XMI기반 객체지향 메타모델 생성

이 دون 양* · 송영재**

요 약

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

Generation of Object-Oriented Metamodel based on XMI

Don-Yang Lee* · Young-Jae Song**

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 Metadatas. Design DB repository for created Metadatas storing and add pattern and information about each class composition and use query and did so that reusability and extension of Metadatas may be easy.

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

1. 서론

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

도가 35.8% 정도 감소되어 소프트웨어설계의 성능향상을 가능할 수 있었다.

2. 관련 연구


![Diagram](그림 1) XMI와 XMI DTD/Schema


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

![Diagram](그림 2) XMI 메타모델

3. XML 메타모델 설계 및 구현

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

3.1 XML 메타데이터 저장소(Repository) 설계 및 구축

본 연구에서는 XML 메타모델의 정보를 저장소에 저장하기 위해서 <XMLheader>, <XMLcontent>, <XMLdifference>, <XMLextension>를 각각의 클래스 부분으로 구분하여 추출하였다. 여기서 클래스를 SuperClass와 SubClass로 분리하여 이들 사이의 상속관계에 대한 정보를 기술하고자 하였다.

본 연구의 저장소 설계에 적용된 엔리먼트는 (그림 4) 여서와 같이 Top 레벨에 해당하는 4개의 엔리먼트만을 포함시킨다. 기타의 많은 엔리먼트들이 존재하고 있지만 사용자의 빈도가 많지 않기 때문에 엔리먼트들은 메타모델로 생성하여 DB에서 메타데이터로 관리하는 것은 매우 복잡할 것으로 예측하였다. 패턴에 대한 메타모델을 구성하기 위한 문법으로는 (그림 5)와 같이 정의할 수 있다. 정의된 문법에서 패턴은 이와 같은 패턴과 관계에 입힌 패턴으로 구성되며, 패턴에는 다시 클래스와 패턴의 집합으로 구성되고, 그리고 클래스는 오빠레이션으로 관계는 관련된 코드의 집합으로 구성된다.

(그림 5) 패턴의 메타모델 구성문법

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

```
SQL> CREATE DATABASE [XMI.Model] ON (NAME = 'XMI.Model', FILENAME = 'xmi.dat') MANAGEABLE Starter FILESIZE = 2MB, FILEGROWTH = 10%
SQL> CREATE DATABASE [XMI.Model_Log] ON (NAME = 'XMI.Model_Log', FILENAME = 'xmi.dat') MANAGEABLE Starter FILESIZE = 2MB, FILEGROWTH = 10%
```

(그림 6) XML 메타모델 DB 생성

(그림 7)의 <XMLheader> 태그들에 생성되는 메타모델의 종류와 XMI 정보기술에 대한 정보를 하는 부분의 엔리먼트로서 메타데이터에 대한 model, metamodel, metametamodel등을 선택해서 기록하는 부분에 포함되었 다. 또한 패턴뿐만 아니라, 패키지 또는 여러 개의 클래스로 구성된 패키지(package)들을 구분하기 위해서 Pattern name으로 편드를 추가하면서 Primary Key로 지정하고 있다. 그리고 <XMLheader>를 Master로 하면서 [XMLmodel]을 Secondary Key로 설정하여 다른 태그들과 연결가능 하도
록 설명하였다.

(그림 8)의 <XML.content> 테이블은 실제 디자인 패턴의 클래스에 대한 메타데이터 정보를 갖는 XMI 정보 기술을 열리먼트로 가지는 부분이다. 여기서 [Class_name1], [Class_name2], [Class_name3]로 세 개의 패턴 클래스에 대한 개수만을 정의하고 있다. 이것은 디자인 패턴에 따라 클래스 수가 차이가 있어 일정한 필드의 숫자를 정정하기가 쉽지 않으므로, 처음 설계할 때의 기본적인 몇 개의 필드만 작성하고 추후 추가가 요구되면 계속적으로 삽입할 수 있도록 하였다. 그리고 Master인 <XML.header>와 연결 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, 
[xml_id1] char(15) NULL, 
[Class_name2] char(30) NULL, 
[xml_id2] char(10) NULL, 
[Class_name3] char(10) NULL, 
[xml_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 10) <XML.extensions> 테이블 생성

(그림 9)와 (그림 10)은 각각 데이터의 기본 구조 표현과 메타데이터의 확장정보 사이 부분을 클래스로 정의하고 있으나 일반적으로 XMI 메타데이터에서는 사용하는 변동이 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 11) 패턴모델 1

3.2 사례 연구를 이용한 평가

3.2.1 적용모델 설계 및 XMI 메타모델 생성

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

```
File="Model1.xml" NameSpace="Model1" :
<XML version="1.1" xmlns :UML="org.omg/UML1.3">
<XML.header>
<XML.model xmlns="Model1" href="Model1.xml"/>
<XML.metamodel xmlns="UML" href="UML.xml"/>
</XML.header>
<XML.content>
<UML :Class name="ClassX" xml:id="ClassX"/>
<UML :Class name="ClassX" xml:id="ClassX" 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" xml:id="ClassB" generalization="ClassX"/>
<UML :ModelElement ownedElements>
<UML :Attribute name="att1" type="idatt_b1"/>
<UML :Operation name="op1" type="idop_b1"/>
</UML :ModelElement ownedElements>
</UML :ModelElement ownedElements>
</XML.content>
</XML>
```

(그림 12) 패턴모델 1 메타모델
대한 합성(composition)에 대해서도 메타데이터를 이용하였 다. 일반적으로 객체지향모델링(object oriented modeling) 에서는 UML을 이용한 방법이 많이 사용되고 있으나, 본 논문에서는 UML에 의해 모델링 각 클래스 대상의 레지럼 형태를 XML형식으로 자동변환시키는 것을 표준화하기 위한 방법으로 제안된 XMI(XML Metadata Interchange)를 이용하였다. 그래서 UML로 작성된 각종 다이어그램들은 XMI의 메타모델로 작성될 수 있으며, 또한 규칙에 따라 XML로 표현될 수 있도록 하였다. 그리고 메타모델의 저장소 설계에서는 관계형데이터베이스(Relational Database)를 이용하여 메타데이터의 무결성을 위한 정규화가 적용된 데이터 볼을 작성하였다.

위 (그림 12)는 객체지향프로그래밍에서 일반적으로 많이 사용되어지는 패턴의 형식이다. ClassX 부분은 추상화 (abstract) 부분의 클래스를 정의하고 있는 SuperClass이고, 아래의 ClassA와 ClassB는 실제화(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 표현하여는 xml:namespace에서 패턴모델의 이름을 부여하고 있으며, <XML:content>의 Class name에서 각각의 SuperClass와 SubClass를 정의하고 있다. 그리고, generalization부분에는 상속된 SubClass에서 SuperClass의 이름을 기입하고, 각 클래스의 오브젝션(operation)과 어트리뷰트(attribute)들을 정의한다. 또한, 이와 같은 방법으로 사례연구에 적용된 나머지 두개의 패턴모델 (그림 13), (그림 15)의 메타모델도 (그림 14), (그림 16)과 같이 작성하였고, 각각의 모델들은 메타데이터로 추출되어 DB저장소에 저장되었다.
XMI 패턴모델에서 <XML.header>에 대한 메타데이터를 DB에서 검색하는 테이블로 생성한 것이다. Model Comment, ModelComment2는 저장소의 패턴 모델의 종류와 특성을 기입하여 접근하여 쉽게 모델의 유형을 파악할 수 있도록 하였다.

(그림 17) 패턴모델 XML.header 테이블

(그림 18)은 실제 패턴모델의 클래스에 대한 메타데이터 정보를 기술하는 부분인 <XML.content>로 적용된 모든 클래스뿐만 아니라 메시지나 데이터스트 등은 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 메소드들 중 각각의 메소드 내의 사용된 클래스의 기능에 대한 특성을 입력하는 필드이다. 이 역시 접근을 통한 패턴모델 및 클래스의 특성을 정확히 파악할 수 있도록 한 것이며, 이들의 재사용에 대한 관심성을 제고하기 위한 방법으로 이용되고 있다.

(그림 19) 패턴모델 SuperClass 테이블

(그림 20)는 모든 SubClass에 대한 정보를 메타데이터로 분리하여 저장한 데이터이며 각 클래스가 속한 패턴 모델뿐만 아니라 Parent Class의 정보까지 가지고 있으며, Comment 부분을 두어 클래스의 특성을 기술하도록 하였다.

(그림 20) 패턴모델 SubClass 테이블

3.2.2 패턴모델에 대한 인식 테이블 작성

본 논문에서는 XMI 패턴모델에 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 사용한 Model1, Model2, Model3에 대한 패턴모델과 클래스에 대한 인
식테이블로서 모든 모델과 클래스가 쉽게 관리될 수 있도록 한 것이다. 특히 패턴 및 클래스의 재사용뿐만 아니라, 합성 및 조립에 있어서 많은 장고 자료 활용이 가능하도록 하였다.

### 표 1 패턴모델 인식 테이블

<table>
<thead>
<tr>
<th>모델명</th>
<th>SuperClass 개수</th>
<th>SubClass 개수</th>
<th>Pattern Type</th>
<th>모델기능</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model1</td>
<td>1</td>
<td>2</td>
<td>Factory</td>
<td>Triangle</td>
</tr>
<tr>
<td>Model2</td>
<td>1</td>
<td>3</td>
<td>Factory</td>
<td>Square</td>
</tr>
<tr>
<td>Model3</td>
<td>1</td>
<td>2</td>
<td>Factory</td>
<td>Circle</td>
</tr>
</tbody>
</table>

### 표 2 클래스 인식 테이블

<table>
<thead>
<tr>
<th>클래스명</th>
<th>클래스 Type</th>
<th>클래스종류</th>
<th>SubClass 개수</th>
<th>Parent 클래스</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassX</td>
<td>Super</td>
<td>Abstract</td>
<td>2</td>
<td>ClassX</td>
</tr>
<tr>
<td>ClassY</td>
<td>Super</td>
<td>Abstract</td>
<td>3</td>
<td>ClassZ</td>
</tr>
<tr>
<td>ClassZ</td>
<td>Sub</td>
<td>Concrete</td>
<td>-</td>
<td>ClassX</td>
</tr>
<tr>
<td>ClassA</td>
<td>Sub</td>
<td>Concrete</td>
<td>-</td>
<td>ClassX</td>
</tr>
<tr>
<td>ClassB</td>
<td>Sub</td>
<td>Concrete</td>
<td>-</td>
<td>ClassZ</td>
</tr>
<tr>
<td>ClassC</td>
<td>Sub</td>
<td>Concrete</td>
<td>-</td>
<td>ClassY</td>
</tr>
<tr>
<td>ClassD</td>
<td>Sub</td>
<td>Concrete</td>
<td>-</td>
<td>ClassY</td>
</tr>
<tr>
<td>ClassE</td>
<td>Sub</td>
<td>Concrete</td>
<td>-</td>
<td>ClassY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>클래스명</th>
<th>Operation 개수</th>
<th>Attribute 개수</th>
<th>클래스기능</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassX</td>
<td>-</td>
<td>-</td>
<td>Triangle</td>
<td></td>
</tr>
<tr>
<td>ClassY</td>
<td>-</td>
<td>-</td>
<td>Square</td>
<td></td>
</tr>
<tr>
<td>ClassZ</td>
<td>-</td>
<td>-</td>
<td>Circle</td>
<td></td>
</tr>
<tr>
<td>ClassA</td>
<td>1</td>
<td>1</td>
<td>Straight line</td>
<td></td>
</tr>
<tr>
<td>ClassB</td>
<td>1</td>
<td>1</td>
<td>Curve line</td>
<td></td>
</tr>
<tr>
<td>ClassC</td>
<td>2</td>
<td>1</td>
<td>Dot line</td>
<td></td>
</tr>
<tr>
<td>ClassD</td>
<td>2</td>
<td>1</td>
<td>Dot line</td>
<td></td>
</tr>
<tr>
<td>ClassE</td>
<td>1</td>
<td>2</td>
<td>Solid line</td>
<td></td>
</tr>
</tbody>
</table>

### 표 3 오퍼레이션 인식 테이블

<table>
<thead>
<tr>
<th>Op 이름</th>
<th>Op Type</th>
<th>Op 기능</th>
<th>작용 클래스</th>
<th>Parent 클래스</th>
<th>종속 모델</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1</td>
<td>string</td>
<td>---</td>
<td>ClassA</td>
<td>ClassX</td>
<td>Model1</td>
</tr>
<tr>
<td>Op2</td>
<td>int</td>
<td>---</td>
<td>ClassB</td>
<td>ClassX</td>
<td>Model1</td>
</tr>
</tbody>
</table>

### 표 4 어트리뷰트 인식 테이블

<table>
<thead>
<tr>
<th>Attr 이름</th>
<th>Attr Type</th>
<th>Attr 크기</th>
<th>작용 클래스</th>
<th>Parent 클래스</th>
<th>종속 모델</th>
</tr>
</thead>
<tbody>
<tr>
<td>att1</td>
<td>int</td>
<td>50</td>
<td>ClassA</td>
<td>ClassX</td>
<td>Model1</td>
</tr>
<tr>
<td>att2</td>
<td>char</td>
<td>50</td>
<td>ClassD</td>
<td>ClassY</td>
<td>Model2</td>
</tr>
</tbody>
</table>

### 3.2.3 XMI 메타모델 합성
모델을 합성하는 방법으로는 두 개 이상의 모델을 합성하여 새로운 모델을 생성하는 방법과 각 모델의 패턴과 클래스, 오퍼레이션, 어트리뷰트 등의 공통부분을 동형하여 새로운 클래스(mapping), 클래스, 오퍼레이션, 어트리뷰트들의 공통부분을 동형하여 새로운 오퍼레이션과 어트리뷰트들에 대한 특성 및 이에 대해 정의하게 되어 있는 상태에서 각각의 특성을 배우고 이를 서로 비교하여 공통의 부분을 합성하는 방법이다. 본 논문에서는 이로 합성에 대한 수원성을 제고하기 위해 메타데이터 생성과 다음과 같은 특성을 추가하여 합성에 효율적으로 이용하도록 하였다.

- 패턴모델의 특성에 대한 명세
- 클래스의 각 어트리뷰트들에 대한 명세
- 오퍼레이션의 특성에 대한 명세
- 오퍼레이션의 특성에 대한 명세
- 어트리뷰트의 특성에 대한 명세

그리고 합성을 위한 조건으로 아래와 같이 패턴모델, 수퍼클래스, 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 = At1, At2, ..., Attn
(그림 21) 클래스함성 알고리즘

<예제 5>

위 (그림 22)에서는 (그림 11)와 (그림 15)의 두개의 패턴 모델을 합성한 것이다. 여기서 ClassX와 ClassZ에서 가지고 있는 ClassB에 대하여 공통의 부분은 오버래핑을 이용하여 합성하였다. 본 논문에서는 ClassB를 가지고 있는 모델을 찾기 위한 방법으로 DB에 저장된 메타데이터들 (그림 23)의 패턴을 이용하였으며 결과로 생성된 태이블인 (그림 24)에서 ClassB의 오피러션과 어트리뷰트들을 확인할 수 있다.

SQL
```sql
CREATE VIEW ClassB 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) 태이블

지금까지 본 논문에서는 패턴 및 클래스설계에서 각각의 구성요소들을 세부적인 메타데이터로 분류하여 DB에 저장하는 방법을 설계하였다. 그 결과 패턴과 클래스의 효율적인 사용과 중복 클래스의 제거, 임의를 통한 패턴 및 클래스의 검색 등을 쉽게 할 수 있도록 하였으며 생성된 인식태이블을 통한 전체적인 소프트웨어 구성요소의 파악도 용이하도록 하였다.

4. 평 가

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

- 구조복잡도: \( S(i) = f_{1}(i) \)
- 자료복잡도: \( D(i) = \rho(i) / (f_{1}(i) + 1) \)
- 시스템복잡도: \( C(i) = S(i) + D(i) \)
- \( i \) =모듈, \( f_{1}(i) \) =모듈의 출력, \( \rho(i) \) =\( i \)의 입력입력 변수의 수
<table>
<thead>
<tr>
<th>구분</th>
<th>목 표</th>
<th>제안전</th>
<th>제안후</th>
<th>비고</th>
</tr>
</thead>
<tbody>
<tr>
<td>구조 복잡도</td>
<td>전체글러스</td>
<td>225</td>
<td>144</td>
<td>-81</td>
</tr>
<tr>
<td>ClassA</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ClassB</td>
<td>25</td>
<td>9</td>
<td>-16</td>
<td></td>
</tr>
<tr>
<td>ClassC</td>
<td>9</td>
<td>9</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>자료 복잡도</td>
<td>전체클래스</td>
<td>1.5</td>
<td>1.46</td>
<td>-0.04</td>
</tr>
<tr>
<td>ClassA</td>
<td>2.67</td>
<td>2.67</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ClassB</td>
<td>1.67</td>
<td>1.25</td>
<td>-0.42</td>
<td></td>
</tr>
<tr>
<td>ClassC</td>
<td>1.5</td>
<td>1.5</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>시스템 복잡도</td>
<td>전체클래스</td>
<td>265.5</td>
<td>145.46</td>
<td>-81.04</td>
</tr>
<tr>
<td>ClassA</td>
<td>6.67</td>
<td>6.67</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ClassB</td>
<td>26.67</td>
<td>10.25</td>
<td>-16.42</td>
<td></td>
</tr>
<tr>
<td>ClassC</td>
<td>10.5</td>
<td>10.5</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

5. 결론 및 확후 연구

스프레드시에 설계패턴을 표현하는 방법에서 표준화된 방법을 적용하여 모델링하기 위해서 OMG의 UML의 클래스 다이어그램을 이용하여 패턴을 설계할 수 있다. 그러나 이 패턴은 경험화되기 위한 방법이 설계자에 따라 서로 달라 클래스나 패턴의 재사용에 많은 문제점을 갖고 있다. 본 논문에서는 이를 개선하기 위한 방법으로 패턴과 클래스에 대한 메소드와 어트리뷰트까지 라이브러리로 저장이 가능한 형태로 설계를 하였다. 즉, XMI를 이용하여 메타정보를 설계하고 다시 세부적인 메타데이터들을 분리하여 정규화 된 메타데이터 저장소를 구축하고 있다. 패턴을 경험화하기 위한 방법으로 UML을 이용하여 클래스 다이어그램을 설계하였고 XMI 메타모델의 Top 레벨에 해당하는 4개의 엘리먼트만을 포함시켰다. 그리고 엘리먼트들을 각각의 메타데이터를 단위로 분리하여 메타데이터를 저장하였다. 여기서 메타데이터를 DB에 저장하기 위해서는 오피레인이나 어트리뷰트의 타입, 이름 설명 및 클래스의 상속관계를 표현해야 하며, 본 연구에서는 이 해석으로 오피레이션이나 어트리뷰트 인스턴스를 작성하였다. 또한, 절리어를 이용하여 저장소의 패턴과 클래스에 대한 자세한 정보를 조회할 수 있으며, 새로운 패턴 클래스를 조립하기 위한 뷰 (view)의 생성도 용이하도록 하였다. 그리고 클래스실행에서도 DB에 저장된 메타데이터를 이용하였다. 그러나 메타데이터를 DB로 저장하기 위해서는 설계자가 직접데이터를 분류하고 태그를 작성해야 한다. 또한, 인스턴스들의 작성이 무기적으로 요구되며, 공동의 클래스 할당에 있어도 도구화된 인터페이스를 통해서 조화 및 힙성을 하는 방법이 연구되어야 한다. 마지막으로 클래스함성 후 단위 클래스의 메타데이터 생성에 대한 이론이 발달함으로 추가적인 연구가 요구되는데 있다.

참고 문헌

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 전임교수

관심분야: EJB, 디자인패턴, XML, OOD, 소프트웨어 제작

송 영 재

e-mail: yjsong@khu.ac.kr
1969년 인하대학교 전자공학과(공학사)
1976년 일본 Keio대학교 전산학과(공학석사)
1980년 명지대학교 전산학과(공학박사)
1976년~현재 경희대학교 컴퓨터공학과

교수

관심분야: 소프트웨어 제작, CASE도구