

# XMI기반 객체지향 메타모델 생성

이 돈 양<sup>†</sup> · 송 영 재<sup>††</sup>

## 요 약

일반적으로 객체지향 모델링으로 UML을 이용한 설계방법이 많이 사용되고 있다. 그러나 UML을 이용한 메타데이터의 생성은 서로 다른 환경에서 언어와 이에 관련된 세부적인 기능들에 대한 표현의 차이점으로 쉽지가 않다. 이를 해결하기 위해 본 논문에서는 패턴 및 클래스에 대한 부분을 정형화, 표준화하기 위한 방법으로 XML Metadata Interchange Format(XMI)를 이용하였다. 그리고 메타모델의 설계를 위해서 XMI 메타모델 중 사용의 빈도수가 많은 4개의 엘리먼트(element)만을 선택하여 메타데이터를 생성하였다. 생성된 메타데이터를 저장하기 위해서 DB를 이용한 저장소를 설계하였으며, 패턴 및 각 클래스 구성에 대한 정보를 추가하고 질의어(query)를 이용하여 메타데이터의 재사용 및 확장이 용이하도록 하였다.

## Generation of Object-Oriented Metamodel based on XMI

Don-Yang Lee<sup>†</sup> · Young-Jae Song<sup>††</sup>

## ABSTRACT

Usually, design method to use UML by Object-Oriented Modelling is used much. But, generation of Metadata that use UML is not easy by difference of expression about detailed functions that is involved language and this in environment that differ. In this paper that solution method use XML Metadata Interchange Format(XMI) for standardization and normalization of Pattern and Class. And, for design of Metamodel select frequency A many 4 element of use among XMI Metamodel and create Metadata. Design DB repository for created Metadata storing and add pattern and information about each class composition and use query and did so that reusability and extension of Metadata may be easy.

키워드 : XML, XMI, UML, 메타모델(Metamodel), 메타데이터(Metadata), 저장소(Repository)

### 1. 서 론

소프트웨어의 설계는 소프트웨어에 대한 생명주기와 매우 중요한 밀접한 관계를 가지고 있다[1-3]. 일반적으로 소프트웨어설계에서는 기술적인 실현가능성과 정확성, 그리고 사용자의 요구사항들을 모두 포함한다[4]. 또한 정교하고 복잡한 부분을 소프트웨어로 생산함에 따라서 소프트웨어 재사용도 크게 향상되었다. 소프트웨어 설계와 관련하여 최근에는 객체지향모델링(Object-Oriented Modelling)[5]을 이용한 방법이 많이 사용되고 있으며, 이는 사용자들의 요구사항에 대한 관점에서 최신의 기술을 적용하고 있다.

특히, OMG의 UML은 객체지향모델링에 대해서 표준화된 언어에 지원이 가능하여 널리 사용되고 있다[6]. 이전에 주로 사용했던 클래스정보에 대한 속성의 추출 및 분류[6]에서는 추출된 클래스의 기능정보가 단지 원시코드의 코멘트에서 추출되었기 때문에 클래스에 대한 정확한 기능 및 용도에 대한 문서가 부족하여 실제로 이용자가 최적의

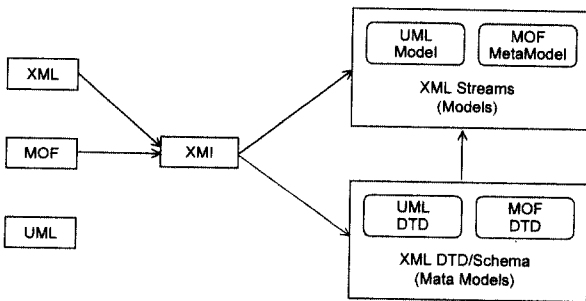
부분을 추출하기가 어려웠다. 이러한 것들을 향상시키기 위하여 본 논문에서는 객체에 대한 클래스뿐만 아니라 패턴 모델의 설계에서도 객체지향모델링 방법을 이용하여 메타모델과 메타데이터를 생성하였다. 그리고 패턴 및 클래스에 대한 부분을 정형화, 표준화하기 위한 방법으로 프로그래머들과 다른 사용자들이 메타데이터(metadata)에 관하여 정보를 교환하기 위한 표준방식으로 제안된 XML Metadata Interchange Format(XMI)를 이용하였다. 이것의 연구로 클래스를 각각의 디자인패턴에 적용하고 분류하여 다시 XMI 메타모델로 변환하여 저장소에 저장하는 방법이 제안되었으나[11], 클래스의 상속관계나 클래스합성에 대한 부분은 제외되었다. 본 연구에서는 이에 대해서 UML 모델링을 XMI Parser를 통하여 XMI 메타데이터로 생성하고 DB에 저장하였으며, 메타데이터의 표현이나 이를 이용하는 사용자들이 데이터를 활용할 수 있도록 인식테이블을 작성하였다. 그 결과 메타데이터를 통한 공통클래스의 합성뿐만 아니라 사용된 패턴모델의 기능과 클래스의 상속관계를 쉽게 파악할 수 있어 소프트웨어 재사용이나 확장을 원활하게 할 수 있도록 하였고, 사례연구에서는 시스템복잡

† 준 회원 : 경인여자대학교 전산정보학 교수  
 †† 총신회원 : 경희대학교 전자계산공학과 교수  
 논문접수 : 2003년 11월 7일, 심사완료 : 2004년 2월 17일

도가 35.8%정도 감소되어 소프트웨어설계의 성능향상을 가질 수 있었다.

## 2. 관련 연구

UML을 이용한 시스템설계에서는 사용되는 언어뿐만 아니라 이에 종속된 세부적인 것들의 차이들로 인하여 모델의 변환에 대한 서로 다른 표현의 개념들을 가질 수 있다 [7,8]. 본 논문에서는 이런 문제점들을 해결하기 위한 방법으로 OMG(Object Management Group)[10]의 XMI(XML Metadata Interchange format)[9]을 이용한 방법을 도입하고자 하였다. XMI는 서로 다른 환경에 분산되어 저장되어 있는 메타데이터와 UML 모델링에서 사용되고 있는 데이터나 메타데이터와의 원활한 변환 및 교환을 목적으로 표준화된 모델링 방법이다.



(그림 1) XMI와 XMI DTD/Schema

XMI 엘리먼트는 XML.header, XML.content 등 21개로 구성이 되어있다[9]. 그러나 본 논문에서는 UML의 클래스 모델에 대한 XMI 메타모델로의 변환에는 (그림 2)에서와 같이 XML의 Top Level에 해당하는 <XML.header>, <XML.content>, <XML.difference>, <XML.extensions>의 4개의 엘리먼트만을 포함시켜서 모델을 구성하였다.

<XML.header>는 XML 엘리먼트에서 model, metamodel, metametamodel을 정의하고 있으며, 이는 메타데이터를 인식하기 위해 사용되고, 메타데이터의 변환에 관한 다양한 정보를 포함하고 있다.

```
<!ELEMENT XML.header <XML.documentation?,
    XMI.model*,
    XMI.metamodel*,
    XMI.metametamodel*,
    XMI.import*>
```

<XML.content>는 메타데이터의 변환에 대한 실질적인 것들을 포함하고 있으며, 모델정보나 메타모델의 정보로 표현된다.

```
<!ELEMENT XMI.content ANY>
```

<XMI.difference>는 기본 데이터에 대한 차이를 표현하는 XML의 엘리먼트이다. 사용자는 이것을 XML파일의 부분이나 XMI.difference Section의 분리에 사용된다. 이 엘리먼트안에서 XLinks나 XPointers등의 속성을 가질 수 있다.

```
<!ELEMENT XMI.difference (XMI.difference | XMI.add |
XMI.delete | XMI.replace)*>
<!ATTLIST XMI.difference
    %XMI.element.att ;
    %XMI.link.att ;
>
```

<XMI.extensions>는 메타모델의 확장에 사용되는 메타데이터를 포함하는 XML 엘리먼트를 가지고 있다. 이것은 메타데이터에 관련된 정보들을 표현하는데 사용된다. 여기서 xmi.extender 속성은 확장에 대한 요구사항들을 가지고 있다.

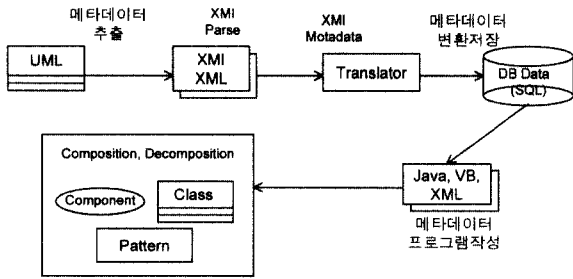
```
<!ELEMENT XML.extensions ANY>
<!ATTLIST XML.extensions
    xmi.extender CDDATA #REQUIRED
>
```

```
File = "CommonClass.xml" Namespace = "CommonClass" :
< XMI version = "1.1" XMLNS : uml = "org.omg/UML1.3" >
< XML.header >
// 메타모델의 종류와 XMI 정보기술
</ XML.header >
< XML.content >
// 실제 디자인패턴의 메타모델정보를 갖는 XMI 정보기술
</ XML.content >
< XMI.Difference >
// 데이터의 기본 구분 표현
</ XMI.Difference >
< XMI.Extensions >
// 메타모델의 확장정보 표시
</ XMI.Extensions >
</ XMI >
```

(그림 2) XMI 메타모델

## 3. XMI 메타모델 설계 및 구현

(그림 3)은 UML로 정의된 디자인패턴의 클래스모듈을 XMI 메타모델에서 메타데이터를 추출한 후 저장소에 저장하는 시스템의 형태를 설계한 것이다. 그리고 저장된 메타데이터를 활용하기 위해서 Java나 XML을 이용하여 새로운 클래스타입의 스키마나 패턴을 합성 또는 분리할 수 있도록 하였다. 여기서 메타데이터의 저장과 운용을 위해서 Windows 2000 Server와 MSSql 2000을 사용하였다.

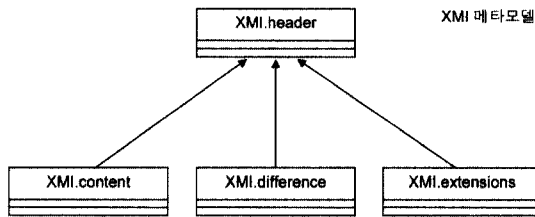


(그림 3) XMI 메타모델 시스템

여기서 DB에 저장된 메타데이터는 각 패턴에 대한 SuperClass와 SubClass의 관계를 파악할 수 있고, 클래스와 패턴의 정보를 상세하게 확인할 수가 있어 재사용에도 많은 이점을 가질 수 있다.

3.1 XMI 메타데이터 저장소(Repository) 설계 및 구축

본 연구에서는 XMI 메타모델의 정보를 저장소에 저장하기 위해서 <XMI.header>, <XMI.content>, <XMI.difference>, <XMI.extensions>를 각각의 클래스 부품으로 구분하여 추출하였다. 여기서 클래스를 SuperClass와 Sub-Class로 분리하여 이들 사이의 상속관계에 대한 정보를 기술하고자 하였다.



(그림 4) XMI 메타모델 클래스

본 연구의 저장소 설계에 적용된 엘리먼트에는 (그림 4)에서와 같이 Top 레벨에 해당하는 4개의 엘리먼트만을 포함시키고 있다. 기타의 많은 엘리먼트들이 존재하고 있지만 사용의 빈도수가 많지 않을 뿐만 아니라 모든 엘리먼트를 메타모델로 생성하여 DB에서 메타데이터로 관리하는 것은 매우 복잡하여 제외시켰다. 패턴에 대한 메타모델을 구성하기 위한 문법으로는 (그림 5)과 같이 정의할 수 있다. 정의된 문법에서 패턴은 또 다른 패턴과 관계의 집합으로 구성되고, 분리된 패턴은 다시 클래스와 관계의 집합으로 구성되고 있다. 그리고 클래스는 오퍼레이션으로 관계는 관련된 코드의 집합으로 구성된다.

```

<pattern> → <pattern> * <association>
<pattern> → <class> * <association>
<class> → <operation> * <attribute>
<association> → <relationship code>
    
```

(그림 5) 패턴의 메타모델 구성문법

(그림 6)은 XMI 메타모델에 대한 데이터베이스를 DDL(Data Definition Language)를 이용해서 생성하는 방법이다.

```

SQL>CREATE DATABASE [XML_MODEL] ON (NAME = N'XML_MODEL_Data', FILENAME = N'c:\MSSQL\data\XML_MODEL-EL_Data.MDF', SIZE = 1, FILEGROWTH = 10%) LOG ON (NAME = N'XML_MODEL_Log', FILENAME = N'c:\MSSQL\data\XML_MODEL_Log.LDF', SIZE = 1, FILEGROWTH = 10%) GO
    
```

(그림 6) XMI 메타모델 DB 생성

(그림 7)의 <XMI.header> 테이블의 생성에서는 메타모델의 종류와 XMI 정보기술에 대한 정의를 하는 부분의 엘리먼트로서 메타데이터에 대한 model, metamodel, metametamodel등을 선택해서 기술하는 부분이 포함되어있다. 또한 패턴뿐만 아니라, 컴포넌트 또는 여러 개의 클래스로 구성된 패키지(package)등을 구분하기 위해서 Pattern\_name으로 필드를 추가하면서 Primary Key로 지정하고 있다. 그리고 <XMI.header>를 Master로 하면서 [XMI.model]을 Secondary Key로 설정하여 다른 테이블과 연결가능 하도

```

SQL> CREATE TABLE [XMI_header] (
    [Pattern_Name] [char] (10) NOT NULL ,
    [XML_documentation] [char] (30) COLLATE NULL ,
    [model_xmi_name] [char] (30) NOT NULL ,
    [herf1] [char] (15) NOT NULL ,
    [metamodel_xmi_name] [char] (30) NOT NULL ,
    [herf2] [char] (15) NOT NULL ,
    [metametamodel_xmi_name] [char] (30) NULL ,
    [herf3] [char] (15) NULL ,
    CONSTRAINT [PK_XML_header] PRIMARY KEY
    CLUSTERED
    (
        [Pattern_Name]
    ) ON [PRIMARY] ,
    CONSTRAINT [FK_XML_header_XML_content]
    FOREIGN KEY
    (
        [model_xmi_name]
    ) REFERENCES [XMI_content] (
        [XMI_model]
    ) ,
    CONSTRAINT [FK_XML_header_XML_difference]
    FOREIGN KEY
    (
        [model_xmi_name]
    ) REFERENCES [XMI_difference] (
        [XMI_model]
    ) ,
    CONSTRAINT [FK_XML_header_XML_extensions]
    FOREIGN KEY
    (
        [model_xmi_name]
    ) REFERENCES [XMI_extensions] (
        [XMI_model]
    )
) ON [PRIMARY]
GO
    
```

(그림 7) <XMI.header> 테이블생성

록 설계하였다.

(그림 8)의 <XML.content> 테이블은 실제 디자인패턴의 클래스에 대한 메타모델정보를 갖는 XMI 정보기술을 엘리먼트로 가지는 부분이다. 여기서 [Class\_name1], [Class\_name2], [Class\_name3]로 세 개의 패턴클래스에 대한 개수만을 정의하고 있다. 이것은 디자인 패턴에 따라 클래스 수가 차이가 있어 일정한 필드의 숫자를 정하기가 쉽지 않으므로, 처음 설계할 때는 기본적인 몇 개의 필드만 작성하고 추후 추가가 요구되면 계속적으로 삽입을 할 수 있도록 하였다. 그리고 Master인 <XML.header>와 연결을 위해서 [XML.model]을 추가하여 Primary Key로 설정하고 있다. [Association\_name]의 역할은 SuperClass와 SubClass의 관계를 정의하는 부분으로 설정되었다.

```
SQL>CREATE TABLE [XML_content] (
    [XML_model] [char] (30) NOT NULL ,
    [Class_name1] [char] (30) NULL ,
    [xmi_id1] [char] (15) NULL ,
    [Class_name2] [char] (30) NULL ,
    [xmi_id2] [char] (10) NULL ,
    [Class_name3] [char] (10) NULL ,
    [xmi_id3] [char] (10) NULL ,
    [Association_name1] [char] (10) NULL ,
    [Association_name2] [char] (10) NULL ,
    CONSTRAINT [PK_XML_content] PRIMARY KEY
    CLUSTERED
    (
        [XML_model]
    ) ON [PRIMARY]
) ON [PRIMARY]
GO
```

(그림 8) <XML.content> 테이블 생성

(그림 9)와 (그림 10)은 각각 데이터의 기본구분 표현과 메타모델의 확장정보표시 부분을 클래스로 정의하고 있으나 일반적으로 XMI 메타모델에서는 사용되는 빈도가 낮다. 그리고 이 두개의 테이블에서도 <XML.header>와 연결을 위해서 [XML.model]을 추가하여 Primary Key로 설정하였다.

```
SQL>CREATE TABLE [XML_difference] (
    [XML_model] [char] (30) NOT NULL ,
    [XML_difference] [char] (30) NULL ,
    [XML_add] [char] (30) NULL ,
    [XML_delete] [char] (30) NULL ,
    [XML_replace] [char] (30) NULL ,
    CONSTRAINT [PK_XML_difference] PRIMARY
    KEY CLUSTERED
    (
        [XML_model]
    ) ON [PRIMARY]
) ON [PRIMARY]
GO
```

(그림 9) <XML.difference> 테이블 생성

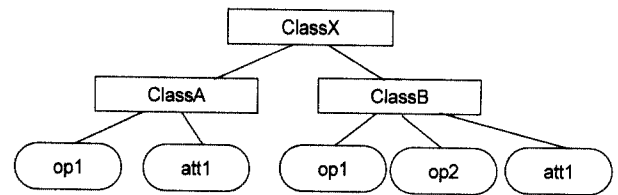
```
SQL>CREATE TABLE [XML_extensions] (
    [XML_model] [char] (30) NOT NULL ,
    [XML_extensions] [char] (30) NULL ,
    [xmi_extender] [char] (30) NULL ,
    CONSTRAINT [PK_XML_extensions] PRIMARY KEY
    CLUSTERED
    (
        [XML_model]
    ) ON [PRIMARY]
) ON [PRIMARY]
GO
```

(그림 10) <XML.extensions> 테이블

### 3.2 사례 연구를 이용한 평가

#### 3.2.1 적용모델 설계 및 XMI 메타모델 생성

본 논문에서는 앞에서 제안한 방법을 적용하기 위해서 세 개의 패턴모델을 제시하였다. 이 세 개의 모델들은 XMI 기반으로 표준의 메타모델을 생성하였으며, 다시 메타데이터로 분리하여 DB를 이용한 저장소에 이를 저장하였다. 그리고 두개 이상의 모델에서 패턴(pattern)과 클래스(class), 오퍼레이션(operation), 어트리뷰트(attribute)의 공통부분에

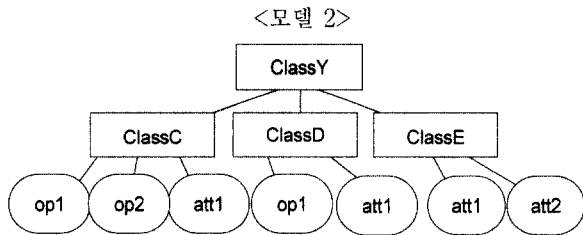


(그림 11) 패턴모델 1

```
File="Model1.xml" Namespace="Model1" :
<XML version="1." xmlns:UML="org.omg/UML1.3">
  <XML.header>
    <XML.model xmi.name="Model1" href="Model1"/>
    <XML.metamodel xmi.name="UML" href="UML.xml"/>
  </XML.header>
  <XML.content>
    <UML:Class name="ClassX" xmi.id="ClassX"/>
    <UML:Class name="ClassA" xmi.id="ClassA"
      generalization="ClassX"/>
    <UML:ModelElement.ownedElements>
      <UML:Attribute name="att1" type="idatt_a1"/>
      <UML:Operation name="op1" type="idop_a1"/>
    </UML:ModelElement.ownedElements>
    <UML:Class name="ClassB" xmi.id="ClassB"
      generalization="ClassX"/>
    <UML:ModelElement.ownedElements>
      <UML:Attribute name="att1" type="idatt_b1"/>
      <UML:Operation name="op1" type="idop_b1"/>
      <UML:Operation name="op2" type="idop_b2"/>
    </UML:ModelElement.ownedElements>
  </XML.content>
</XML>
```

(그림 12) 패턴모델 1 메타모델

대한 합성(composition)에 대해서도 메타데이터를 이용하였다. 일반적으로 객체지향모델링(object oriented modeling)에서는 UML을 이용한 방법이 많이 사용되고 있으나, 본 논문에서는 UML에 의해 산출된 각 클래스 다이어그램의 형식들을 XML형식으로 자동변환시키는 것을 표준화시키기 위한 방법으로 제안된 XMI(XML Metadata Interchange)를 이용하였다. 그래서 UML로 작성된 각종 다이어그램들은 XMI의 메타모델로 작성될 수 있으며, 또한 규정에 따라 XML로 표현될 수 있도록 하였다. 그리고 메타모델의 저장소 설계에서는 관계형데이터베이스(Relational Database)를 이용하여 메타데이터의 무결성을 위한 정규화가 적용된 테이블을 작성하였다.



(그림 13) 패턴모델 2

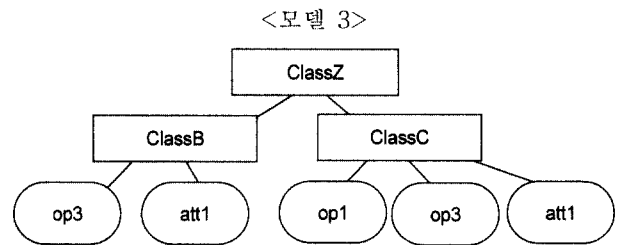
```

File="Model2.xml" Namespace="Model2" :
<XMI version="1." xmlns:UML="org.omg/UML1.3">
  <XML.header>
    <XML.model xmi.name="Model2" href="Model2"/>
    <XML.metamodel xmi.name="UML" href="UML.xml"/>
  </XML.header>
  <XML.content>
    <UML:Class name="ClassY" xmi.id="ClassY"/>
    <UML:Class name="ClassC" xmi.id="ClassC"
      generalization="ClassY"/>
    <UML:ModelElement.ownedElements>
      <UML:Attribute name="att1" type="idatt_c1"/>
      <UML:Operation name="op1" type="idop_c1"/>
      <UML:Operation name="op2" type="idop_c2"/>
    </UML:ModelElement.ownedElements>
    <UML:Class name="ClassD" xmi.id="ClassD"
      generalization="ClassY"/>
    <UML:ModelElement.ownedElements>
      <UML:Attribute name="att1" type="idatt_d1"/>
      <UML:Operation name="op1" type="idop_d1"/>
    </UML:ModelElement.ownedElements>
    <UML:Class name="ClassE" xmi.id="ClassE"
      generalization="ClassY"/>
    <UML:ModelElement.ownedElements>
      <UML:Attribute name="att1" type="idatt_e1"/>
      <UML:Attribute name="att2" type="idatt_e2"/>
    </UML:ModelElement.ownedElements>
  </XML.content>
</XMI>
    
```

(그림 14) 패턴모델 2 메타모델

위 (그림 12)는 객체지향프로그램에서 일반적으로 많이 사용되어지는 패턴의 형식이다. ClassX 부분은 추상화(abstract) 부분의 클래스를 정의하고 있는 SuperClass이고, 아래의 ClassA와 ClassB는 실체화(concrete)된 부분을 정의하는 SubClass이다.

그리고 (그림 12)는 (그림 11)의 패턴모델을 XMI 메타모델로 정의한 것이다. 여기서 SuperClass인 ClassX는 두개의 SubClass가 상속을 받고 있다. XMI 메타모델로 표현에서는 xmi.name에서 패턴모델의 이름을 부여하고 있으며, <XML.content>의 Class name에서 각각의 SuperClass와 SubClass를 정의하고 있다. 그리고 generalization부분에는 상속된 SubClass에서 SuperClass의 이름을 기입하고, 각 클래스의 오퍼레이션(operation)과 어트리뷰트(attribute)등을 정의한다. 또한, 이와 같은 방법으로 사례연구에 적용된 나머지 두개의 패턴모델 (그림 13), (그림 15)의 메타모델도 (그림 14), (그림 15)와 같이 생성하였고, 각각의 모델들은 메타데이터로 추출되어 DB저장소에 저장되었다.



(그림 15) 패턴모델 3

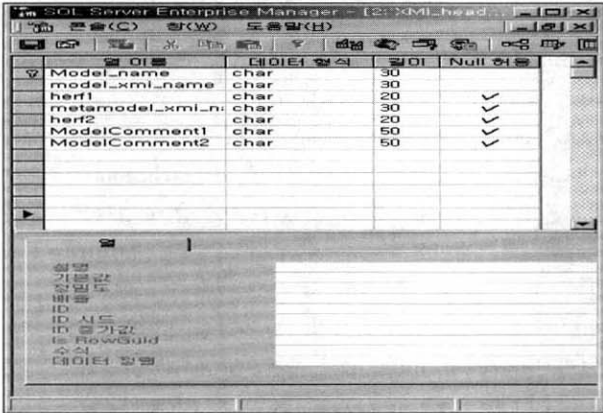
```

File="Model3.xml" Namespace="Model3" :
<XMI version="1." xmlns:UML="org.omg/UML1.3">
  <XML.header>
    <XML.model xmi.name="Model3" href="Model3"/>
    <XML.metamodel xmi.name="UML" href="UML.xml"/>
  </XML.header>
  <XML.content>
    <UML:Class name="ClassZ" xmi.id="ClassZ"/>
    <UML:Class name="ClassB" xmi.id="ClassB"
      generalization="ClassZ"/>
    <UML:ModelElement.ownedElements>
      <UML:Attribute name="att1" type="idatt_z1"/>
      <UML:Operation name="op3" type="idop_z3"/>
    </UML:ModelElement.ownedElements>
    <UML:Class name="ClassC" xmi.id="ClassC"
      generalization="ClassZ"/>
    <UML:ModelElement.ownedElements>
      <UML:Attribute name="att1" type="idatt_z1"/>
      <UML:Operation name="op1" type="idop_z1"/>
      <UML:Operation name="op3" type="idop_z3"/>
    </UML:ModelElement.ownedElements>
  </XML.content>
</XMI>
    
```

(그림 16) 패턴모델 3 메타모델

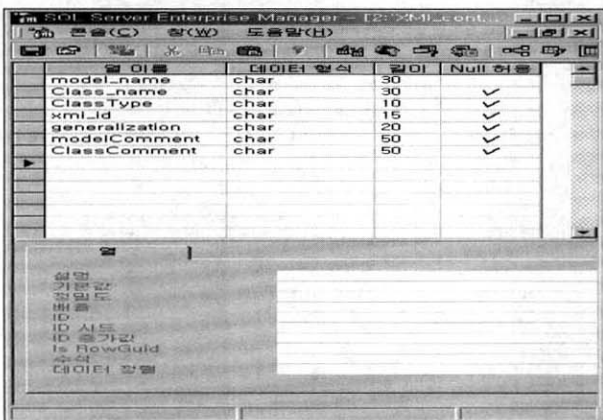
아래의 (그림 17)은 (그림 12), (그림 14), (그림 16)의

XMI 메타모델에서 <XMI.header>에 대한 메타데이터를 DB에서 하나의 테이블로 생성한 것이다. Model Comment1, ModelCommnet2는 저장소의 패턴 모델의 종류와 특징들을 기입하여 질의어를 통하여 쉽게 모델의 유형을 파악할 수 있도록 하였다.



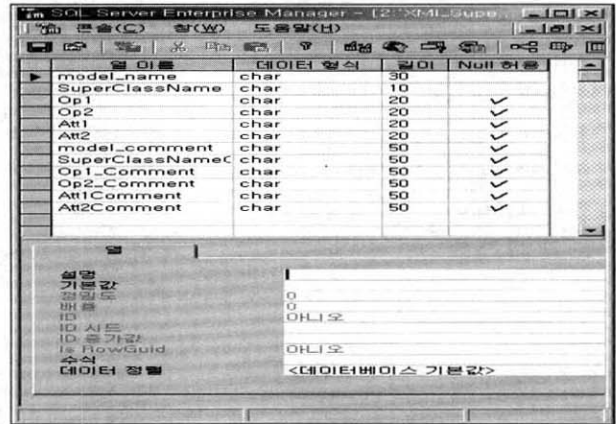
(그림 17) 패턴모델 XMI.header 테이블

(그림 18)은 실제로 패턴모델의 클래스에 대한 메타모델 정보를 기술하는 부분인 <XMI.content>로 적용된 모든 클래스뿐만 아니라 오퍼레이션이나 어트리뷰트 등을 메타데이터로 생성하여 DB에 저장한 테이블이다. 여기서 model\_name은 다른 테이블에서 사용되고 있는 동일한 패턴모델을 구분하기 위한 필드이고, model\_name을 Primary Key로 부여하면 필드에 대한 값으로 동일한 모델의 이름이 저장될 수 없으므로 유의해야 한다. modelComment와 Class Comment는 테이블에 저장된 많은 레코드들 중 각각의 패턴모델과 사용된 클래스의 기능에 대한 특징을 입력하는 필드이다. 이 역시 질의어를 통한 패턴모델 및 클래스의 특징을 정확히 파악할 수 있도록 한 것이며, 이들의 재사용에 대한 수월성을 제고하기 위한 방법으로 이용되고 있다.



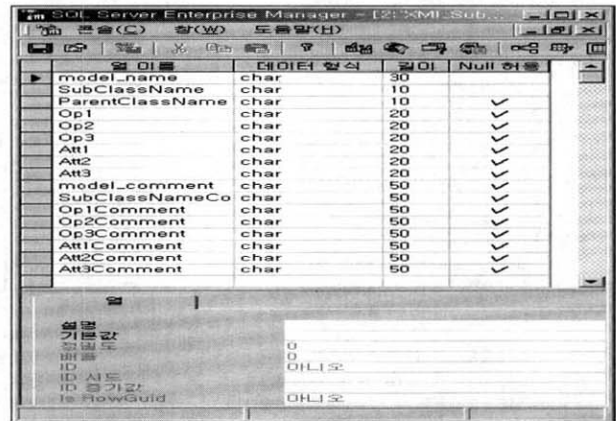
(그림 18) 패턴모델 XMI.content 테이블

(그림 19)는 패턴모델에서 SuperClass에 해당되는 부분을 메타데이터로 생성하여 DB에 저장한 테이블의 형식이다. 이 테이블에서도 model\_name 필드를 가지고 이 클래스가 속해있는 패턴모델을 파악할 수 있도록 하였다.



(그림 19) 패턴모델 SuperClass 테이블

(그림 20)은 모든 SubClass에 대한 정보를 메타데이터로 분리하여 저장한 테이블이며 각 클래스가 속한 패턴 모델 뿐만 아니라 Parent Class의 정보까지 가지고 있으며, Comment 부분을 두어 클래스의 특징을 기술하도록 하였다.



(그림 20) 패턴모델 SubClass 테이블

### 3.2.2 패턴모델에 대한 인식 테이블 작성

본 논문에서는 XMI 메타모델에 대한 세부적인 메타데이터를 관리하기 위한 방법으로 각 테이블에 대한 특징 및 기능을 각각의 테이블로 정의하여 관리할 수 있도록 하였다. 이 제안은 중복 모델 및 중복클래스의 생성을 방지할 수 있을 뿐만 아니라 재사용 측면에서도 전체적인 모델의 특성을 한눈에 파악할 수 있도록 한 것이다.

<표 1>, <표 2>는 본 논문에서 사례연구로 적용한 Model1, Model2, Model3에 대한 패턴모델과 클래스에 대한 인

식테이블로써 모든 모델과 클래스가 쉽게 관리될 수 있도록 한 것이다. 특히 패턴 및 클래스의 재사용뿐만 아니라, 합성 및 조립에 있어서 많은 참고 자료로 활용이 가능하도록 하였다.

<표 1> 패턴모델 인식 테이블

모델이름	SuperClass 개수	SubClass 개수	Pattern Type	모델기능
Model1	1	2	Factory	Triangle
Model2	1	3	Factory	Square
Model3	1	2	Factory	Circle

<표 2> 클래스 인식 테이블

클래스이름	클래스 Type	클래스종류	SubClass 개수	Parent 클래스
ClassX	Super	Abstract	2	-
ClassY	Super	Abstract	3	-
ClassZ	Super	Abstract	2	-
ClassA	Sub	Concrete	-	ClassX
ClassB	Sub	Concrete	-	ClassX
				ClassZ
ClassC	Sub	Concrete	-	ClassY
				ClassZ
ClassD	Sub	Concrete	-	ClassY
ClassE	Sub	Concrete	-	ClassY
클래스이름	Operation 개수	Attribute 개수	클래스기능	Comment
ClassX	-	--	Triangle	
ClassY	-	-	Square	
ClassZ	-	-	Circle	
ClassA	1	1	Straight line	
	2	1	Curve line	
ClassB	1	1	Curve line	
	2	1	Dot line	
ClassC	2	1	Dot line	
	2	1	Dot line	
ClassD	1	1	Solid line	
ClassE	-	2	Double line	

<표 3> 오퍼레이션 인식 테이블

Op 이름	Op Type	Op 기능	적용 클래스	Parent 클래스	종속 모델
Op1	string	---	ClassA	ClassX	Model1
	char	---	ClassB	ClassX	Model1
	char	---	ClassC	ClassY	Model2
	char	---	ClassD	ClassY	Model2
	---	---	---	---	---
Op2	int	---			
---					

<표 4> 어트리뷰트 인식 테이블

Att 이름	Att Type	Attr 크기	Att 기능	적용 클래스	Parent 클래스	종속 모델
att1	int	50	---	ClassA	ClassX	Model1
	long	30	---	ClassB	ClassX	Model1
	int	20	---	ClassC	ClassY	Model2
	int	20	---	ClassD	ClassY	Model2
att2	char	50				
---						

<표 3>과 <표 4>는 독립적인 테이블로 각 클래스에서 사용되고 있는 오퍼레이션과 어트리뷰트 등을 관리하는 테이블로써 각각의 특징과 소속 모델, 소속 클래스, Parent 클래스의 정보까지 정의하고 있다.

3.2.3 XMI 메타모델 합성

모델을 합성하는 방법으로는 두개 이상의 모델을 합성하여 새로운 모델을 생성하는 방법과 각 모델의 패턴과 클래스, 오퍼레이션, 어트리뷰트등의 공통부분을 통합하여 새롭게 오버래핑(overlapping)하는 방법으로 나눌 수 있다. 전자는 패턴이나 클래스에 대한 오퍼레이션이나 어트리뷰트들에 대한 특성 및 이해가 정확하게 되어 있지 않은 상태에서 공통의 비슷한 부분을 찾아서 합성하는 방법이며, 후자는 패턴 및 클래스, 오퍼레이션, 어트리뷰트들의 특성 및 기능, 버전(version) 관리 등이 정확히 되어 있는 상태에서 이들을 서로 비교하여 공통의 부분을 합성하는 방법이다. 본 논문에서는 이들 합성에 대한 수월성을 제고하기 위해 메타데이터 생성 시 다음과 같은 특성을 추가 하여 합성에 효율적으로 이용하도록 하였다.

- 패턴모델의 특성에 대한 명세
- 클래스의 각 엘리먼트들에 대한 명세
- 클래스의 특성에 대한명세
- 오퍼레이션의 특성에 대한 명세
- 어트리뷰트의 특성에 대한 명세

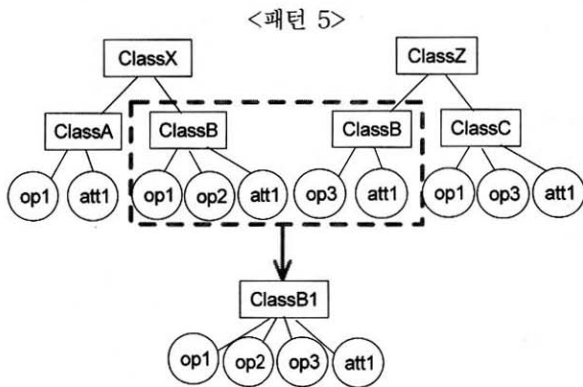
그리고 합성을 위한 조건으로 아래와 같이 패턴모델, 슈퍼클래스, 서브클래스, 오퍼레이션, 어트리뷰트를 정의하고 클래스의 합성을 위한 알고리즘을 정의하였다.

- Model = M1, M2, ..., Mn
- SuperClass = SpC1, SpC2, ..., SpCn
- SubClass = SuC1, SuC2, ..., SuCn
- Operation = Op1, Op2, ..., Opn
- Attribute = Att1, Att2, ..., Attn

```

While Model
  if Mi corresponds with Mj then
    if SpCi corresponds with SpCj then
      if Opi corresponds with Opj then
        composition Classi.Classi.Opi and Classj.Classj.Opj
      /* Classi.Classi(superclass.subclass) */
    else if atti corresponds with attj then
      composition Classi.Classi.atti and Classj.Classj.attj
    end if
  end if
end if
end While
    
```

(그림 21) 클래스합성 알고리즘



(그림 22) 클래스합성

위 (그림 22)에서는 (그림 11)와 (그림 15)의 두개의 패턴 모델을 합성한 것이다. 여기서 ClassX와 ClassZ에서 가지고 있는 ClassB에 대하여 공통의 부분을 오버래핑을 이용하여 합성하였다. 본 논문에서는 ClassB를 가지고 있는 모델을 찾기 위한 방법으로 DB에 저장된 메타데이터를 (그림 23)의 질의어를 이용하였으며 결과로 생성된 테이블인 (그림 24)에서 ClassB의 오퍼레이션과 어트리뷰트등을 확인할 수 있다.

```

SQL>select xmi_superclass.model_name, superclassname,
subclassname, xmi_subclass.op1, xmi_subclass.op2, xmi_subclass.op3,
xmi_subclass.att1, xmi_subclass.att2
from xmi_superclass join xmi_subclass
on xmi_superclass.model_name = xmi_subclass.model_name
    
```

(그림 23) 클래스 검색 질의어

(그림 24) 검색된 클래스 테이블

(그림 25)는 패턴모델에서 ClassB를 찾아서 두개의 클래스를 합성하는 뷰(view)를 생성하는 방법이다. 그리고 (그림 26)은 새로 합성된 클래스의 테이블을 보여주고 있다.

```

SQL>create view ClassB1
as
select model_name, subclassname, op1, op2, op3, att1,att2,att3
from xmi_subclass
where subclassname = 'ClassB'
    
```

(그림 25) ClassB의 합성 뷰(view) 생성

(그림 26) ClassB 합성 뷰(view) 테이블

지금까지 본 논문에서는 패턴 및 클래스설계에서 각각의 구성요소들을 세부적인 메타데이터로 분리하여 DB에 저장하는 방법을 설계하였다. 그 결과 패턴과 클래스의 효율적인 재사용과 중복 클래스의 제거, 질의어를 통한 패턴 및 클래스의 검색 등을 쉽게 할 수 있도록 하였으며 명세된 인식테이블을 통한 전체적인 소프트웨어 구성요소의 파악도 용이하도록 하였다.

#### 4. 평가

본 연구에서 제안한 방법의 성능평가를 위해서 앞절 3.2의 사례연구(Triangle, Square, Circle) 모델과 클래스를 가지고 Card와 Glass[12]의 소프트웨어 설계복잡도를 <표 5>와 같이 측정하였다. 그 결과로 복잡도의 기준으로 판단할 수 있는 시스템복잡도가 35.8%정도 감소되어 본 제안방법이 소프트웨어설계에서 성능향상을 가질 수 있음을 알 수 있다.

- 구조복잡도 :  $S(i) = f_{out}^2(i)$
  - 자료복잡도 :  $D(i) = v(i) / [f_{out}(i) + 1]$
  - 시스템복잡도 :  $C(i) = S(i) + D(i)$
- \*  $i$  = 모듈,  $f_{out}(i)$  = 모듈의 출력력,  $v(i)$  =  $i$ 의 입출력 변수의 수



〈표 5〉 설계복잡도

구분	모델	제안전	제안후	비고
구조 복잡도	전체클래스	225	144	-81
	ClassA	4	4	-
	ClassB	25	9	-16
	ClassC	9	9	-
자료 복잡도	전체클래스	1.5	1.46	-0.04
	ClassA	2.67	2.67	-
	ClassB	1.67	1.25	-0.42
	ClassC	1.5	1.5	-
시스템 복잡도	전체클래스	226.5	145.46	-81.04
	ClassA	6.67	6.67	-
	ClassB	26.67	10.25	-16.42
	ClassC	10.5	10.5	-

## 5. 결론 및 향후 연구

소프트웨어 설계패턴을 표현하는 방법에서 표준화된 방법을 적용하여 모델링하기 위해서 OMG의 UML의 클래스 다이어그램을 이용하여 패턴을 설계할 수 있다. 그러나 이 패턴을 정형화하기 위한 방법이 설계자에 따라 서로 달라 클래스나 패턴의 재사용에 많은 문제점들을 갖고 있다. 본 논문에서는 이를 개선하기 위한 방법으로 패턴과 클래스에 대한 메소드 및 어트리뷰트까지 라이브러리로 저장 가능한 형태로 설계를 하였다. 즉, XMI를 이용하여 메타모델을 설계하고 다시 세부적인 메타데이터들을 분리하여 정규화된 관계데이터베이스 저장소를 구축하고 있다. 패턴을 정형화하는 위한 방법으로는 UML를 이용하여 클래스다이어그램을 설계하였고 XMI 메타모델의 Top 레벨에 해당하는 4개의 엘리먼트만을 포함시켰다. 그리고 엘리먼트를 각각의 테이블 단위로 분리하여 메타데이터를 저장하였다. 여기서 메타데이터를 DB에 저장하기 위해서는 오퍼레이션이나 어트리뷰트의 타입, 이름설정 및 클래스의 상속관계를 표현해야 하므로, 본 연구에서는 이 해결책으로 오퍼레이션이나 어트리뷰트 인식테이블을 작성하였다. 또한, 질의어를 이용하여 저장소의 패턴과 클래스에 대한 자세한 정보를 조회할 수 있으며, 새로운 패턴 클래스를 조립하기 위한 뷰(view)의 생성도 용이하도록 하였다. 그리고 클래스합성에 대해서도 DB에 저장된 메타데이터를 이용하였다. 그러나 메타데이터를 DB로 저장하기 위해서는 설계자가 직접데이터를 분류하고 테이블을 작성해야 한다. 또한, 인식테이블의 작성성이 부가적으로 요구되어지며, 공통의 클래스 합성에 있어서도 도구화된 인터페이스를 통해서 조회 및 합성하는 방법이 연구되어야 한다. 마지막으로 클래스합성 후 단위 클

래스의 메타데이터 생성에 대한 어려움이 발생하므로 추가적인 연구가 요구되어진다.

## 참고 문헌

- [1] Grady Booch, "Object-Oriented Analysis and Design with Applications," 2nd Edition, The Benjamin/Cummings Series in Object-Oriented Software Engineering, 1994.
- [2] Derek Coleman, Patrick Arnold, Stephanie, Chris Dollin, Helena Gilchrist, Fiona Hayes, Paul Jeremes, "Object-Oriented Development. The Fusion Method" Prentice Hall, 1994.
- [3] Steve Cook, John Daniels, "Designing Object Systems. Object-Oriented Modelling with Syntropy," Prentice Hall, 1994.
- [4] Gregor Engels, Luuk Groenewegen. "Object-Oriented Modeling : A Roadmap," In proceedings of "The Future of Software Engineering 2000," Editor : Anthony Finkelstein, International Conference on Software Engineering.
- [5] Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen, Michael Tyrsted, "Creative Object Oriented Modelling," Department of Computer Science, University of Aarhus, Aabogade 34, 8200 Aarhus N, Denmark, www.ideogramic.com/download/resources/ecoop2000.pdf.
- [6] "OMG Unified Modeling Language Specification (draft)," Version 1.3. beta R7, June, 1999.
- [7] Brigit Demuth, Sven Obermaier, "Experiments with XMI Based Transformations of Software Models," <http://ase.arc.nasa.gov/wtuml01/submissions/demuth-hussman-obermaier.pdf>, 2003.
- [8] Wu, I. C., Hsieh, S. H., "An UML-XML-RDB Model Mapping Solution for Facilitating Information Standardization and Sharing in Construction Industry," International Symposium on Automation and Robotics in Construction, 19th (ISARC). Proceedings. National Institute of Standards and Technology, Gaithersburg, Maryland, pp.23-25, pp. 317-321, September, 2002.
- [9] "XMI Gets the Capability to convey information," <http://www.3.ibm.com/software/ad/standards/xmiwhite0399.pdf>, 1999.
- [10] The Object Management Group, <http://www.omg.org>, 2003.
- [11] 최한용, "XMI기반의 디자인패턴 설계 및 지원환경 구축", 경희대학교 박사학위논문, 2002.
- [12] Card, D. N. and R. L. Glass, "Measuring Software Design Quality," Prentice Hall, 1990.



### 이 돈 양

e-mail : dylee6211@hanmail.net

1987년 대구대학교 통계학과(이학사)

1993년 경희대학교 전자계산학과(공학석사)

2002년 경희대학교 전자계산공학과  
(박사수료)

1995년~2002년 대한상공회의소

2003년~현재 경인여자대학 전산정보학 겸임교수

관심분야 : EJB, 디자인패턴, XML, OOD, 소프트웨어 재사용



### 송 영 재

e-mail : yjsong@khu.ac.kr

1969년 인하대학교 전자공학과(공학사)

1786년 일본 keio대학교 전산학과(공학석사)

1980년 명지대학교 전산학과(공학박사)

1976년~현재 경희대학교 컴퓨터공학과  
교수

관심분야 : 소프트웨어 재사용, CASE도구