객체지향 웹 애플리케이션 개발을 위한 메타 클래스

서울대학교 경영학과 노상규
서울대학교 경영학과 우한균

ABSTRACT

Web-based applications development using object technology is a promising method to significantly improve the systems development productivity. However, its touted benefits have not been fully realized due to the lack of conceptual object modeling semantics in object-oriented programming languages such as Java and an impedance mismatch between programming languages and database languages. We propose a set of meta-classes in Java to support the conceptual object modeling semantics and to provide standard user interfaces in HTML. A system developer can efficiently build platform- independent object-oriented web applications by re-using these classes.

I. 연구의 배경 및 필요성

객체지향 시스템과 웹 애플리케이션 개발은 시스템 개발 생산성을 현저히 향상시킬 수 있는 유력한 방법이다. 객체지향 시스템 개발은 개발자가 설계 및 구현에 있어서의 구성요소를 재활용할 수 있기 때문에 전통적 개발 방법보다 효율적일 수 있는 잠재력을 가지고 있다[Coad, 1995; Yourdon, 1994; Rumbaugh, et. al., 1991]. 웹 애플리케이션 개발은 자체의 플랫폼 독립성, 시스템 유지의 용이함 등으로 더욱 큰 인기를 얻어나가고 있다[Orfali and Harkey, 1997].

그러나, Java와 같은 객체지향 언어를 이용한 웹 애플리케이션 개발은 각각의 영역에서 오는 문제점으로 인해 쉽게 통합되기 어렵다. 우선, 객체지향 언어를 이용하여 개념적 객체 모델을 구현하는 것은 많은 작업을 요구하는데, 이는 객체지향 언어가 개념적 객체 모델의 의미, 특히 Relationship이나 Association등을 완전하게 지원하지는 못하기 때문이다[Ling and Teo, 1993; Bertino, et. al., 1992, Kilian, 1991]. 객체지향 언어에서 제공하는 클래스, 인스턴스 변수, 내장 객체 등을 이용하여 객체 모델 개념을 구현하는 것은 메타데이타 질의가 어렵다는 점, 제약조건 정의가 중복된다는 점 등의 문제점을 야기시킨다[March and Rho, 1996, 1998]. 둘째로, 웹 애플리케이션 개발에서는 프로그래밍 언어(예를 들면 Java)와 데이타베이스 언어(예를 들면 SQL) 사이의 상호 불일치가 발생하는데[Copeland and Maier, 1984; Javasoft, 1998], 이는 객체 지향 패러다임이 개발 전체 프로세스에 적용되는 것이 아니라, 사용자 인터페이스나 미들 웨어에 부분적으로 적용되기 때문이다. 또한 이런 객체지향 패러다임의 부분적인 적용은 설계 및 구현에서의 구성요소를 재활용하는 데에 있어서 제약을 가져온다. 재활용은 모델 영역 전체 수준에서가 아니라 사용자 인터페이스 수준에서만 발생한다.

이런 문제들을 해결하기 위해 본 연구에서는 Java를 이용하여 UML(Unified Modeling Language)[Folwer and Scott, 1997]을 지원하는 메타클래스 집합(Object System)과 Java Servlet을 이용하여 HTML 형식의 표준 인터페이스를 제공하는 또다른 클래스 집합(Interface System)을 제안한다. 시스템 개발자는 플랫폼 독립적인 객체지향 웹 애플리케이션 개발 노력을 감소시키기 위해 이런 클래스들을 재활용할 수 있다.

II. 객체지향 웹 애플리케이션 개발

객체지향 언어가 개념적 객체 모델의 의미를 제대로 지원하지 못하는 문제를 해결하기 위해, March와 Rho[1996, 1998]는 "객체지향 프로그래밍 언어를 이용한 개념적 데이터 모델 구현에 관한 연구(A Semantic Object-Oriented Data Access System, SOODAS)"에서 E-R 방법론과 집합 수준의 질의 언어를 지원하는 메타 클래스 집합을 제안하였다. SOODAS는 객체 모델 또는 데이타 모델의 정의 및 유지, 엔터티 무결성 원칙과 관계 카디널러티 조건을 포함한 자체의 제약조건을 지원한다. 또한 어떤 객체의 메소드와도 통합될 수 있고 쉽게 정의되는 질의 언어를 제공한다. 하지만 SOODAS는 몇가지 제한점을 가지고 있는데, 객체의 영구 저장을 Smalltalk가상 메모리에 의존한다는 점, 독립 실행형 애플리케이션으로 구현되었는다는 점 등이다.

JDBC나 J/SQL을 이용한 Java 기반 웹 애플리케이션 개발에서 개발자는 SQL과 Java 모두를 알아야할 뿐만 아니라, 각각의 언어에서 애플리케이션의 데이타가 어떻게 표현되어야 하는지에 대해서도 익숙해야만 한다[Javasoft, 1998]. 이런 문제들을 해결하기 위해 Javasoft는 관계형 데이타베이스의 데이타에 대응하는 Java 객체를 자동으로 생성해주는 JavaBlend를 제안하였다. 그러나 JavaBlend 또한 모델 구성요소의 제한적 재활용 문제를 해결하지는 못하였다. 본 연구에서는 SOODAS를 Java 클래스로 변환하고, 이를 웹 애플리케이션 개발을 지원할 수 있도록 확장할 것이다.

III. 클래스의 구조

본 연구의 시스템은 Object System과 Interface System, 두 개의 클래스 집합으로 구성된다. Object System은 UML을 지원하고, Interface System은 HTML 테이블과 폼 형식의 재활용 가능한 표준 인터페이스를 제공한다.

1. Object System

그림 1에서 보여지는 것과 같이 Object System은 UML 방법론을 지원하기 위한 4개의 자바 클래스 —*ObjectBase, PermanentObject, DomainObject, Relationship*—로 구성되어 있다. 이 클래스들의 최종적인 부모는 *java.lang.Object* 이며, *ObjectBase*와 *PermanentObject*는 이를 상속한다.

*ObjectBase*는 객체를 형성하고, 객체 인스턴스를 저장한다. 현재 구현된 시스템에서 모든 객체들은 collections[Lea, 1997] 패키지의 RBMap 클래스의 인스턴스로 유지되고, PSE for Java[ObjectDesign, 1998]에 저장된다.

*PermanentObject*는 각각의 객체에 영속성을 부여한다. 또한 객체 인스턴스를 추가, 조회, 삭제할 수 있는 기능 —**AddInstance(PermanentObject e), Instances(String className), FindInstance (String className, String anID), DeleteInstance(PermanentObject e)**—을 제공한다. 이 메소드들은 *ObjectBase*의 메소드를 이용하여 정의된다. *PermanentObject*는 클래스들이 E-R 방법론의 외부 식별자를 지원하도록 **id()**를 추상 메소드로 정의한다. 이 메소드는 엔터티 무결성 조건을 집행하는 데에 사용된다.

UML 방법론의 type 이나 class에 해당하는 모든 클래스(예를 들어 *Employee* 나 *Department*)는 *DomainObject*의 서브클래스이다. *DomainObject*는 *PermanentObject*로부터 영속성을 상속한다. 또한 자신과 관련된 *Relationship*인스턴스를 응답하는 **Relationships(String className)**을 제공한다.

*Relationship*은 *PermanentObject*의 서브클래스로 UML 방법론의 Association을 지원한다. Association은 이 클래스의 인스턴스로 정의된다[March and Rho, 1996, 1998]. 각 Association은 이름(name), 관련된 클래스(class1, class2), Multiplicity 정

의(min1, min2, max1, max2), 역할 이름(rName1, rName2), 관련된 인스턴스의 쌍(instances1, instances2)으로 구성된다. *Relationship*은 자신의 인스턴스를 생성하기 위한 생성자를 제공한다. 또한 모든 Association 데이타와 제약조건을 통제한다. 메소드 **checkRef(i1, i2, rName2), checkMax(i1, i2, rName2), checkMin(i1, i2, rName2)**는 각각 참조무결성 조건, 최대 카디낼러티 조건, 최소 카디낼러티 조건을 집행한다.

2. Interface System

Interface System은 인스턴스의 열람과 관리를 위한 재활용 가능한 표준 인터페이스를 제공하며, 4개의 Java Servlet 클래스—*DomainObjectInterface, DomainObjectAdd, DomainObjectList, DomainObjectDetail*로 구성된다(그림 2). Interface System의 모든 클래스는 *javax.servlet.HttpServlet*의 서브클래스이다. Java Servlet은 네크워크 상에서 동적으로 로드될 수 있는 서버사이드 객체이다 [Javasoft, 1997].

*DomainObjectInterface*는 재활용 가능한 Html 문서의 구성요소(인스턴스의 리스트나 객체 입력, 수정, 삭제를 위한 입력 폼 등)를 제공한다. *DomainObjectInterface*는 *DomainObjectAdd, DomainObjectList, DomainObjectDetail*의 수퍼 클래스이다.

*DomainObjectAdd*는 *DomainObjectInterface*로부터 상속한 메소드 **Add(String className, PrintWriter out)**를 이용하여 인스턴스를 추가하기 위한 HTML 문서와 기능을 제공한다. *DomainObjectList*는 *DomainObjectInterface*로부터 상속한 메소드 **List (String className, PrintWriter out)**를 이용하여 인스턴스 전체를 리스트하고, 상세 화면으로 들어가기 위한 링크를 제공한다. *DomainObjectDetail*은 *DomainObjectInterface*로부터 상속한 메소드 **Detail(String className, PrintWriter**

out)을 이용하여 인스턴스를 수정, 삭제하기 위한 HTML 문서와 기능을 제공한다.

IV. 시스템 구현 예

개발자는 UML 방법론에서의 type/class를 *DomainObject*의 서브클래스로, association을 *Relationship*의 인스턴스로 정의함으로써 쉽게 프로토타입(prototype)을 구현할 수 있다. 또한 Interface System의 클래스(*DomianObjectAdd, DomianObjectList, DomianObjectDetail*)를 상속함으로써 HTML 테이블과 폼 형식의 표준 사용자 인터페이스를 구현할 수 있다. 이번 장에서는 Object System과 Interface System이 어떤 방식으로 그림 3의 객체 모델 예를 웹 애플리케이션으로 구현하는데 사용되는지를 보일 것이다.

1. UML 방법론의 지원

1.1. Type/Class, 외부 식별자

Type/class는 *DomainObject*의 서브 클래스로, 속성은 인스턴스 변수로 구현된다. 외부 식별자는 해당 클래스의 인스턴스 메소드인 **id()**를 정의함으로써 구현된다. 클래스 속성은 정적 변수로 구현된다. 그림 3의 *Employee*는 다음과 같이 정의된다. 아래 예에서 *Employee*의 외부 식별자는 eno이고, **id()**는 eno의 값을 응답하도록 정의되어 있다. **id()**가 *PermanentObject*의 추상 메소드이기 때문에, 모든 클래스는 **id()**를 인스턴스 메소드로 반드시 정의해야 한다.

```
public class Employee extends DomainObject  {
    // instance variables
    public String eno;
```

```
        public String ename;
        ...
        // instance methods
        public String id() {
                return eno;
    }
    ...
        // class variables
        public static double FitPerExemption;
        public static double FitRate;
        ...
    }
```

1.2. Association

Association은 *Relationship*의 인스턴스로 구현된다. 관계의 생성은 다음과 같은 두 종류의 생성자를 통해 이루어지게 된다. 하나는 role name이 정해지지 않았을 경우에 사용되며, 다른 하나는 role name이 주어질 경우에 사용된다. *Employee*와 *Department* 사이의 association은 다음과 같이 정의된다. 첫번째 줄은 이름이 Report인, 클래스 *Employee*와 *Department* 사이의 새로운 association을 생성한다. 클래스 이름인 "Employee"과 "Department"는 role name으로도 사용된다. *Employee*의 최대 multiplicity, 최소 multiplicity는 각각 0과 1000이며, *Department*의 최대 multiplicity, 최소 multiplicity는 각각 1과 1이다.

```
Relationship r1 = new Relationship("Report", Employee, 0, 1000, Department, 1, 1);
Relationship r2 = new Relationship("Manage", Department, "DeptInCharge", 0, 1,
                                                    Employee, "Manager", 0, 1);
```

Association은 재귀적(recursive)일 수 있다. 아래 예에서 OrgStructure는 재귀적 association이다. 두 Department 중 하나는 ChildUnit(하위 부서)의 역할을 하며, 다른 하나는 ParentUnit(상위 부서)의 역할을 한다.

```
Relationship r5 = new Relationship("OrgStructure", Department, "ChildUnit", 0, 1000,
                                   Department, "ParentUnit", 0, 1);
```

Association의 정의를 저장하기 위해서는 *DomainObject*의 다른 인스턴스와 마찬가지로 *Relationship*에 메소드 **AddInstance(PermanentObject e)**가 보내져야 한다.

```
Relationship.AddInstance(r1);
```

두 인스턴스를 정의된 association에 따라 연결하기 위해서는 *Relationship*에 클래스 메소드 **Connect(PermanentObject i1, PermanentObject i2, String rName2)** 가 보내져야 한다. 아래 예에서 첫번째 경우는 *Employee* 인스턴스 e1과 *Department* 인스턴스 d3를 연결하는데, 이때 d3의 role name은 "Department"가 된다. 두번째 경우는 *Department* 인스턴스 d1과 *Employee* 인스턴스 e5를 연결하는데, 이때 e5의 role name은 "Manager"가 된다. 세번째 경우는 *Department* 인스턴스 d1과 d2를 재귀적 관계로 연결하는데, 이때 d2의 Role Name은 "ParentUnit"가 된다. 이 때 *Relationship*은 참조 무결성 조건과 최대, 최소 multiplicity 조건을 검사한다.

```
Relationship.Connect(e1, d3, "Department");
Relationship.Connect(d1, e5, "Manager");
Relationship.Connect(d1, d2, "ParentUnit");
```

1.3. Generalization

Generalization은 서브타입을 수퍼타입의 서브클래스로 정의하고, **getSubclasses()**를 수퍼타입의 인스턴스 메소드로 정의함으로써 구현된다. 서브타입은 수퍼타입으로부터 모든 변수와 메소드를 상속한다. 이 때 서브타입에 고유한 변수는 서브클래스에서 정의되고, 수퍼타입과 다른 방식으로 작동하는 서브타입의 메소드는 수퍼타입의 메소드를 오버라이드(over-ride)하게 된다. 서브타입 사이의 관계가 partition인지 아닌지를 정의해주기 위해서는 수퍼타입의 클래스 변수 Partition의 값을 정의해주어야 한다. *Employee*와 서브타입 *HourlyEmployee*와 *SalariedEmployee*는 다음과 같이 정의된다.

```
public class Employee extends DomainObject {
    // define a method to answer Subclasses
    public Class[] getSubclasses() {
            Class[] subclasses =  HourlyEmployee.class, SalariedEmployee.class ;
              return subclasses;
    }
    // instance variables
    // class variables
    // constructor
    Employee() {
            Partition = true;
    }
    // instance methods
}

public class HourlyEmployee extends Employee {
    ...
}
```

```
public class SalariedEmployee extends Employee {
    ...
}
```

2. 사용자 인터페이스의 생성

한 클래스와 인스턴스들에 대한 사용자 인터페이스는 *DomainObjectList, DomainObjectAdd, DomainObjectDetail*을 상속함으로써 생성된다. 서브클래스의 인스턴스들은 수퍼클래스의 인터페이스를 통해 관리된다. 예를 들어 *HourlyEmployee*와 *SalariedEmployee* 인스턴스들은 *EmployeeList, EmployeeAdd, EmployeeDetail* 화면을 통해 관리될 수 있다.

2.1. 객체 입력 화면의 생성

Employee 인스턴스와 서브클래스 인스턴스를 추가하기 위한 기본 데이타 입력 윈도우(*EmployeeAdd*)는 다음과 같이 *DomainObjectAdd*를 상속함으로써 정의된다.(그림 4) 서브클래스를 가지는 경우에는 인스턴스를 삽입할 때, 어떤 타입의 인스턴스를 삽입할 것인지를 선택하는 화면을 생성한다(그림 5).

```
public class EmployeeAdd extends DomainObjectAdd {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
            PrintWriter out = new PrintWriter(res.getOutputStream(),true);
            this.InitContent("Employee", out, req, res);
}

    public void doPost(HttpServletRequest req, HttpServletResponse res)
```

```
        throws ServletException, IOException {
                PrintWriter out = new PrintWriter(res.getOutputStream(),true);
                this.Content("Employee", out, req, res);
        }
}
```

doGet(HttpServletRequest req, HttpServletResponse res) 메소드는 전송방식이 GET일 경우 호출된다. **doPost(HttpServletRequest req, HttpServletResponse res)** 메소드는 전송방식이 POST일 때 호출된다.

2.2. 객체 리스트 화면의 생성

모든 *Employee* 인스턴스들을 열람하는 기본 리스트 화면(*EmployeeList*)은 다음과 같이 *DomainObjectList*를 상속함으로써 정의된다. 객체 리스트 화면은 어떤 클래스와 서브클래스의 모든 인스턴스를 리스트하고, 객체 상세 화면으로 들어가는 링크를 제공한다(그림 6).

```
public class EmployeeList extends DomainObjectList {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
            PrintWriter out = new PrintWriter(res.getOutputStream(),true);
            this.Content("Employee", out, req, res);
    }
}
```

2.3. 객체 상세 화면의 생성

Employee 인스턴스를 수정, 삭제하기 위한 기본 상세 화면(*EmployeeDetail*)은

마찬가지로 *DomainObjectDetail*을 상속함으로써 각각 정의된다(그림 7).

```
public class EmployeeDetail extends DomainObjectDetail {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
            PrintWriter out = new PrintWriter(res.getOutputStream(),true);
            this.InitContent("Employee", out, req, res);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
            PrintWriter out = new PrintWriter(res.getOutputStream(),true);
            this.Content("Employee", out, req, res);
    }
}
```

V. 요약과 이후 연구 방향

본 연구에서는 객체지향 웹 애플리케이션 개발을 지원하기 위한 Java 클래스 집합을 제안하였다. Object System은 UML 방법론을 이용하여 객체를 정의, 유지, 접근할 수 있는 구조와 기능을 제공한다. Interface System은 Java Servlet을 이용하여 객체 관리 화면 개발을 지원한다. 본 연구는 SOODAS의 유용성을 재확인하고, 웹 애플리케이션 개발의 통합 환경으로 제공하였다.

그러나 본 연구의 시스템은 몇 가지 제한점을 가지고 있는데, 이것이 이후 연구 방향의 내용이 될 것이다. 첫째, 현재 시스템은 질의 언어를 제공하지 않

는다. OQL[Cattell, 1995]과 호환하거나 SOODAS와 같은 형식의 질의 언어가 개발되어야 한다. 둘째, 현재 시스템은 설계 패턴[Rho and March, 1997]을 지원하지 않는다. 설계 패턴은 메타클래스의 재활용성을 극대화할 수 있다. 다양한 갱신 의미(update semantics)를 지원하는 설계 패턴이 지원되어야 할 것이다.

VI. 참고 문헌

Bertino, E., Negri, M., Pelagatti, G., and Sbattella, L., Object-Oriented Query Languages: The Notion and the Issues, *IEEE Transactions on Knowledge and Data Engineering*, vol 4, no 3, June 1992, pp. 223-237.

Cattell, R. G. G., The Object Database Standard: ODMG-93, Release 1.2, Morgan Kaufmann Publishers, 1995.

Coad, P., *Object Models: Strategies, Patterns, and Applications*, Yourdon Press Prentice Hall, Englewood Cliffs, NJ, 1995.

Copeland, G. and Maier, D., Making Smalltalk a Database System, *Proceedings of SIGMOD Conference*, 1984, pp. 316-325.

Fowler, D. and Scott, K., UML Distilled, Addison-Wesley, 1997.

Javasoft, "Java Blend: Integrating Java Objects With Enterprise Data", *http://java.sun.com:80/ marketing/collateral/javablend.html*, 1998.

Javasoft, "Java Servlet API Whitepaper", *http://jserv.java.sun.com/products/java-server/documentation/ webserver1.1/servlets/api.html*, 1997.

Kilian, M. F., "Bridging the Gap Between OO and ER," *Proceedings of the 10th International Conference on Entity-Relationship Approach*, University of Michigan Press, Ann Arbor, October 23-25, 1991.

Lea, D., "Overview of the collections Package", *http://gee.cs.oswego.edu/dl/classes/collections/*, 1997.

Ling, T. W. and Teo, P. K., Toward Resolving Inadequacies in Object-Oriented Data Models, *Information and Software Technology*, (35, 5), May 1993, pp. 267-276.

March, S. T. and Rho, S., "Object Support for Entity Relationship Semantics," *Proceedings of Workshop on Information Technologies and Systems*, December 1996.

March, S. T. and Rho, S., "A Semantic Object-Oriented Data Access System", *Information Systems*, forthcoming, 1998.

Object Design, "PSE/PSE Pro for Java Tutorial", *http://www.odi.com/content/products/pse/doc/tutorial/index1.htm*, 1998

Orfali, R. and Harkey, D., *Client/Server Programming with Java and CORBA*, Wiley, 1997.

Rho, S. and March, S. T., "Object-Oriented Development: Patterns, Interfaces, and Update Semantics", *Seoul Journal of Business*, Vol. 3, No. 1, Fall 1997, pp. 37-63

Rumbaugh, J., "Relations as Semantic Constructs in an Object-Oriented Language", *Proceedings of OOPLSA Conference*, 1987.

Rumbaugh, J., Blaha, M., Premerlani, W., and Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Yourdon, E., *Object-Oriented Systems Design*, Yourdon Press-Prentice Hall, Englewood Cliffs, NJ, 1994.

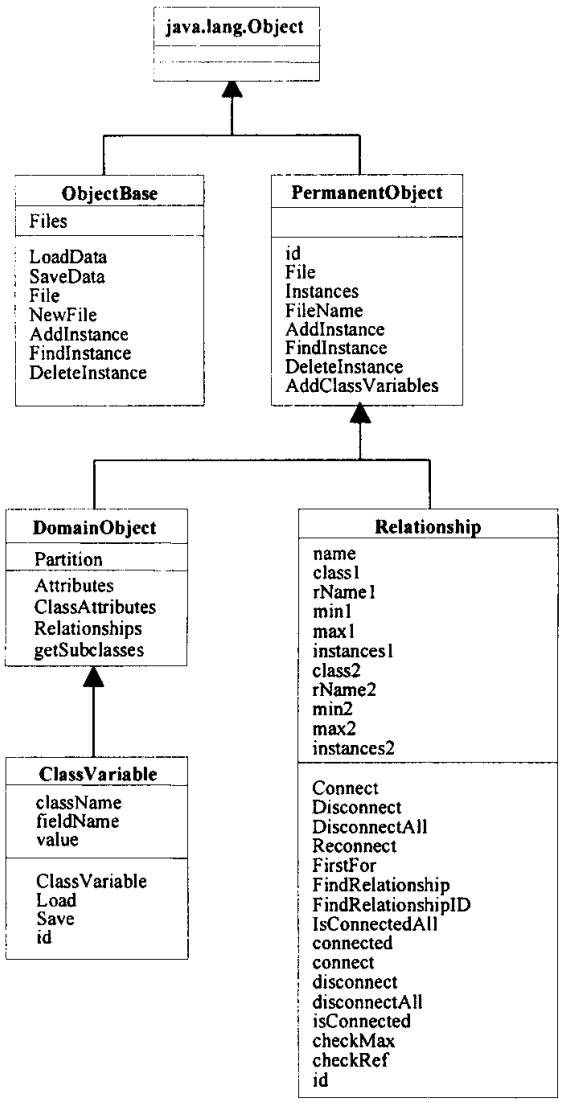

그림 1. Object System의 클래스 계층구조

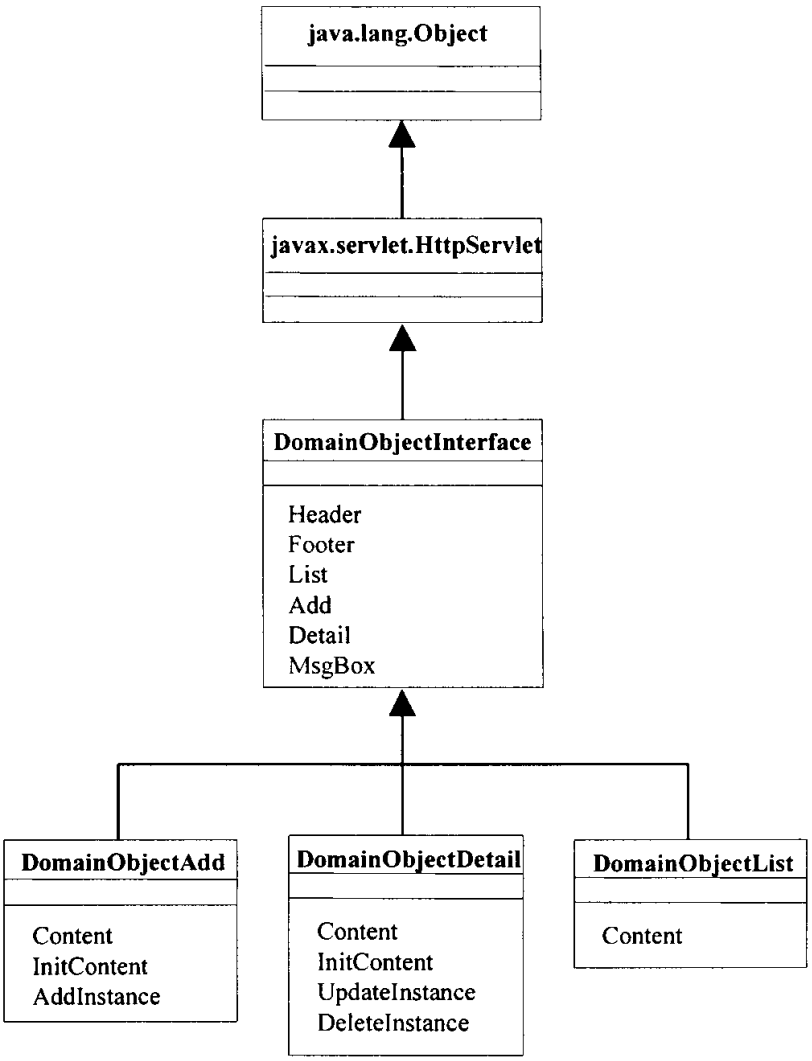

그림 2. Interface System의 클래스 계층구조

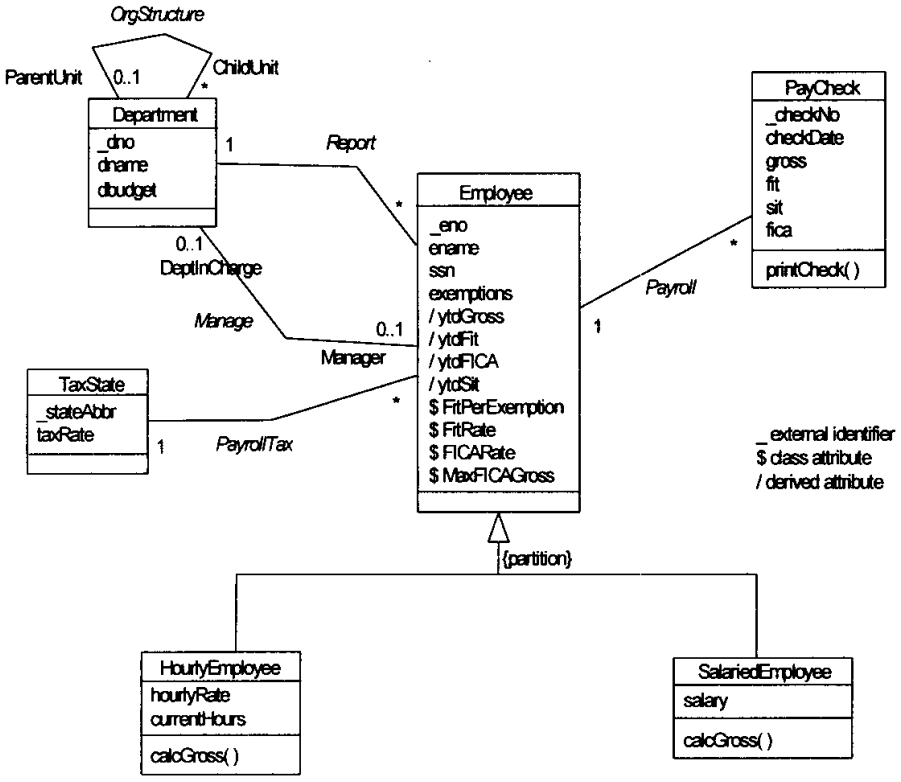
OrgStructure
ParentUnit
0..1
ChildUnit
*
Department
_dno
dname
dbudget
1
Report
*
Employee
_eno
ename
ssn
exemptions
/ ytdGross
/ ytdFit
/ ytdFICA
/ ytdSit
$ FitPerExemption
$ FitRate
$ FICARate
$ MaxFICAGross
PayCheck
_checkNo
checkDate
gross
fit
sit
fica
printCheck( )
*
Payroll
1
0..1
DeptInCharge
Manage
0..1
Manager
TaxState
_stateAbbr
taxRate
1
PayrollTax
*
_ external identifier
$ class attribute
/ derived attribute
{partition}
HourlyEmployee
hourlyRate
currentHours
calcGross( )
SalariedEmployee
salary
calcGross( )

그림 3. 객체 모델 사례

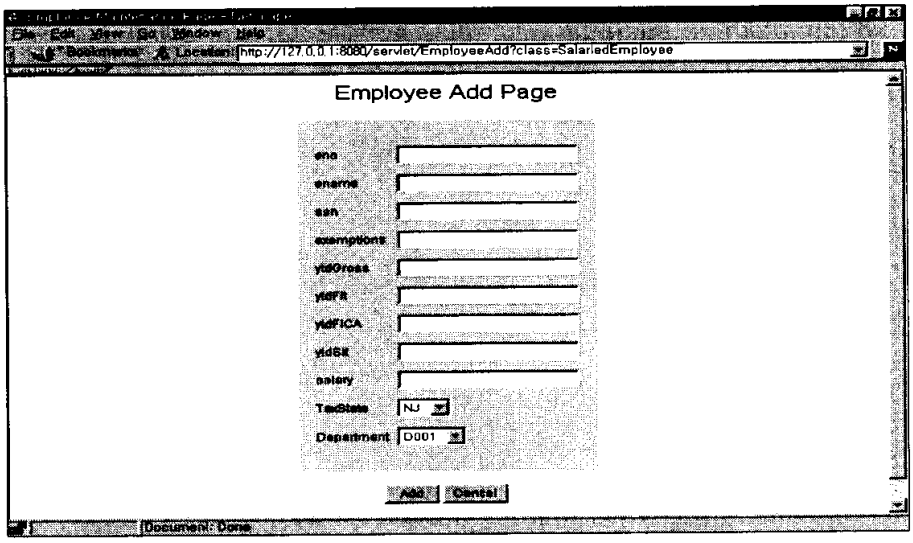

그림 4. 객체 입력 화면의 예

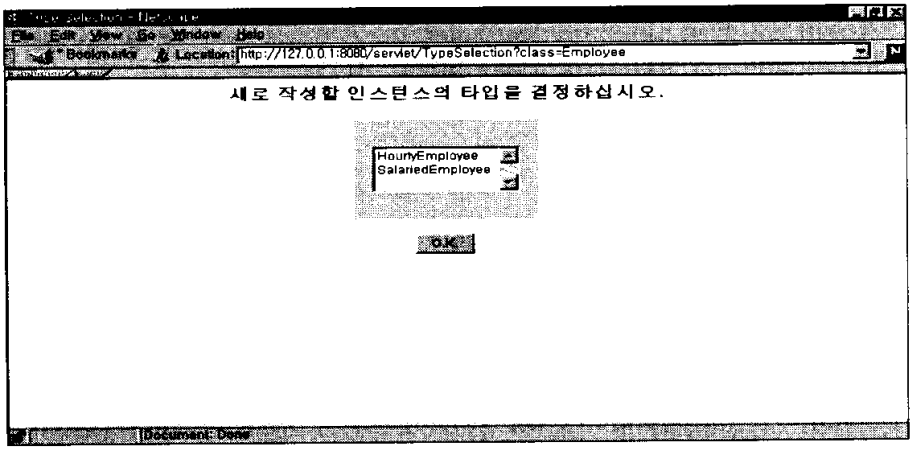

그림 5. 서브타입 선택 화면의 예

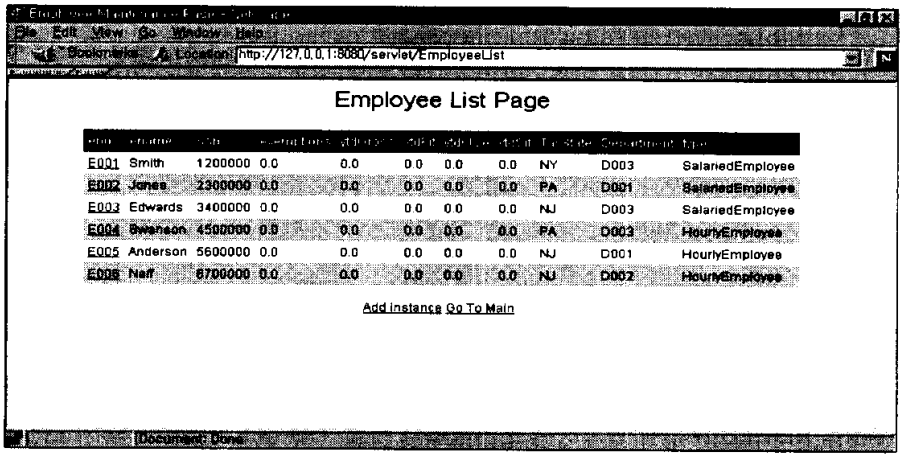

그림 6. 객체 리스트 화면의 예

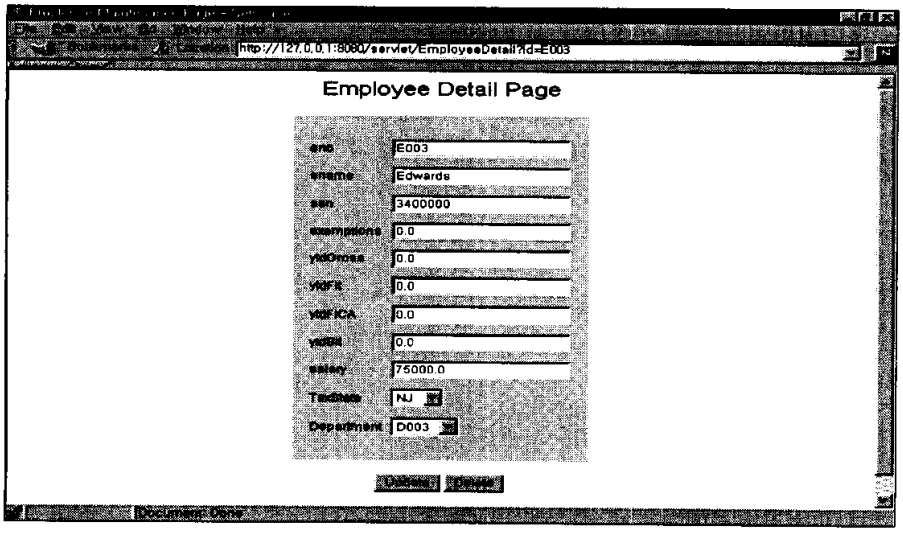

그림 7. 객체 상세 화면의 예