EER 모델을 이용한 Java Object 모델링과 Object 파서의 구현

김 경 식* · 김 창 화*

Java Object Modeling Using EER Model and the Implementation of Object Parser

Kyoung-Sik Kim* · Chang-Wha Kim*

Abstract

The modeling components in the object-oriented paradigm are based on the object, not the structured function or procedure. That is, in the past, when one wanted to solve problems, he would describe the solution procedure. However, the object-oriented paradigm includes the concepts that solve problems through interaction between objects.

The object-oriented model is constructed by describing the relationship between object to represent the real world. As in object-oriented model the relationships between objects increase, the control of objects caused by their insertions, deletions, and modifications comes to be very complex and difficult. Because the loss of the referential integrity happens and the object reusability is reduced. For these reasons, the necessity of the control of objects and the visualization of the relationships between them is required.

In order that we design a database necessary to implement Object Browser that has functionalities to visualize Java objects and to perform the query processing in Java object modeling, in this paper we show the processes for EER modeling on Java object and its transformation into relational database schema. In addition we implement Java Object Parser that parses Java object and inserts the parsed results into the implemented database.

* 강릉대학교 컴퓨터과학과

1. 서 론

컴퓨팅 환경이나 개발 환경을 급격한 변화로 이끌어내는 분산 네트웍 환경에서 객체지향 모델이 중요한 요소로 자리잡고 있다. 특히 클라이언트/서버 및 복잡한 소프트웨어 패키지가 객체라는 관점에서 개발되기 때문에 객체의 개념을 도입하고 있다. 이러한 객체지향 모델에서 필요한 객체들의 역할과 기능 분산 등을 하나의 모델로 추상화시킴으로 가치 있는 이전 설계 경험을 재사용할 수 있다는 장점을 가지고 있다. 그러나 실제 모델링을 위하여 객체들의 재사용에 관한 문제는 그렇게 쉬운 것만은 아니다[Rumbaugh et al. 1995; Cattell 1995].

캡슐화, 상속 그리고 다형성이라는 원리를 가지고 있는 객체지향 프로그래밍은 객체지향 모델을 구현하는데 도움을 주는 메커니즘을 제공하고 있다. 이러한 객체지향적인 특징을 가지고 있는 Java를 살펴보자. Java는 이식성과 보안성에 기반을 두고 개발이 되었지만, 이외에 가지고 있는 Java에 대한 특징은 단순성, 안전성, 이식가능성, 객체지향성, 강인성, 멀티 스레드, 아키텍처 중립성, 인터프리터, 높은 성능, 분산, 동적 등의 특성 및 장점들이 고려되었다[Naughton and Schildt 1997]. Java를 이용한 객체지향 모델링에 있어서 재사용을 위한 기초로 잘 설계된 Class와 상속(Inheritance)이 존재하고, 캡슐화로 클래스의 공용 인터페이스에 의존하는 코드를 쪼개지 않고 구현할 수 있으며, 다형성으로 명백하고, 분별력이 있고, 읽기 쉽고, 탄력성 있는 코드를 만들어 낼 수 있는 특징도 가지고 있다[Gosling et al. 1996].

프로시저 중심의 모델링은 프로그램을 더 복잡하게 만들 수 있다. 반면에 이렇게 증가하는 복잡성을 관리하기 위해 객체 지향 프로그래밍 방법을 고안하였다. 이러한 객체 지향 프로그래밍은 주요 소프트웨어 프로젝트의 라이프 싸이클을 수반하는 필수 불가결한 변경들을 가지고 있는 프로그램을 만들기 위한 강력한 본질적인 패러다임으로 발전하였다. 발전된 객체지향 모델에서는 잘 정의된 인터페이스를 갖는 객체 또는 클래스 단위로 문제를 분할하도록 함으로써 소프트웨어 개발에 많은 장점들이 이용되고 있다. 개발된 또는 개발하려고 하는 객체들의 관계가 많아짐에 따라 요구되는 모델의 복잡도는 더욱 커지게 된다. 복잡도가 커짐에 따라 객체 재사용 정도가 낮아지는 문제점이 발생된다[김루진, 김한우 1997; 김강태, 김정아 1997].

문제점을 해결하기 위한 것 중에 하나로 잘 정의된 인터페이스를 갖는 객체들을 분석하여 데이터베이스 체제로 전환한다면 재사용을 활용할 수 있는 개발 환경 즉, 분산 체제의 프로젝트 개발 및 분석에 좀 더 효율적인 개발 환경으로 전환할 수 있다. 데이터 베이스의 체제로 전환은 클래스, 속성 및 객체의 인스턴스가 객체지향 언어에서 나타나는 것과 동일한 방식으로 데이터 베이스 내부에서 표현되어야 하는 점을 고려하여 객체지향 모델의 필요성이 요구된다[EIL Lab 1994; Cattell et al. 1996]. 그러나, 현재의 개발 환경 즉, 순수하고 이론적인 관계형 모델도 어떠한 객체 유형, 데이터 구조 또는 분산 아키텍처를 수용할 수 있을 정도로 충분히 다용도로 사용되고 있다. 따라서 관계형 데이터 베이스로의 체제 변환도 그 가치가 있다고 할 수 있다.

본 연구는 Java 객체를 대상으로 객체의 구조와 객체들간의 관계(Relationship) 및 상속(Inheritance) 내용들을 시각적으로 나타내고[Bertino et al. 1991], 모델링에서 필요한 질의 처리 기능을 갖춘 Object Browser를 설계 및 구현하는데 필요한 데이터베이스를 설계하기 위해 Java 객체를 EER 모델로 표현하였고, 이를 관계형 DB로 변환하기 위하여 Object를 분석하는 Parser를 구현하였다. 이렇게 함으로써 객체의 재사용성을 높이고, 객체들의 유지보수를 수월하게 하며 더 나아가 객체의 분석 및 설계 품질을 높일 수 있는 방법을 유도하고자 한다.

이러한 배경 하에 본 논문의 2절에서는 EER 모델의 소개와 아울러 객체지향 패러다임에 맞춘 Java에서의 Object 표현방법에 대하여 설명하였고, 3절에서는 Java에서 표현되는 Object를 EER Diagram을 사용하여 모델링하는 방법과 관계형 데이터베이스 스키마로의 변환과정에 대하여 설명하였으며, 4절에서는 Object Browser의 구현에 필

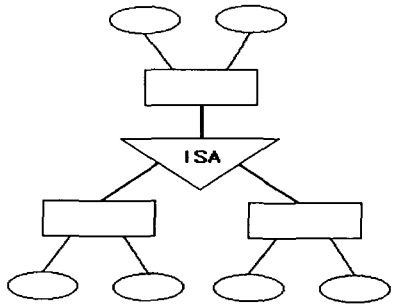
요한 Java Object의 DB 전환을 위한 파싱 알고리즘을 구현하였다. 마지막으로 5절에서는 결론 및 향후 연구 과제를 논의하였다.

2. EER 모델과 Java의 Object 표현 기법

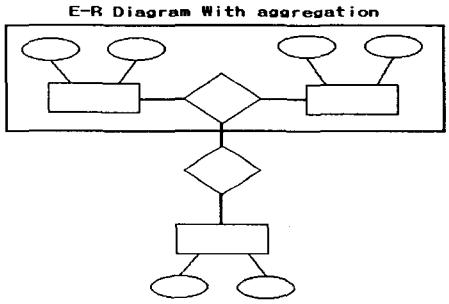
2.1 EER 모델

ER(Entity-Relationship) 모델은 엔티티 세트와 그들 사이에 존재하는 관계를 도형으로 표현한다. 엔티티 세트는 사각형으로 표현하고, 그것을 구성하는 속성(attribute)은 타원으로 표현하며, 엔티티 세트 사이의 관계를 마름모로 표현한다. 엔티티의 관계는 엔티티 세트에 따라 1:1 관계가 될 수도 있고, n:1 또는 1:n 관계가 될 수 있으며, n:m 관계가 될 수도 있다[Korth et al. 1991].

Java 언어는 객체지향 모델을 기반으로 하는데 객체지향 모델은 IS-A 관계를 나타내는 일반화(generalization)의 관계와 PART-OF 관계를 나타내는 집단화(aggregation)의 표현을 기본적으로 포함한다. 그러나, 본래의 ER 데이터 모델을 가지고 이러한 관계를 표현하기란 사실상 불가능하다. 따라서 EER(Extended ER) Diagram이라는 ER 데이터 모델을 확장한 모델이 소개되었다. EER Diagram은 원래의 ER Diagram의 기능에 객체지향 모델에서 가지는 객체들의 관계를 나타내는 IS-A 관계와 PART-OF 관계의 표현을 제공하는데 이러한 기능은 객체지향 모델을 관계형 데이터 모델로의 전환을 가능하게 한다. (그림 1)은 IS-A 관계를 나타내는 EER Diagram의 일반적 형태이고, (그림 2)는 PART-OF 관계를 나타내는 EER Diagram의 일반적 형태이다. (그림 1)에서 나타난 바와 같이 엔티티 세트 사이의 일반화 관계는 링크로 연결된 역삼각형을 이용하여 표현하며, 이 역삼각형에 연결된 상위 엔티티 세트는 이 역삼각형에 연결된 각 하위 엔티티 세트와 일반화 관계를 갖는다. (그림 2)에서 엔티티 세트들의 집단화 관계를 표현하기 위하여 엔티티 세트와 관계성들을 포함하는 사각형으로 표시되며, 여러 엔티티가 집합으로 있는 엔티티 세트의 형태로 포괄성 있게 표현한다.



(그림 1) 일반화 관계를 표현한 EER Diagram



(그림 2) 집단화 관계를 표현한 EER Diagram

2.2 Java Object의 표현 기법

객체지향 패러다임에서는 실세계를 객체들과 이들간의 통신으로 보는 관점을 취하고 있다. 여기에서 정의되어야 할 요소들은 Object, Class, Instance, Instance Variable, Method를 정의하여야 한다[Gosling et al. 1996].

Java에서 가지고 있는 Object는 체계적인 Class와 Interface로 구분할 수 있다. Class는 설계 이념인 단순한 단일 상속 체제로 표현된 Object 형태이고, Interface는 다중 상속을 구현하기 위한 추상 Object 형태를 가지고 있다. Java에서는 다중 상속의 복잡성을 배제하기 위하여 다중 상속을 지원하지 않는다. 하지만 다중 상속의 편리성을 제공하기 위하여 완전히 미완성된 채로 남겨진 Interface를 이용한다. 좀더 자세히 설명하면 In-

terface는 Interface안의 어떤 Method도 Method Body가 존재하지 않아 실행 부분이 존재하지 않는 Class를 위한 틀(template)로써 기능을 수행하는 추상 Class의 한 종류로서 하나의 Class가 둘 이상의 Class의 성질을 상속받는 것으로 부분적으로 집단화(aggregation)의 개념을 이용하여 다중 상속을 해결한다.

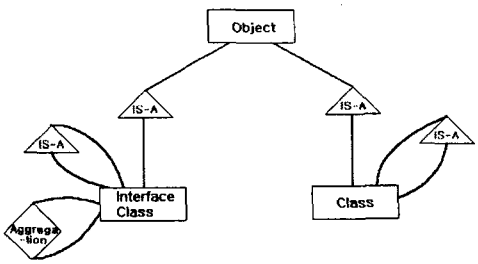
실제로 Interface는 Class를 위한 틀(template)이기 때문에 사용 가능한 Interface를 위하여 Java에서 Interface를 구현해 주어야 하는데 이 부분이 동적으로 운영되는 Implements 부분이다. 따라서 class는 Interface 안에 속해 있는 Method를 모두 자신의 코드 상에 정의해 놓고, 실행 부분을 추가해 줌으로써 Interface의 다중 상속의 개념을 활용할 수 있다[Gosling et al. 1996].

Class와 Interface로 표현된 Object는 객체지향 패러다임에 따라서 속성(attribute)과 메소드(method)로 정의하여야 한다.

3. EER 모델에 의한 Java Object 모델링

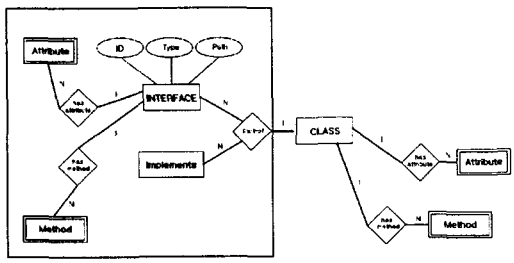
3.1 Java Object의 EER Diagram으로의 변환

Object는 Class와 Interface로 구분한다. Class는 Object와 Class와 일반화(IS-A) 관계를 가진다. Interface Class는 Object와 일반화(IS-A) 관계를 가지고, Interface Class와는 부분적으로 일반화(IS-A) 관계와 집단화(Aggregation) 관계를 가진다. 이것을 EER Diagram 형식으로 표현하면 (그림 3)과 같다.



(그림 3) Object의 EER Diagram

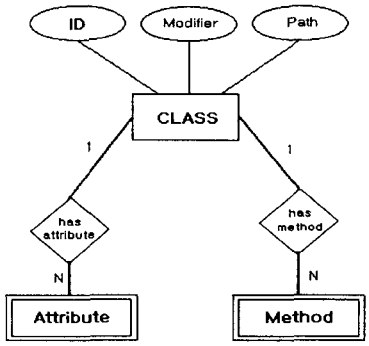
엔티티는 ID, Modifier, Path의 속성을 가진다. Interface 엔티티의 관계는 Attribute, Method 엔티티와 Has 관계를 가질 수 있다. 또한 Implements 엔티티와 집단화(Aggregation) 관계를 통하여 Class 엔티티에 대하여 Has 관계를 유지시킬 수 있고, Implements 엔티티는 Interface_ID와 Class_ID 형태의 결합형태로 사용이 되어지며, Implements 엔티티는 Class 엔티티와 Has 관계를 유지할 수 있다. 그러나, 대부분 Interface와 결합하여 Class 엔티티에서 동적으로 사용되는 관계를 유지한다. 이러한 관계를 EER Diagram에서 Interface 엔티티와 Implements 엔티티에 대하여 표현 불가능한 집단화(aggregation) 관계 등에 대하여 확장이 필요하다. 여기에서 확장된 ER Diagram 즉, EER Diagram을 사용하여 표현하였다. Class 엔티티는 ID, Modifier, 그리고 Path의 속성을 가진다. Interface 엔티티와 구분이 되는 것은 Class 엔티티는 Java에서 구현되는 단일상속에 대한 일반화(IS-A) 관계만을 가지고 있다는 점이다. 반면에 Interface는 다중상속을 위하여 Implements를 Interface 집단화(Aggregation) 관계로 표현할 수 있고, Interface는 여러 개의 Interface를 다중상속을 할 수 있으므로 Interface와 Class를 따로 분리하였다. 이러한 관계성을 가지고 EER Diagram으로 변경하면 (그림 4)와 같이 표현할 수 있다. (그림 4)의 EER Diagram에서 단일 사각형은 Strong Entity를 나타내며, 이중 사각형은 Weak Entity를 나타낸다.



(그림 4) Interface와 Implements의 EER Diagram

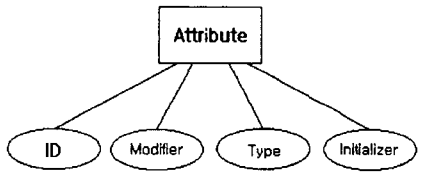
Class는 Attribute와 Method라는 종속 엔티티

(Weak Entity)와 Has 관계를 가진다. 실제 Java 컴파일러에서는 interface의 구현부분인 implements 메소드 부분이 동적으로 삽입되어 동작된다. Class 엔티티를 EER Diagram으로 표현하면 (그림 5)와 같이 나타낼 수 있다.



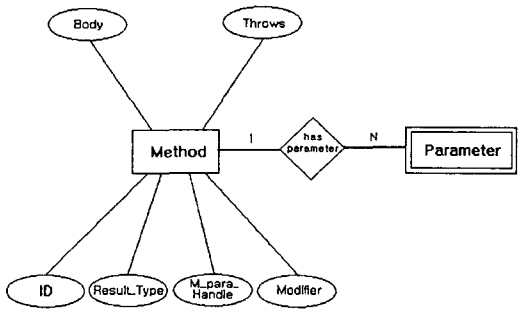
(그림 5) class의 EER Diagram

Attribute 엔티티는 ID, Modifier, Type, Initializer(초기값)들의 속성을 가지고 있으며, 이것을 EER Diagram 형태로 표현하면 (그림 6)과 같이 나타낼 수 있다.

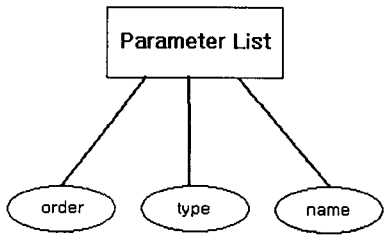


(그림 6) Attribute의 EER Diagram

Method 엔티티는 ID, Result_Type, Method_Parameter Handle, Modifier, Body, Throws라는 속성을 가지고 있고, Parameter라는 종속 엔티티와 Has 관계를 유지할 수 있으며, Parameter 엔티티는 Order, Type, Name의 속성을 가질 수 있다. Method_Para_Handle은 메소드의 Overloading을 표현하기 위한 속성으로 부여하는 메커니즘이 3.3절에서 논의되었다. Method 엔티티와 Parameter 엔티티를 EER Diagram으로 표현하면 (그림 7)과 (그림 8)과 같이 나타낼 수 있다.



(그림 7) Parameter의 EER Diagram



(그림 8) Method의 EER Diagram

3.2 EER Diagram의 데이터베이스 스키마로의 변환

객체지향 패러다임을 따르는 Java의 언어 스택을 보면 Class, Class Members, Field Declarations 그리고 Method Declarations의 형태로 구분한다[4]. 이를 기반으로 이 절에서는 3.1절의 EER Diagram을 기반으로 관계형 데이터 베이스에 적합한 스키마 구조로 변환한다. EER Diagram에서 각 엔티티를 테이블화 시킬 때 Interface는 IF, Class는 O, Implements는 IM, Attribute는 A, Method는 M 그리고 Parameter는 P를 사용하여 엔티티 별로 구별하였다.

우선 Class Table에 대한 구조를 보면 {O_ID, O_Modifier, O_Path, IS_A_OID}로 나타낼 수 있다. O_ID는 Object를 표현하는 Class Identifier이고, O_Modifier는 Class Modifier로써 public, final, abstract 중의 하나의 값을 가지며, O_Path는 Class가 포함되어 있는 File의 URL형식으로 분산 컴퓨팅 아키텍처(DCA)에서 효율성 있게 적용할 수 있도록 변환한 필드이고, IS_A_OID는 상

속이 이루어지는 Super Class의 ID이다. Java는 단일 상속만을 다루고 있어, 다중 상속의 혼란을 방지하였지만, 다중 상속의 유용성을 언급하기 위하여 Interface를 통하여 구현이 되어진다. 따라서, 다중 상속을 처리하기 위하여 Interface의 취급이 필요하다. Class의 구조를 Table로 표현하면 <표 1>과 같이 나타낼 수 있다.

〈표 1〉 Class Table

O_ID	O_Modifier	O_Path	IS_A_OID	IF_ID

Class Table에 Interface에 대한 정보를 포함하여 {O_ID, O_Modifier, O_Path, IS_A_OID, IF_ID}으로 확장할 수 있다. IF_ID는 Interface Identifier를 의미하고, Interface와 연결이 필요하다. 따라서, Interface를 처리하기 위한 Interface Table을 표현하면 {IF_ID, IF_Modifier, IF_Path, IF_Super}으로 나타낼 수 있다. IF_ID는 O_ID에 연계할 수 있는 형태로 Interface Identifier를 나타내고, IF_Modifier는 Interface Modifier로써 public과 abstract 중의 하나의 값을 가질 수 있다. IF_Path는 Interface가 포함이 되어 있는 File의 위치를 포함하는 URL 형식이고, IF_Super는 Interface의 Super Interface의 리스트 ID이다. Interface를 Table로 나타내면 <표 2>로 표현할 수 있다.

〈표 2〉 Interface Table

IF_ID	IF_Modifier	IF_Path	IF_Super

Class Table에서 사용되는 Primary Key는 {O_ID}이고, Interface Table에서 사용되는 Primary Key는 {IF_ID}를 사용한다.

Interface는 실제의 추상 형태로 제공이 되는 틀(template)의 형태로써, 구현이 되는 코드 부분이 필요하다. 따라서 Java에서는 implements라는 부분으로 처리된다. 이를 표현하기 위하여 Implements라는 Table이 더 필요하게 된다. Implements Table의 구조는 {O_ID, IF_ID, IM_Order_IF}로 표현된다. IM_Order_IF는 Implements에서 Interface의 상속에 관련된 순서를 가지는 필드이다. 이 Table에서 사용되는 Primary Key는 {O_ID, IF_ID}로 사용된다. 구현하기 위해 선언된 Interface의 이름과 실제 코드에서 사용되는 Class의 이름과 관계를 가지고 있다. Implements Table의 구조로 나타내면 <표 3>과 같이 표현할 수 있다.

〈표 3〉 Implements Table

O_ID	IF_ID	IM_Order_IF

Attribute에 대하여 RDB에서 적용되는 Table의 형태로 변환하면 {O_ID, A_Modifier, A_Type, A_ID, A_Initializer}로 정의할 수 있다. 여기서 A_ID는 Attribute Identifier를 나타내고 A_Initializer이라는 Attribute의 초기값으로 정의하여 구분되는 항목이다. A_Modifier는 A_ID가 가질 수 있는 필드의 특성을 가지며, public, protected, private의 엑세스 군이나 final, static, transient, volatile의 중에 하나를 가질 수 있다. A_Type은 데이터의 Type으로 시스템에서 제공하는 primary type이나 class type로 분류할 수 있다. 여기에서 사용되는 Primary Key는 {O_ID, A_ID}를 사용할 수 있다. O_ID는 Interface, Class의 이름에서 연결되어진 종속 엔티티와 관계를 가지므로 O_ID는 두 가지의 Object ID를 내포하고 있다. Attribute Table의 구조로 나타내면 <표 4>와 같이 표현할 수 있다.

〈표 4〉 Attribute Table

O_ID	A_Modifier	A_Type	A_ID	A_Initializer

Method를 Table로 변환하면 {O_ID, M_Modifier, M_Result_Type, M_ID, M_Parameter, M_Throws, M_Body}로 정의할 수 있다. M_Modifier에는 public, protected, private와 abstract, final, static, synchronized, native 중의 하나 이상을 가질 수 있

다. M_Result_Type은 Method에서 반환하는 값의 Type이고, M_ID는 Method Identifier를 나타내고, M_Parameter는 Method가 가질 수 있는 Parameter List이다. 또한 예외처리를 위한 M_Throws를 가지고, M_Body는 Method의 몸체부분을 갖는다.

여기에서 발생할 수 있는 문제는 Method들의 오버로딩(Overloading)이 발생하였을 때, Method를 구분하기 위한 문제가 발생할 수 있다. 따라서, 각 Method들을 구분하기 위하여 Method의 parameter의 형태에 따라 구분할 수 있는 핸들러가 필요하다. 이를 위해 16진수의 코드 두 자리의 수를 사용하였다. 여기의 정보를 반영하기 위하여 M_Para_Handle라는 필드부분이 필요로 하다. 그럼, Method Table을 변경하면 {O_ID, M_Modifier, M_Para_Handle, M_Result_Type, M_ID, M_Throws, M_Body}로 확장하여 표현할 수 있다. Parameter Handle를 주어 Method 오버로딩을 구분하는 방법에 대하여 3.3절에서 논의되었다.

이 Method의 Table에서 사용하는 Primary Key는 {O_ID, M_ID, M_Para_Handle}을 사용한다. O_ID는 Attribute에서 제시한 것과 같이 Interface, Class 필드와 연관되어지는 항목으로 종속적인 의미를 지니고 있다. 또한 Method의 함수 중복에 대하여 서로 구분을 하기 위하여 M_Para_Handle를 부여하여, 각각의 Method에 대하여 구분을 짓고, 각 Method의 Parameter에 대한 Table을 하나 따로 설정하여 Parameter와 관계를 갖는다. Method Table의 구조로 나타내면 <표 5>로 표현할 수 있다.

〈표 5〉 Method Table

O_ID	M_Modifier	M_Para_Handle	M_Result_Type	M_ID	M_Body	M_Throws

또한, Parameter에 대한 Table의 구조를 보면 {O_ID, M_ID, M_Para_Handle, P_Order, P_Type, P_Name}로 변환할 수 있다. P_Order는 Parameter의 순서를 유지하기 위해 사용하는 필드이고, P_Type은 Parameter의 Type을 나타내고, P_Name은 Parameter Name을 나타낸다. 이 테이블에서 사용하는 Primary Key는 {O_ID, M_ID, M_Para_Handle, P_Order}이다. O_ID는 Interface, Class를 나타내고, M_ID는 이 파라미터를 갖는 Method Identifier이고, 여기에 부여되는 M_Para_Handle은 함수의 중복 표현을 구분하기 위하여 제시한 핸들러이며, Method가 가지는 고유의 Handle 부여에 따라 각각의 Method가 갖는 Parameter를 따로 관리하게 된다. Parameter Table의 구조로 나타내면 <표 6>으로 표현할 수 있다.

〈표 6〉 Parameter Table

O_ID	M_ID	M_Para_Handle	P_Order	P_Type	P_Name

3.3 오버로딩을 해결하기 위한 파라미터 핸들 메커니즘

Class 안에서 Method들의 중복을 해결하기 위하여 M_Para_Handle에 16진수 두 자리를 부여하였다. 첫자의 코드는 함수의 종류를 나타내고, 나머지 한 자리의 숫자는 오버로딩되는 함수를 구분하기 위한 함수의 갯수를 의미한다. 오버로딩에서 함수를 각각 구분하기 위하여 다음의 예를 제시한다.

```
예) void a( int i );
    void a( int a, int b );
    void a();
    void b();
```

위 형태의 Method가 있다고 가정한다면, Method의 파서(Parser)에 의하여 void a(int i)라는 최초의 메소드에 함수의 종류를 나타내는 코드값으로 "0"과 함수의 중복의 개수를 검사하는 코드값 "0"이 부여된다. 따라서 void a(int i)의 M_Para_Handle은 "00"의 값을 가지게 된다.

void a(int a, int b)라는 메소드의 M_Para_Handle의 값은 첫 번째 코드의 값으로 기존의 함

수 중복이 발생하므로 처음 부여된 "0"을 부여하게 되고, 두 번째 코드값은 중복함수의 개수를 검사로 "1"를 부여한다. 따라서 이 메소드의 M_Para_Handle의 값은 "01"이 부여되게 된다. 이와 같은 방법으로 void a()는 "02"가 부여된다.

마지막 메소드인 void b()는 새로운 함수이므로 중복이 일어나지 않으므로 새로운 코드 값인 "1"을 부여하고, 두 번째 코드 값으로 함수의 개수를 검사하므로 "0"의 값이 부여되어 void b()에는 M_Para_Handle의 값은 "10"이 부여되게 된다.

이러한 메커니즘을 알고리즘 형태로 정리하면 다음의 *Algorithm-1*과 같다.

***Algorithm-1.* Parameter Handle의 제어 Algorithm**

```
// 구하고자 하는 Method_ID에 대한 Parameter Handle을 구하는
   루틴
// M_ID는 메소드 파싱이 되어진 파서 항목이다.
String M_Parameter_Handle(Method_Text_Vector, Method_ID )
{
   int  Same_Name = 0;

    // Method_Text_Vector.size는 Vector의 크기를 검사하는 Method
   for ( int i = 0 ; i < Method_Text_Vector.size() ; i++ )
   {
       M_ID를 출력;
       if( M_ID == Method_ID )  // Overloading인 경우의 검사
       {
          Same_Name ++;
          Method_Handle --;
          Method_ID에 대한 Same_Name의 값과 Method_
              Handle의 값을 출력
       };
   }

   파라미터 테이블을 위하여 Same_Name과 Method_Handle의
       값을 String으로 변환한다.;
   Method_Handle ++;
   return 변화되어진 String값을 반환한다.
   }
```

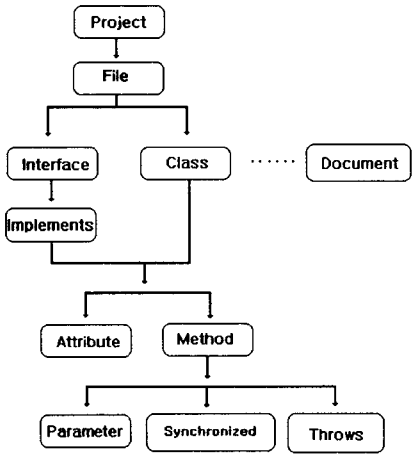
4. Table 생성을 위한 파서(Parser) 구현

4.1 파싱 알고리즘

본 연구에서 구현한 파서(Parser)는 Java Object들을 파싱(Parsing)하여 앞에서 도출한 관계형 DB 테이블로 저장하는 기능을 수행한다. 즉, Java Object의 구조와 Object들간의 관련성 정보를 관계형 데이터베이스로 변환 및 저장하는 역할을 담당한다. 이렇게 변환된 관계형 데이터베이스는 궁극적으로 현재 진행 중인 연구를 통하여 분산 환경에서의 Java Object Browser에서 이용될 것이다.

Java를 이용한 Project의 개발 과정을 보면, Java 코드로 구성된 Java Project의 파일을 분석하여, 각 분산되어 있는 파일들의 URL를 추적하여 수집한 후 각 파일에 대한 파싱 작업을 수행한다. 이때 각 파일은 Interface, Class의 형태로 큰 단위의 파싱이 이루어져 각 파싱의 결과를 Vector로 할당한다. 각 할당되어진 Vector에 의하여 Attribute와 Method Vector로 파싱되고, Method Vector를 이용하여 Parameter, Synchronized 그리고 Throws Vector를 생성하게된다. 이때 Comment에 대한 Vector도 생성이 된다.

이 과정을 3장에서 제시한 EER Diagram 및 Table의 전환에 필요한 Java 코드의 파싱 과정은 (그림 9)와 같다.



(그림 9) Java Object의 Parsing 절차

(그림 9)에서 제시된 메커니즘은 우선 Project를 로딩하는 과정을 포함하는데 이 과정을 보면 다음의 Routine-1과 같다.

Routine-1. Java Project 로딩 루틴

```
class Project_Open
{
    String   project_name;
    int      File_Number;

    void Project_File_Loading()
    {
        Text_Vector = File_URL_pack();
        File_Number++;
    }
};
```

위의 Routine-1 과정대로 작성 중이거나 작성한 프로젝트를 로딩하면, 호출된 프로젝트를 구성하는 URL 형식으로 분산된 파일들을 모두 집계하여 Text_Vector 형태로 저장한다.

저장된 URL 형식의 파일 이름들은 File_Open 루틴에 의하여 File_Number에 할당된 파일의 개수만큼 Text_Vector에 저장된 파일을 class_chk()에 의하여 Interface와 Class Vector로 변환한다.

파일의 수만큼 Class를 체크하는 루틴을 제시하면 다음의 Routine-2와 같다.

Routine-2. Java Source Program 분석 루틴

```
void File_Check()
{
    while( File_Number < 0 )
    {
        File_Open( Text_Vector );
        Class_Check();
    }
};
```

위의 루틴에서 Class_Check()는 File에서 Class와 Interface 단위를 검사하여 Class_Number라는 개수를 검사하고, String을 통하여 Class나 Interface 단위의 Text_Vector에 할당시킨다. 이렇게 클래스 단위로 저장이 되어있는 Class_Text_Vector나 Interface_Text_Vector에서 하나의 Keyword로 각각을 분류할 수 있게 되어 결국, 각각의 Class_Text_Vector나 Interface_Text_Vector에는 하나의 Class나 Interface 단위로 할당하게 된다. Class_Number라는 Class나 Interface의 개수를 저장하는 변수를 통한 루틴이 다음의 Routine-3에서 제시된다.

Routine-3. Java Class & Interface 검사 루틴

```
class Class_Check()
{
   int  Class_Number;

   void class_check()
   {
      while( Class_Number < 0 )
      {
         switch( token )
         {
            case  "class" :
              class_parser(Class_Text_Vector나
                    Interface_Text_Vector );
              break;
            case "interface" :
              interface_parser(Class_Text_Vector나
                    Interface_Text_Vector );
              break;
            case "implements" :
              implements_parser(Class_Text_Vector나
                    Interface_Text_Vector );
              break;
         }
         Class_Number --;
      } // End of while
   };
}; // End of Class
```

Routine-3 루틴의 implements_parser();에서는 각각의 테이블 항목별로 파싱하여 JDBC[9]를 통하여 데이터 베이스와 연결하여 SQL 문장을 보냄으로 SQL Table에 삽입한다. 또한, class_parser(); 와 interface_parser();에서는 Attribute와 Method는 Text_Vector로부터 각각의 분류 조건에 의하여 파싱하는 루틴을 가지고 있다. 이 메커니즘이 다음의 Routine-4에서 제시된다.

Routine-4. Java Attribute 검사 루틴

```
Attribute_Parser( Class_Text_Vector나 Interface_
Text_Vector )
{
   Class_Text_Vector이나 Interface_Text_Vector를
     String으로 처리하고 "\n"을 삭제한다 ;
   "class"나 "interface"라는 키워드 추출한다 ;
   첫 번째 "{" 까지를 각각의 Attribute_Text_
     Vector를 생성한다 ;
   JDBC에 연결하여 Table에 삽입한다
};
```

다음의 Routine-5는 Java method를 분석하여 해당 테이블로 그 정보를 삽입하는 기능을 수행한다.

Routine-5. Java Method 검사 루틴

```
Method_Parser( Class_Text_Vector나 Interface_Text_Vector )
{
   Class_Text_Vector이나 Interface_Text_Vector를 String으로
   처리하고 "\n"를 삭제한다 ;
   "{"와 "}"의 개수를 이용하여 Method_Text_ Vector에 할당하며
   이때 다음을 적용한다

      { 이면 +1
      } 이면 -1
      { 일 때 2이면 Method의 시작 표시
      } 일 때 1이면 Method의 끝 표시 ;

   JDBC로 연결하여 Parameter_Text_Vector에 부분을 테이블로
   삽입한다 ;
};
```

한편, 여기에서 생성된 Method_Text_Vector는 Parameter를 파싱하는 루틴으로 전달된다.

Routine-6. Java Parameter 검사 루틴

```
Parameter_Parser( Method_Text_Vector )
{
  Method를 구분한 항목에서 "("와 ")"의 사이에 있는
  String 형태를
  ","를 기준으로 Parameter_Text_Vector에 삽입한다.
  JDBC로 연결하여 Parameter_Text_Vector에 부분을
    테이블로 삽입한다.
};
```

추가로, 파서의 구현에 각 Vector 별로 파싱된 항목에서 "//"와 "/*" 그리고 "*/"에 의한 Comment 처리 루틴을 별도로 두어 Comment도 Table에 추가되도록 하여 각 객체들에 대한 설명을 관리할 수 있도록 하였다.

4.2 파싱 알고리즘의 구현 및 적용 예

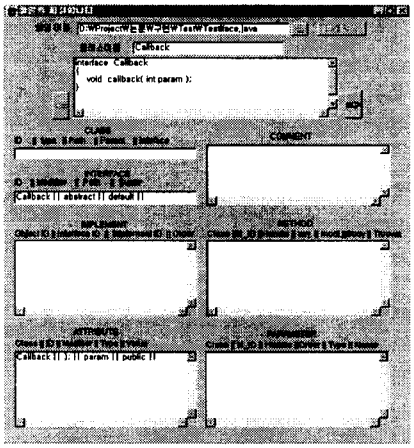
4.1절의 파싱 알고리즘은 Windows NT 환경 하에서 Java 언어를 이용하여 실제로 구현되었다. 이 시스템은 데이터베이스관리시스템으로서 SQL Server를 이용하였고, Java와 SQL Server와의 연동을 위해 JDBC를 이용하였다. 이 시스템을 적용한 예를 보이기 위해 Interface를 사용하는 다음 Java 예제 프로그램을 이용하였다.

예제 : 파싱되어 DB로 저장될 Java 예제 프로그램

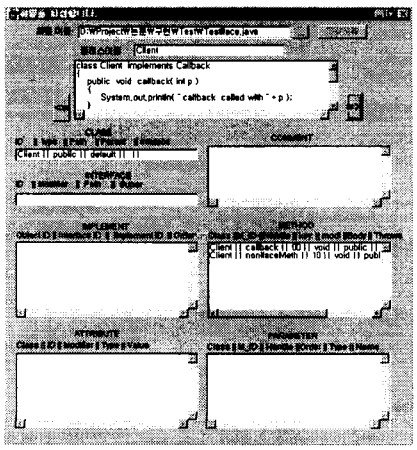
```
// class declaration
interface  Callback
{
    void  callback( int param );
}

// class implements
class Client  implements Callback
{
    public  void  callback( int p )
    {
      System.out.println( " callback  called
      with " + p );
    }

    void  nonIfaceMeth()
    {
      System.out.println("Classes that implement
      interface " +
            "may also define other members, too.");
      }
    }

    class TestIface
    {
      public static void main( String args[] )
      {
          Callback  c = new  Client();
```

```
        c.callback( 42 );
    }
}
```

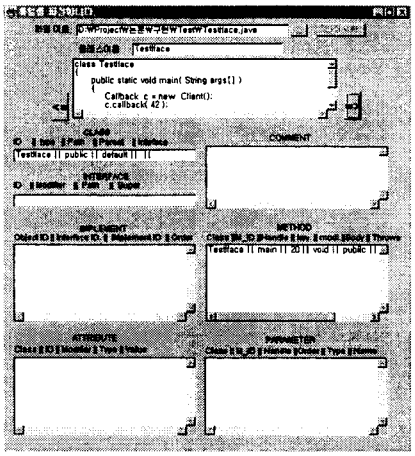
위의 예제 프로그램은 한 개의 Callback이라는 Interface와 Client와 TestIface라는 2 개의 Class로 구성이 되어진 Java 예제 프로그램이다. 이 예제 프로그램을 파싱하여 그 결과를 출력하는 사용자 인터페이스가 (그림 10), (그림 11), (그림 12)에서 제시된다. (그림 10)은 Callback이라는 Interface에 대한 내용을 보여주고 있고, (그림 11)은 Client라는 Class의 내용을 나타내며, (그림 12)는 TestIface라는 Class를 나타낸 결과이다.



(그림 10) 파싱 프로그램 동작 예 1



(그림 11) 파싱 프로그램의 동작 예 2



(그림 12) 파싱 프로그램 동작 예 3

구현된 파싱 프로그램에 의하여 나타난 결과를 DB 테이블 형태로 나타내면 다음과 같다. 이 테이블은 JDBC를 이용하여 SQL Server에 기록된다.

▶ Interface Table

IF_ID	IF_Modifier	IF_Path	IF_Super
Callback	public	default	NONE

▶ Class Table

O_ID	O_Modifier	O_Path	IS_A_OID	IF_ID
Client	public	default	NULL	Callback
TestIface	public	defalut	NULL	NULL

▶ Implements Table

O_ID	IF_ID	IM_Order_IF
Client	Callback	0

▶ Attribute Table

O_ID	A_Modifier	A_Type	A_ID	A_Initializer
NONE	NONE	NONE	NONE	NONE

▶ Method Table

O_ID	M_Modifier	M_Para_Handle	M_Result_Type	M_ID	M_Body	M_Throws
Callback	public	00	void	callback	NULL	NONE
Client	public	00	void	callback	link	NONE
Client	public	00	void	nonIfaceMeth	link	NONE
TestIface	public	00	void	main	link	NONE

▶ Parameter Table

O_ID	M_ID	M_Para_Handle	P_Order	P_Type	P_Name
Callback	callback	00	0	int	param
Client	callback	00	0	int	p
Client	nonIfaceMeth	00	0	NULL	NULL
TestIface	main	00	0	String	args[]

5. 결 론

본 논문은 객체 지향적 특징을 가지고 있는 Java 객체의 구조와 객체들간의 관계 및 상속 정보들을 나타내기 위해 필요한 데이터를 EER diagram을 이용하여 모델링하고 이 diagram을 중심으로 관계형 데이터베이스 테이블들을 유추하였다. 또한, 본 논문은 Java프로그램의 object들을 분석하여 유추된 테이블들로 변환 및 저장하는 파싱 알고리즘을 제안하고 이 알고리즘을 구현한 시스템을 개발하였다. 본 논문에서 제안한 EER 다이어그램과 테이블들, 그리고 파싱 알고리즘은 분산 환경 하에서 Java 객체의 구조 및 객체들간의 관계를 시각화시키고 상속성에 따른 추적성(traceability)을 지원하는 Object 브라우저의 개발에 그대로 도입될 수 있다. 이 Object browser는 여러 개발자들의 프로그램의 동시개발 환경 하에서 프로그램의 개발 및 분석을 용이하게 하며 객체들의 재사용을 지원할 수 있다.

객체들의 분석은 객체지향 설계에 만족하는 객체의 인식과 정의, 클래스의 구성, 클래스들간의 관계를 파악할 수 있고, 클래스 계층 구조 구성이 쉬워지며, 재사용 가능한 클래스 라이브러리와 응용 프레임워크의 구축을 쉽게 할 수 있다. 또한 데이터 베이스 체제의 관리를 이용한 객체들의 관리 및 전환은 개발 환경의 개선뿐만 아니라 개발비용의 절감에 크게 기여할 수 있다.

본 논문에서 구현된 데이터 베이스를 기반으로 네트웍 환경에서 Java Object 구조 및 관계성 분석, 그리고 재사용성을 지원하기 위하여 이들 정보를 네트웍 상의 사용자들에게 시각적인 형태로 지원하는 Object 브라우저 시스템을 인터넷상에서 현재 개발 중에 있다. 이 Object 브라우저 시스템은 클라이언트-서버 모델 아키텍처를 기반으로 한다. 이 시스템에서 클라이언트는 Java 프로그램을 파싱하는 기능을 제공하며, 네트웍 상에서 여러 개발자에 의해 동시 개발된 Java 프로그램에서 객체의 구조와 상속관계를 비롯한 관계성을 시각화시키고 사용자에게 이들 관계에 따른 추적성을 지원하는 GUI를 제공한다. 한편, Object 브라우저 시스템의 서버는 클라이언트의 파싱 정보를 기록 및 저장하고, 클라이언트가 사용자에게 시각적인 정보를 제공하는데 필요한 데이터를 클라이언트의 요청에 따라 전송하는 기능을 갖는다. 이 시스템은 차후 Java 객체의 삽입과 변경, 그리고 삭제 기능을 갖추어 객체를 관리할 수 있도록 하고 Java 프로그램을 이 시스템 상에서 직접 개발하고 유지보수하는 기능을 갖추도록 확장할 예정이다.

참 고 문 헌

[1] EIL Lab, *A Users' Guide for Oak*, University of Toronto, September 1994.

[2] Bertino, Elisa and Lorenzo Martino, *Object-Oriented Database Systems(Concepts and Architectures)*, Addison-Wesley Publishing Company, 1991.

[3] Korth, Henry F. and Abraham Silberschats, *DataBase System Concepts*, Second Edition, McGraw-Hill Inc., 1991.

[4] Gosling, James Bill Joy, and Guy Steele, *The Java Language Specification Edition 1.0*, 1996.

[5] Rumbaugh, James Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall International Edition, 1995 .

[6] Arnold, Ken James Gosling, *The Java Programming Language, Second Edition*, Addison Wesley, 1998.

[7] Naughton, Patrick and Herbert Schildt, *JAVA : The Complete Reference*, McGraw-Hill, 1997.

[8] Cattell, R.G.G. *Object Data Management (Object-Oriented and Extended Relational Database-Systems, Revised Edition*, Addison-Wesley Publishing Company, 1995.

[9] Cattell, Rick Graham Hamilton, and Maydene Fisher, *JDBC Database Access with Java : A Tutorial and Annotated Reference*, Addison Wesley, 1997.

[10] _____, *O_2 Technology*, Technical Overview of the O_2 system, May 1996.

[11] Chan, Patrick and Rosanna Lee, *The Java Class Libraries : An Annotated Reference*, Addison Wesley, 1997.

[12] Chan, Patrick Rosanna Lee, and Doug Kramer, *The Java Class Libraries : Volume 1*, Addison Wesley, 1994.

[13] Chan, Patrick Rosanna Lee, and Doug Kramer, *The Java Class Libraries : Volume 2, Second Edition*, Addison Wesley, 1995.

[14] 김루진, 김한우, "MSQL 접근을 위한 JAVA 클래스 개발", *'97년 가을 학술발표 논문집 (I), 한국정보과학회*, 1997.

[15] 김강태, 김정아, "자바 프로그램 개발을 지원하는 프레임워크 구축", *'97년 가을 학술발표 논문집(I), 한국정보과학회*, 1997.

■ 저자소개

김 경 식

강릉대학교 전자계산학과를 졸업하고, 강릉대학교 교육대학원 전산교육석사를 취득하였다. 현재 강릉문성고등학교에서 전산담당 교사로 재직하고 있으며, 주 관심분야는 객체지향 데이터베이스, 관계형 데이터베이스 분야이다.

김 창 화

고려대학교 수학교육과를 졸업하고, 고려대학교 일반대학원 전산학전공 석사, 그리고 동 대학원에서 박사 학위를 취득하였으며, 1994년부터 1년간 카나다 토론토 대학 EIL 연구소에서 Post-Doc.과정을 거쳤다. 1989년부터 강릉대학교 컴퓨터과학과에서 전임강사, 조교수를 거쳐 현재 부교수로 재직하고 있다. 현재 정보처리학회지 편집위원이며, 주 관심분야는 인터넷 정보검색, 멀티미디어데이터베이스, 지식관리시스템, 분산 시스템 등이다.