Z명세를 이용한 EJB 컴포넌트의 구현

마대성
광주교육대학교 전산교육과

요 약

소프트웨어 명세는 비정형, 반 정형, 정형 기법이 존재한다. 이 중 정형 기법은 수학적인 이론을 바탕으로 명세의 모호성, 애매성, 불완전성을 제거하는데 효율적으로 사용되어 왔다.

본 논문에서는 Z 정형명세로부터 EJB의 자바 소스코드 까지의 구현 단계를 제안하였다. 또한, 사례 연구로 Stack을 명세하고 EJB 컴포넌트를 구현하는 단계를 제시하였다. 결론적으로 Z의 명세는 스키마의 분류, 인터페이스의 정의, Post-condition과 Pre-condition의 정의 단계를 거치고 정제를 통해서 EBJ의 인터페이스, 예외 클래스, 메소드 구현에 사용할 수 있음을 보였다.

Implementation of EJB Component by Using Z specification

Dai Sung Ma
GwangJu National University of Education, Dept of Computer Science of Education

Abstract

There are informal, semi-formal and formal methods in software specification. Among them, formal method which is based on mathematical theory had been used to remove ambiguity, incompleteness and contradiction efficiently.

In this paper, we propose implementation steps from Z specification to EJB source code, Also, as a case study we show steps consisted of specifying stack data structure and implementing it in EJB. In conclusion, Z specification proved to be capable of implementing EJB interface, exception class, method, through refinement and definition of schema, interface, post-condition, pre-condition.

keywords : Z specification, Component based development, EJB

1. 서론

소프트웨어 규모는 커지고 있으나, 소프트웨어 개발 기술의 발전 속도는 하드웨어 발전 속도에 비해 낙후된 실정이다. 대부분의 소프트웨어 프로젝트들은 개발 기간이 지연되거나 예산을 초과하거나 고객이 원하는 고품질의 소프트웨어를 생산하지 못하고 실패하는 경우가 많다[1]. 이러한 소프트웨어 위기에 대한 대안으로 소프트웨어 재사용 기술이 등장하였다.

소프트웨어 재사용 기술로 컴포넌트 기반 개발 방식(Component Based Development)이 대표적인데 이는 소프트웨어를 컴포넌트라는 작은 모듈로 나누어 마치 하드웨어처럼 모듈들을 결합하여 원하는 기능을 하는 소프트웨어를 개발하는 방식이다. 기존의 컴포넌트 명세는 자연의 명세나 UML과 같은 그래픽 명세가 사용된다. 이러한 명세기법은 컴포넌트 명세의 의미적인 측면을 보장해 주지는 못한다. 이는 곧 재사용 측면이 강한 컴포넌트 기반 개발 방법에서 모호성, 불완전성, 모순등으로 인한

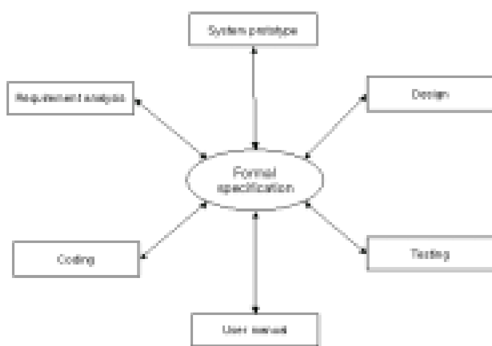
부작용을 발생 시킬 수 있다. 컴포넌트 기반 개발 방식은 결국 작은 컴포넌트 모듈이 각각이 요구사항을 만족하는 기능을 발휘해야 결국 이러한 컴포넌트의 묶음으로 이루어지는 소프트웨어 시스템도 품질을 인정받을 수 있기 때문에 각각의 컴포넌트에 대한 기능적인 정확성은 특히 중요하다. 기존의 대부분의 정형명세관련 연구들은 요구사항의 분석 및 검증에 관한 연구가 대부분으로 직접적인 구현과 관련된 연구는 부족한 실정이다.

따라서 본 논문에서는 EJB(Enterprise Java Beans) 아키텍처를 따르는 컴포넌트의 의미적인 측면을 Z 정형명세언어를 이용하여 명세하고 명세를 바탕으로 EJB를 구현하는 단계를 제시한다.

2. 관련연구

2.1 컴포넌트 명세에 관한 연구

컴포넌트의 정형명세는 개발주기의 프로토 타입 작성, 요구사항 분석, 설계, 구현, 테스트등의 대부분의 소프트웨어 생명주기에서 사용되어 진다. 이는 하나의 명세를 바탕으로 시스템의 일관성을 유지할 수 있고 정형화된 표현으로 완전성 확보 및 검증이 가능하게 된다[2].



(그림1) Formal Specification의 활용

현재의 소프트웨어 개발 프로세스는 다양한 모델을 사용하여 소프트웨어의 일관성과 완전성 검사를 위한 수단을 제공한다. 그러나 이러한 표기법들은 기본적으로 자연어 문장이나 다이어 그램에 기초하기 때문에 부정확함은 물론 해석에 따른 애매함이 존재하게 된다. 자연어로 표현된 명세에서 발생할 수 있는 대표적인 문제점은 다음과 같다.

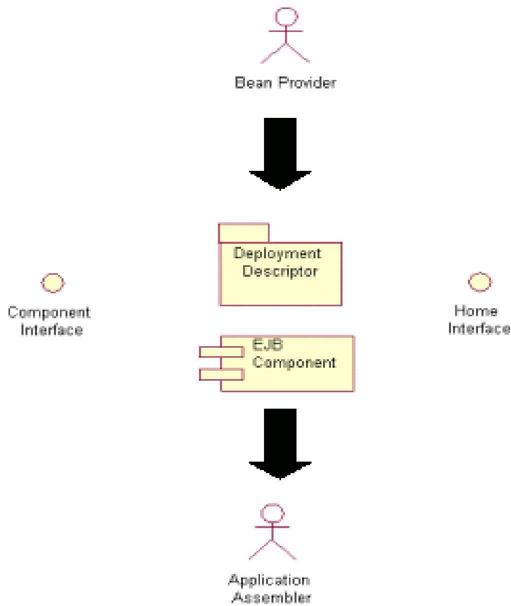
'학생들의 성적을 정렬하여 반환한다'

위의 문장에서 정렬의 의미가 오름차순인지 내림차순인지 의미가 언급되어 있지 않고 주변의 문맥으로 그것이 일련의 숫자라는 것은 추론할 수 있지만 정확한 의미를 알아내기가 어렵다. 위의 경우 자연어의 모호성(ambiguity)가 발생하게 된다. 또한 언제 성적이 반환되는가라는 문제가 발생한다. 이처럼 자연어 명세에는 완전하게 명세되지 않고 많은 의미를 불완전하게 남겨두는 것을 허용하게 돈다. 이 때 자연어의 불완전성(imcompleteness)가 발생하게 된다. 또한 요구사항의 문장이 서로 경쟁적인 문장을 포함하는 경우 모순(contradiction)을 발생하게 된다. 이러한 문제점은 소프트웨어의 복잡도가 증가하고 신뢰도에 대한 요구가 증가하는 상황에서 비정형적인 기법의 한계로 인한 에러의 가능성을 증가시킨다[1].

2.2 EJB

EJB(Enterprise Java Beans)는 분산 컴포넌트 환경에서 사용할 수 있는 서버 측 컴포넌트 모델이다. EJB 에서는 Java Servlet과 Java Server Page와 같은 Web Component와 Java Class파일의 묶음으로 구성된 Enterprise Beans의 두 가지 형태의 컴포넌트를 제공한다[7].

결국 이러한 컴포넌트들은 J2EE(Java2 Enterprise Edition) Application으로 구성되고 J2EE Server에 배치되어 그 기능을 제공하게 된다. (그림 2)와 같이 실제로 사용자의 요구 사항으로부터 Bean Provider가 컴포넌트를 작성해서 컴포넌트와 관련된 컴포넌트 인터페이스와 홈 인터페이스 배치 디스크립터, EJB 컴포넌트를 Application Assembler에게 전달 하고 Assembler가 조립을 하게 된다.



(그림 2) EJB의 조립과 배치

배치 디스크립터는 컴포넌트의 정보를 XML 형식으로 작성하면 이는 컴포넌트의 배치 정보를 Assembler에게 전달한다. 문서의 양식은 (그림 3)과 같다[8].

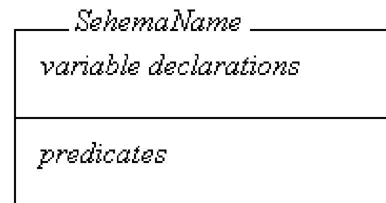
```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar (View Source for full doctype...)>
- <ejb-jar>
    <description>no description</description>
    <display-name>CartJAR</display-name>
  - <enterprise-beans>
    - <session>
        <display-name>CartBean</display-name>
        <ejb-name>CartBean</ejb-name>
        <home>CartHome</home>
        <remote>Cart</remote>
        <ejb-class>CartEJB</ejb-class>
        <session-type>Stateful</session-type>
        <transaction-type>Bean</transaction-type>
      </session>
    </enterprise-beans>
  </ejb-jar>
```

(그림 3) Deploy Descriptor

따라서 Bean Provider는 컴포넌트의 요구사항을 정확히 파악할 필요가 있고 Bean Provider가 컴포넌트를 작성하는데 있어서 컴포넌트의 정형명세를 입력 산출물로 사용할 경우 모호성, 모순, 불완전성 등의 문제점을 해결할 수 있게 된다.

2.3 Z의 명세를 통한 EJB의 구현

Z 명세는 Schema로 구성되어 있다. Schema의 구조는 (그림 4)와 같이 Schema Name과 Variable declaration, Predicates로 구성된다[3].



(그림 4) Z Schema의 구조

Schema는 크게 변수 사이의 관계를 명세한 State schema와 State Schema의 상태 변화를 표현하는 Operator schema, 초기 상태를 표현하는 Initialization Schema, 에러 상태를 표현하는 Error Schema로 분류할 수 있다[3].

Bean의 경우 기본적으로 EJB Package안에 들어있는 클래스를 사용하여 구현을 하는데 크게 Bean의 인터페이스를 정의하는 Interface 부와 Interface에 선언된 각 메소드를 구현하는 클래스로 구성되어 있다[7].

본 논문에서 제안하는 Z의 명세를 이용한 Bean의 구현 단계는 다음과 같다.

[단계 1] Z 명세에서 각 Schema를 State Schema, Initialization Schema, Operator Schema, Error Schema를 분류한다.

[단계 2] Bean에서 제공해야 할 인터페이스를 하나 이상의 Operator Schema와 Error Schema를 조합하여 정의한다.

[단계 3] 정의된 Interface를 통하여 Bean의 Inteface를 선언한다.

[단계 4] Operator Schema와 Error Schema의 Pre-condition과 Post-condition을 식별한다.

[단계 5] Pre-condition 조건을 판별하여 Post-condition을 만족하는 Bean의 템플릿

을 구현한다.

[단계 1]은 요구사항을 명세한 Z 명세에서 각 Schema의 구분을 짓는다. 이 때 Schema는 위에서 설명한 State Schema, Initialization Schema, Operator Schema, Error Schema의 4개 중 하나로 분류할 수 있다. [단계 2]는 Bean에서 제공해야할 인터페이스를 Schema의 조합으로 정의한다. 이때, 하나의 인터페이스에 하나의 Operator Schema로 정의되는 경우, 하나의 인터페이스에 복수개의 Operator Schema로 정의되는 경우, Operator Schema와 Error Schema의 조합으로 정의되는 경우의 3가지가 존재한다. [단계 3]은 [단계 2]에서 정의된 인터페이스를 사용하여 Bean의 인터페이스를 선언한다. 이 때, Bean의 Remote Interface가 정의된다. [단계 4]는 Operator Schema와 Error Schema의 Pre-condition과 Post-condition을 식별한다. 이 때, Pre-condition은 Schema가 실행되기 이전에 만족해야할 조건이고 Post-condition은 Schema의 실행 후에 만족해야 할 조건이다. [단계 5]는 Bean의 Interface에서 선언된 Interface의 구현부로 이 때, [단계 2]에서 정의된 Interface의 정의에 따라, 하나의 Operator Schema로 Interface가 정의되는 경우, 복수 개의 Operator 로 정의되는 경우, Error와 Operator Schema의 조합으로 정의되는 경우의 각각의 코드 템플릿은 (그림 5), (그림 6), (그림 7)과 같다.

```
public return_type interface_name(parameter_list)
{
        if(Pre_condition)
        {
                Post_condition;
        }
}
```

(그림 5) 하나의 Operator Schema로 정의된 인터페이스 코드의 템플릿

```
public return_type interface_name(parameter_list)
{
        if (Pre_condition 1)
        {
                Post_condition 1;
        }
        else if (Pre_condition 2)
        {
                Post_condition 2;
        }
        ...
        {
        else if (Pre_condition n)
        {
                Post_condition n;
        }
}
```

(그림 6) 하나이상의 Operator Schema로 정의된 인터페이스 코드의 템플릿

```
public return_type interface_name(parameter_list)
{
        try
        {
                if (Pre_condition_Error)
                {
                        throw new ErrorClass();
                }

                else (Pre_condition_Normal)
                {
                        Post_condition_Normal;
                }
        {
        catch (ErrorClass e)
        {
                Post_condition_Error
        }
}
```

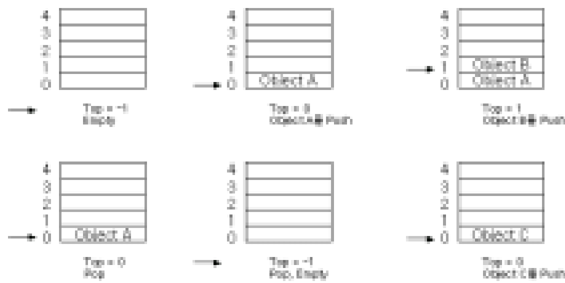
(그림 7) Operator Schema와 Error Schema의 조합으로 정의된 인터페이스 코드의 템플릿

3. 사례연구

3. 1 Stack의 Z명세

본 연구에서는 자바의 Object 형의 객체를 Push와 Pop를 수행하는 Stack을 명세 한다. Stack의 일반적인 자연어 명세는 다음과 같이 나타낼 수 있다.

"스택은 원소의 삽입과 삭제연산이 리스트의 한 쪽 끝에서만 발생하도록 제한되어 있는 특별한 자료구조이다. 스택에서 처리할 수 있는 원소는 가장 나중에 들어온 원소뿐 이며 이 때, 스택의 삽입 연산을 Push라 부르고 삭제연산을 Pop라 부른다. 삽입과 삭제의 두 연산이 스택의 같은 한 쪽 끝에서만 발생하므로 삭제되는 원소는 삽입된 반대 순서의 원소들이다. 즉, 먼저 삽입된 것이 나중에 삭제된다. 이러한 방식을 후입선출(Last-In First Out, LIFO)라 한다."



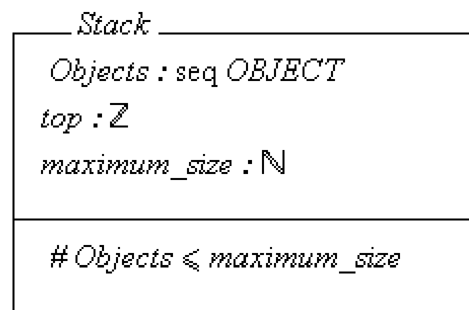
(그림 8) Stack의 Push, Pop 연산과정

이러한 스택 연산을 자연어 명세로 표현하면 다음 <표 1>과 같다.

<표 1> Stake 연산의 비 정형 명세

연산명	입력값	처리	출력값
Push	Object	Object를 저장할 배열에 입력으로 들어온 입력값 Object를 추가하고 top의 값을 1증가 한다.	
Pop		Stake의 top이 가리키는 원소를 출력하고 top의 값을 1감소한다.	Object
IsFull		Stake에 원소가 다 채워져 있는 경우 'StakeFull'의 response를 출력한다.	response
IsNotfull		Stake에 원소가 다 채워져 있지 않은 경우 'StakeNotFull'의 response를 출력한다.	response
IsEmpty		Stake에 원소가 존재하지 않는 경우 'StakeEmpty'의 response를 출력한다.	response
IsNot Empty		Stake에 원소가 존재하는 경우 'StakeNotEmpty'의response를 출력한다.	response
Overflow		Stake에 원소가 다 채워져 있는 경우에 다시 원소를 Push연산이 발생하는 경우 'Stake Overflow'의 response를 출력한다.	response
Underflow		Stake에 원소가 존재하지 않는 경우에 원소를 Pop연산이 발생하는 경우 'Stake Underflow'의 response를 출력한다.	response

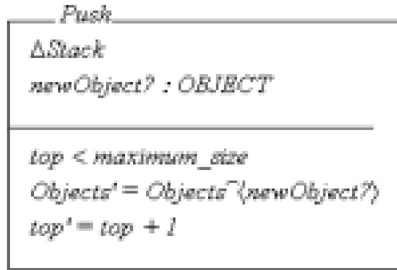
Stake의 형태를 Z 스키마를 통해 표현하면 그림과 같이 표현할 수 있다.

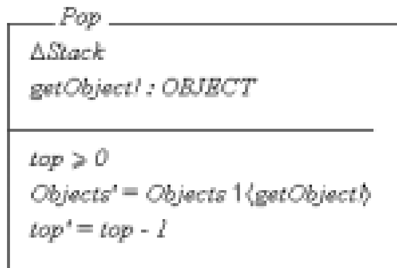


(그림 9) Stake의 스키마 표현

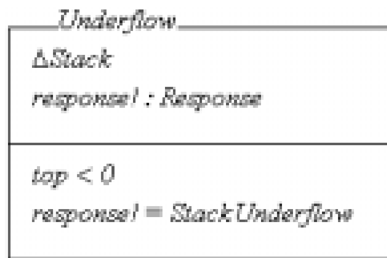
스택에 저장될 수 있는 데이터의 형태는 Java의 Object 클래스 타입으로 OBJECT로 표현되었고 스택의 마지막 위치와 최대 크기를 지정하기 위해 정수형 타입의 top와 maximum_size가 선언되었고 이때 maximum_size는 100으로 스택의 크기를 고정하고 스택의 데이터 크기는 100을 초과 할 수 없음이 표현되었다.

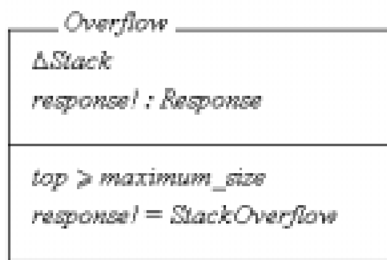
Stake의 연산이 Push와 Pop은 (그림 10)과 같이 표현할 수 있는데 Push의 경우는 새로운 OBJECT 타입의 입력변수 newObject?를 입력으로 하여 기존의 리스트에 추가하고 이때 top의 값은 1증가하게 된다. Pop의 경우는 OBJECT타입의 getObject!를 출력하게 되고 이때는 반대로 top의 값이 1 감소하게 된다.





(그림 10) Push와 Pop의 스키마 표현





(그림 11) Underflow와 Overflow의 스키마 표현

Stake에 자료가 하나도 없는 경우에 Pop를 수행하게 되는 경우와 이미 최대 사이즈의 100개를 갖는 상태에서 Push를 수행하는 경우 각각, Underflow와 Overflow가 발생하게 되고 이러한 상태를 Z 스키마로 표현하면 위의 (그림 11)과 같이 나타낼 수 있다. Stake에 담겨있는 Object의 Push 연산과 Pop연산은 다음과 같이 정의된다.

$$ObjectPop \mathrel{\widehat{=}} Pop \vee Overflow$$
$$ObjectPush \mathrel{\widehat{=}} Push \vee Underflow$$

(그림 12) Pop와 Push Interface 정의

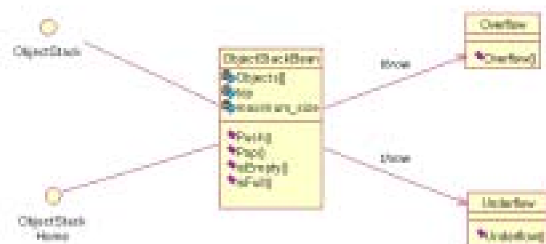
3. 2 Stack의 구현

[단계 1]를 적용하여 Stack의 Z 명세에 나타난 스키마를 분류하면 다음과 같다.

<표 2> Schema의 분류

Schema Name	Schema Type
Stack	State Schema
push	Operator Schema
Pop	Operator Schema
Underflow	Error Schema
Overflow	Error Schema
IsFull	Operator Schema
IsNotFull	Operator Schema
IsEmpty	Oterator Schema
IsNotEmpty	Oterator Schema

<표 2>에서 분류된 State Schema의 겨우 Bean의 Property에 해당이 되고 Operator Schema의 경우 Interface에서 선어되고 Bean에서 구현되어야 할 Method에 해당된다. 또한 Underflow와 Overflow는 예외를 처리하는 Exception Class에 매핑을 시켜 Bean을 구성한다.



(그림 13) StachBean의 구조

[단계 2]를 적용하여 StackBean의 인터페이스는

Push, Pop, IsFull, IsEmpty인터페이스는 다음 (그림 14)과 같이 정의 된다.

$$ObjectPush \triangleq Push \lor Overflow$$
$$ObjectPop \triangleq Pop \lor Underflow$$
$$StackIsFull \triangleq IsFull \lor IsNotFull$$
$$StackIsEmpty \triangleq IsEmpty \lor IsNotEmpty$$

(그림 14) StackBean의 Interface정의

[단계 2]의 과정을 통해 나온 Interface정의를 토대로 [단계 3]은 Bean의 Interface를 선언 한다.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface ObjectStack extends EJBObject
{
    public Object Pop() throws RemoteException;
    public void Push(Objet obj) throws RemoteException;
    public String IsEmpty() throws RemoteException;
    public String IsFull() throws RemoteException;
}
```

(그림 15) Bean의 Interface선언

[단계 4]를 적용하여 Stack의 Z 명세를 바탕으로 Pre-condition과 Post-condition을 표로 정리하면 아래 <표 3>과 같이 나타난다.

<표 3> 각 스키마의 Pre-condition과 Post-condition

Schema	Precondition	PostCondition
Push	top<maximum_size	top' = top + 1 Objects' =Objects ∧ <gerObjert?>
Pop	top≥0	top' = top - 1 Objects' = Objects ＼ <getObject!>
Underflow	top<0	response! = StackUnderflow
Overflow	top ≥ maximum_size	response! = StackOverflow
IsFull	top=maximum_size -1	response! = StackFull
IsNotfull	top<maximum_size -1	response! = StackNotFull
IsEmpty	top=-1	response! = StackEmpty
IsNotEmpty	top≥0	response! = StackNotEmpty

[단계 5]에서는 단계 2,3,4에서 나온 결과물을 토대로 Bean의 Interface를 구현하는데 먼저 Pre-condition을 판별하여 조건에 맞으면

Post-condition을 수행하는 문장을 구성한다. 인터 페이스 정의에 Schema의 개수와 종류에 따라 앞서 제시된 (그림5), (그림 6), (그림 7)중의 하나의 템플릿 코드를 사용하게 된다. 예를 들어 Stack의 Pop Interface의 경우 Pop Operator Schema와 Error를 표현한 Underflow Schema의 조합으로 이를 표현한 코드의 양식은 (그림 16)과 같이 표현된다.

```
public Object Pop()
{
    Object getObject;

    try
    {
        if (top < 0)
        {
            throw new Underflow();
        }
        else
        {
            getObject = Objects[top];
            top--;
        }
    }
    catch (Underflow e)
    {
        getObject = "StackUnderflow";
    }

    return getObject;
}


public return_type interface_name(parameter_list)
{
    try
    {
        if (Pre_condition_Error)
        {
            throw new ErrorClass();
        }
        else
        {
            Post_condition_Normal;
        }
    }
    catch (ErrorClass e)
    {
        Post_condition_Error;
    }
}
```

(그림 16) 구현된 POP interface와 사용된 템플릿

4. 결 론

컴포넌트는 모듈화된 소프트웨어 덩어리로 컴포넌트 기반 시스템을 구성하는 요소가 된다. 본 논문에서는 이러한 컴포넌트 명세에 있어서 정형 언어인 Z를 이용하여 Stack의 명세를 유도하였고 Z명세를 바탕으로 EJB 컴포넌트를 구현하는 예를 제시하였다 Stack의 예의 경우 일반적인 Push, Pop, Empty 체크, Full 체크를 수행하였고 Push와 Pop 연산도중에 발생할 수 있는 Overflow와 Underflow의 명세를 포함하였다. 이들 연산에 대한 Z 스키마는 다시 오퍼레이테 스키마, 에러 스키마로 분류되어 에러 스키마의 경우 Java의 Exception 처리를 사용하여 구현되었고 다시 인터페이스로 정의된 Push(), Pop(), IsEmpty(), IsFull() 메소드를 각 Z 명세의 Pre-condition과 Post-condition을 사용하여 구현하였다.

결론적으로 Z의 명세는 스키마의 분류, 인터페이스의 정의, Post-condition과 Pre-condition의 정의 단계를 거치고 정제를 통해서 EBJ의 인터페이스, 예외 클래스, 메소드 구현에 사용할 수 있음을 보였다. 추후 연구를 통하여 자동화된 Z명세의 EJB 템플릿 코드 변환과 여러 Bean이 조합된 시스템의 명세에 관한 연구가 필요하다.

참 고 문 헌

[1] 장종표, “컴포넌트 품질 향상을 위한 정형 명세 프로세스”, 전남대학교 박사학위 논문 2002, 2.

[2] Jonathan jacky, "the way of Z", Cambridge University Press. 1997

[3] J. B. Wordsworth, "software Development with Z", Addison-Wesley, 1993.

[4] David Rann, John Turner and Jenny Whitworth, "Z: A Beginger's Guide". Chapman & Hall, 1994.

[5] Bryan Ratcliff, "Introducing Specification Using Z", McGraw-Hill, 1994.

[6] Paul Tremblett, "Instant Enterprise JavaBeans" McGraw-Hill, 2001.

[7] ED Roman, "Mastering Enterprise JavaBeans" Wiley, 1999.

[8] Sun Microsystems, Enterprise Javabeans Specification, Version 2.0

마 대 성

1994 호남대학교 전산통계학과 졸업(이학사)

1996 전남대학교 대학원 전산통계학과 졸업(이학석사)

2000 전남대학교 대학원 전산통계학과 졸업(이학박사)

1997~2000 광주교육대학교 전산교육과 조교

2003~현재 광주교육대학교 전산교육과 전임강사

관심분야 : 컴퓨터 교육, ICT, 프로그래밍 언어, 소프트웨어공학, 모바일 인터넷

E-mail : dsma@gnue.ac.kr