 정인대학 사무정보계열 조교수

관심분야 : 소프트웨어공학, 전자상거래, 모바일 컴퓨팅



유 철 중

e-mail : cjyoo@moak.chonbuk.ac.kr

1982년 전북대학교 전산통계학과 졸업 (학사)

1984년 전남대학교 대학원 계산통계학과 졸업(석사)

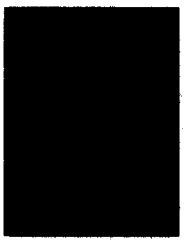
1994년 전북대학교 대학원 전자계산학과 졸업(박사)

1982년~1985년 전북대학교 전자계산소 조교

1985년~1996년 전주기전여자대학 전자계산과 부교수

1997년~현재 전북대학교 자연과학대학 컴퓨터과학과 조교수

관심분야 : 소프트웨어공학, 지리정보시스템, 객체 및 컴포넌트 기술, 에이전트공학, 멀티미디어, HCI, 분산객체 컴퓨팅, 인지과학



장 옥 배

e-mail : okjang@moak.chonbuk.ac.kr

1966년 고려대학교 수학과 졸업(학사)

1973년 고려대학교 수학과 졸업(석사)

1974년~1980년 죠지아 주립대, 오하이오 주립대 박사과정 수료

1988년 산타바바라대 대학원(Ph. D.)

1980년~현재 전북대학교 자연과학대학 컴퓨터과학과 교수

관심분야 : 소프트웨어공학, 전산교육, 수치해석, 인공지능