클래스의 인터페이스와 구현부의 분리를 지원하기 위한 객체지향 프로그래밍 언어의 확장

(Extension of Object-Oriented Languages with the Separation of Class Interface and Implementation)

조 은 선 † 한 상 영 †† 김 형 주 †††
(Eun-Sun Cho) (Sang-Yong Han) (Hyoung-Joo Kim)

요 약 객체지향 프로그래밍 언어에서 클래스는 객체의 인터페이스와 객체의 구현을 모두 기술한다. 클래스의 인터페이스와 구현부는 항상 일대일로 대응되지 않기 때문에, 이 두 가지가 각각 분리되어 기술되어야 한다는 지적들이 있다. 이를 지원하는 언어 모델에 대한 연구도 진행되어 왔으나, 인터페이스에만 주로 치중하여 클래스 구현부가 가지는 기능들에 제약을 두는 경우가 대부분이고, 사용자가 의도하지 않은 인터페이스-구현부 관계를 야기시키는 등의 단점이 있었다.

본 논문에서는 각 인터페이스와 클래스 구현부 모두의 기능을 보존하며, 의도하지 않은 부수작용(side-effect)이 발생하지 않는 새로운 분리 모델을 제안하고 그 타입 공간을 형식화하여 기술한다.

Abstract In most object-oriented programming languages, the notion of a class is responsible for both interface and implementation of its interfaces. However, interface and implementation of an object should not be described in a single unit, because interfaces and implementations are not always isomorphic. The existing approaches on the separation of interface and implementation are not satisfactory, because of their unfair treatment of interface and implementation by a class, as well as the inadvertent insertion of interface-implementation relationships.

This paper presents a new systematic separation model for interface and implementation of a class, where entire functionalities of both interface and implementation are preserved without causing any side-effect.

1. 개 요

클래스가 가지는 슬롯(slot)이나 메쏘드(method)들은, 클래스 외부에 제공되는 것과 클래스 내부에서 구현을 위해 사용되는 것으로 분류될 수 있다. '인터페이스(interface)'란 한 클래스가 외부에 제공하는 메쏘드의 명세와 외부에서 볼 수 있는 데이타들을 말한다. C++[1]이나 *Eiffel*[2], *Smalltalk*[3] 등의 많은 객체지향 프로그래밍 언어들은, 인터페이스를 별도로 지원하기 보다는, 클래스 정의 내부에서 슬롯이나 메쏘드 선언에 키워드(예, 'public')등을 덧붙이거나[1,2], 메쏘드만을 인터페이스로 허용하는 방식[3]으로 구분하고 있다.

그러나, 인터페이스와 클래스의 구현부를 분리하여 별도로 둘 경우, 여러 잇점이 있다는 연구가 있어 왔다[4,5,6,7,8,9,10,11,12,13]. 이들 연구에 의하면, 인터페이스가 클래스와 별도로 존재하지 않는 경우, 프로그래밍 언어의 표현 능력과 타입 시스템의 유연성(flexibility)이 제한된다는 것을 알 수 있다[4,5,8,9,11,12,13].

· 본 연구는 한국 과학재단에서 부분적으로 지원을 받아 수행되었음(과제번호 · 94-2180, 과제명 · 'ODMG 표준 객체모델을 근간으로 C++을 확장한 객체지향 데이타베이스 프로그래밍 언어에 관한 연구')
· 본 연구는 통상 산업부에서 지원하는 객체지향 데이타베이스 'SOP(SNU OODBMS Platform)' 개발 과제의일부인 객체지향 데이타베이스 프로그래밍 언어 개발 과정 중 수행되었음.
† 학생회원 : 서울대학교 컴퓨터공학과
†† 종신회원 : 서울대학교 전산과학과 교수
††† 종신회원 : 서울대학교 컴퓨터공학과 교수
논문접수 : 1996년 4월 2일
심사완료 : 1996년 10월 28일

이와 같은 문제를 극복하기 위해, 기존의 연구와 전통적인 타입 이론을 토대로하여, 인터페이스와 클래스의 구현부를 분리 정의를 지원하는 언어들이 등장하였으며[5,13], 또, 이를 지원하고자 기존의 객체지향 프로그래밍 언어를 확장하려는 노력 또한 있어왔다[4,9,12,14]. 그러나, 이들을 좀 더 자세히 살펴본다면, 사용자의 의도를 충분히 반영하고, 클래스 구현부의 기능을 제한하지 않으면서, 이들의 분리를 지원하는 프로그래밍 언어 모델은, 아직 존재하지 않는 것을 알 수 있다.

본 논문에서는 인터페이스와 클래스 구현부의 분리를 보다 완전하게 지원하기 위해 새로운 프로그래밍 언어 모델을 제안한다. 본 논문에서 제안된 모델은 다음과 같은 사항들을 다루는 것을 목표로 한다.

- 본 모델을 가지는 프로그래밍 언어를 새로 정의할 수 있을 뿐아니라, 본 모델을 통해 기존의 객체지향 언어에 적용되어 확장될 수 있도록 한다.
- 사용자가 클래스의 인터페이스와 클래스 구현부를 분리된 모듈로서 지정할 수 있도록 한다.
- 인터페이스와 클래스 구현부의 관계는 사용자의 명시를 근간으로 판단한다.
- 인터페이스와 클래스 구현부 모두가 타입 도메인에 포함되어, 타입을 나타내는 단위로 사용될 수 있다.
- 인터페이스와 클래스 구현부 각각에 계층 구조가 존재하며, 이들 계층 구조들이 반영되어 만들어지는 새로운 인터페이스-클래스 구현부 관계들도 정의한다.

즉, 본 논문에서 제안된 모델은 기존의 대다수의 분리 모델들과는 달리, 사용자가 클래스의 인터페이스와 클래스 구현부간의 관계를 지정할 수 있도록 하고 있고, 인터페이스 뿐 아니라 클래스 구현부가 가지는 기능도 함께 고려하고 있다. 또한 본 논문에서는 기존의 타입 이론을 토대로 하여 제안된 모델의 타입 공간을 형식화하여 기술하였다.

본 논문의 구성은 다음과 같다. 2절에서는 클래스의 인터페이스와 구현부의 분리의 필요성을 예를 들어 설명한다. 3절에서는 본 논문 전체에서 사용되는 기본 용어들과 개념들을 정의한다. 4절에서는 클래스의 인터페이스와 클래스 구현부를 분리한 프로그래밍 언어 모델을 소개하고, 5절에서는 분리된 인터페이스와 클래스 구현부를 모두 가지는 타입 공간에 대해 기술한다. 6절에서는 기존의 관련 연구들과 비교 고찰하며, 7절에서 결론을 맺는다.

2. 인터페이스와 구현부 분리의 필요성

본 연구의 동기를 보다 명확히 하기위해, 인터페이스와 구현부 분리의 필요성을 다음과 같은 예를 들 수 있다.

'Set'이나 'Bag' 같은 군집 타입(aggregation type)은 공통적인 인터페이스로 'insert'나 'delete'등의 메쏘드의 명세를 가진다. 이러한 공통적인 메쏘드를 뽑아내어 추상 클래스(abstract class)인 'Collection'을 만들고, 'Set'과 'Bag'은 이로부터 상속되도록 하는 것이 직관적이다. 그림 1에서의 실선은 바로 이러한 상속 관계를 나타낸다. 그러나, 실제로 이를 C++등의 언어로 구현하는 경우 코드의 재사용을 위해, 'Bag'은 'BPlusTree'등의 클래스에서 상속받고, 'Set'이 다시 'Bag'에서 상속받을 가능성이 높다. 이것은 그림 1에서 점선으로 표시되어 있다. 따라서, 결과적인 클래스 계층 구조는 두 가지 모두를 반영하기 위해 그림과 같이 서로 관련 없는 두 개의 계층 구조가 섞인 다중 상속으로 표현되게 된다[4,8,9,10,12,13]. 즉, 클래스 'Collection'이 포함된 계층 구조는 인터페이스만의 상속을 나타내는 것으로 볼 수 있으며, 클래스 'BPlusTree'를 포함한 트리는 클래스의 구현을 재사용하기 위한 것이다.

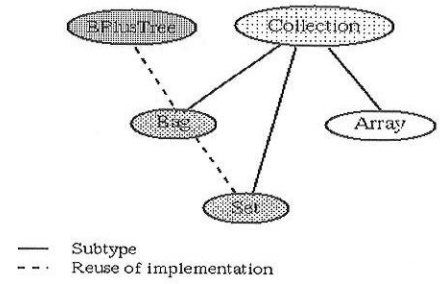

그림 1 두 개의 계층 구조의 공존

더우기 여기서, 클래스 'Set'만을 고려해볼 때에, 'Set'의 구현 방법이 B+트리, 해쉬(hash), 비트 스트링(bit string)등 여러 가지가 가능하기 때문에, 추상 클래스 'Set'을 만들어 클래스 'BPlusTreeSet'과 클래스 'HashSet'등이 이를 상속할 수도 있다. 그러나, 이러한 계층 구조가 그림 1에 합류되게 되면 클래스 그래프가 상당히 복잡해지거나, 경우에 따라서는 표현이 불가능해질 수도 있다.

따라서, 클래스의 인터페이스와 나머지 구현부를 분리시키고, 계층 구조도 하위 타입 다형성을 위한 것과

코드 재사용을 위한 것을 별도로 가질 수 있다면, 위와 같은 계층 구조의 뒤섞임을 피할 수 있으므로, 클래스 계층 구조의 단순화가 가능하다[4,8,9,10,12,13].

3. 기본 용어 및 개념

본 모델은 인터페이스와 클래스 구현부를 분리하는 것을 중심으로 정의되었지만, 분리 기능을 제외하면 나머지 부분에서는 일반 객체지향 프로그래밍 언어와 동일한 기능을 제공해야한다.

이 부분을 본 논문에서는 '기반 언어' 부분이라 부른다. '기반 언어' 부분에서는 가장 많이 공통적으로 받아들여지고 있는 객체지향 프로그래밍 언어들의 기능들을 제공하는 것을 원칙으로 한다. 따라서, 기반 언어는 '클래스'로 불리는 객체 생성자(object constructor)를 지원하며, 클래스들로 이루어지는 계층 구조를 사용자가 정의할 수 있도록 한다. 또, 정적인 타입 검사가 수행되며, C++이나 Smalltalk과 비슷하게 단일 메쏘드 추출(single-method dispatch)[3]과 동적인 self지정 규칙(open-self)[15]을 가정하고 있다.

본 모델에서는 클래스의 인터페이스와 구현부의 분리를 위해, 기반 언어에서 제공되는 클래스 외에도 두 개의 객체 생성자를 지원한다. '인터페이스(interface)'는 공용 슬롯과 메쏘드 시그니쳐(signature)로 구성된 객체 생성자이며, '클래스 구현부(implementation)'는 전용 슬롯과 메쏘드를 모두 포함하는 객체 생성자로 자신의 인터페이스를 명시할 수 있다. 인터페이스 'I'가 클래스 구현부 'M'을 '구현한다(implement)'고 하는 것은, 'M'에 의해 생성된 객체들은 'I'의 객체로 간주되어도 좋다는 의미이다. 클래스는 인터페이스와 클래스 구현부가 분리되어 정의될 수도 있지만, 일반적인 객체지향 프로그래밍 언어의 클래스처럼 함께 정의될 수도 있다. 이러한 성질에 의해 본 모델은 새로운 프로그래밍 언어를 정의하는데 뿐 아니라, 기존의 객체지향 프로그래밍 언어를 확장시키는데에도 적용될 수 있다.

4. 인터페이스와 클래스 구현부의 분리 모델

정의 구문

클래스로 부터 분리된 인터페이스와 클래스 구현부도 각기 기존의 클래스와 비슷한 구조이기 때문에, 인터페이스와 클래스 구현부의 분리는 클래스 하나를 클래스 두 개로 나누어 표현하는 개념으로 볼 수 있다. 먼저, 인터페이스는 사용자에게 공개되는 부분으로써 클래스의 일부를 나타낸다. 본 모델에서는 클래스 구현부는 클래스 구조 자체를 그대로 사용하여 나타내기로 하고, 인터페이스를 위한 구조만을 새로 도입하고 있다.

다음은 C++과 비슷한 문법으로 나타내어진 인터페이스 정의의 예이다.

```
interface BLOB { ... };
interface Image{
    BLOB * content();
    int show_image();
    ...
};
```

인터페이스를 구성하는 슬롯이나 메쏘드의 타입-명세에는 인터페이스와 기본 타입 외에 다른 생성 단위가 사용될 수 없다. 인터페이스 'Image'는 인터페이스 이름인 'BLOB'과 기본 타입인 'int' 만으로 슬롯과 메쏘드 명세가 정의되었다. 즉, 인터페이스의 정의는 '닫혀있다(closed)'고 볼 수 있다.

클래스 구현부는 기존의 클래스와 거의 비슷하게 정의되나, 어떤 인터페이스를 구현하고 있는지를 'implements' 키워드 다음에 명시하는 점이 특징이다. 'implements'에 의해 명시된 인터페이스에서 이미 나타난 슬롯이나 메쏘드의 선언은 클래스 구현부에 다시 나타날 필요가 없다[1]. 클래스 구현부를 나타내는 클래스는 C++과 비슷한 문법으로 다음과 같이 나타낼 수 있다.

```
class BitImage{
    ....
        implements Image;
    BitString * toBitstring();
    Image * nextImage();
    ...
};
```

만일 'BitString'이 클래스 구현부를 나타내는 이름이라면, 인터페이스 'Image'와 클래스 구현부 'BitString'이 모두 클래스 구현부 'BitImage'의 정의를 위해 사용되었음을 알 수 있다.

상속 원칙

클래스 상속 관계는 새로운 클래스의 정의 구문(syntax)에 의해 나타내어진다. 즉, 기존의 클래스에서 유도된 새로운 클래스의 정의는, 재사용하고자하는 기존의 클래스의 이름과 새로운 클래스와 기존의 클래스의

1) 이것은 C++, Smalltalk, Eiffel등의 클래스 상속 방식과 비슷하다.

정의의 차이에 의해 기술된다. 이러한 이어 붙이기식 상속 관계(inheritance-by-concatenation)가 상/하위 타입의 관계를 결정하는데에 그대로 사용되는 경우, 타입 정확성(type-correctness)을 쉽게 보장하게 된다[16]. 그러나, 이러한 상속 방식으로만 지원되는 하위 타입 관계는 너무 제한적이기 때문에, 객체지향 프로그래밍 언어에서는 이보다 조금 유연성있게 변형된 상속 원칙들이 나타났다[1,5,13]. 즉, 기본적으로는 이어 붙이기식 상속 관계이지만, 경우에 따라서는 슬롯이나 메쏘드가 새 클래스에서 재정의 되는 것도 허용하는 것이다. 단, 상속 관계에 따라 하위 타입을 안전하게 결정하기 위해서는, 재정의를 다음과 같은 조건이 만족되는 경우로 한정하여 허용한다.

- 슬롯의 타입-명세의 재정의는, 새 타입-명세에 나타나는 타입이 원래의 타입-명세의 타입의 직접 또는 간접 하위 타입[2]으로 재정의하는 것만 허용된다.
- 메쏘드의 타입-명세가 재정의는, 새 결과 타입-명세에 나타나는 타입이 원래 타입에 정의될 때 나타난 결과 타입-명세가 가지는 타입의 직접 또는 간접 하위 타입으로 재정의되고, 새 인자 타입-명세들에 나타나는 타입들이 원래의 인자 타입-명세에 나타나는 타입들의 직접 또는 간접 상위 타입으로 재정의되는 경우만 허용한다.

이러한 재정의 제약은 반-변환(contravariance)이라는 이름으로 널리 사용되고 있으며, 이에 따라 정의된 상속에 의해 하위 타입 관계가 결정되는 경우에, 타입-정확성(type-correctness)이 보장되는 것이 이미 밝혀진 바 있다[13,16,17,18,19]. 본 모델에서는, 인터페이스와 클래스 구현부 각각에 계층 구조가 존재하며, 두 가지 모두 반-변환 방식으로 이루어지도록 정의한다.

클래스 구현부 상속의 의미

인터페이스의 계층구조는 사용자가 의도하는 서브 타입관계를 의미한다. 클래스 구현부의 상속은 코드의 재사용을 위한 것인데, 실제로는 클래스 구현부의 상속은 인터페이스와의 관계와 결합되어, 여러가지 성격을 가질 수 있다. 여기서는 사용자의 의도와 가장 잘 일치한다고 판단되는 두 가지를 골라 도입하였다. 즉, 다음과 같다.

- 클래스 구현부를 상속할 때, 상위 클래스 구현부[3]가 '구현'하는 인터페이스도 함께 상속 받는다.
- 클래스 구현부를 상속할 때, 상위 클래스 구현부가 '구현'하는 인터페이스는 상속 받지 않는다.

이것을, 본 모델에서는 키워드 'implement'에 의해 구분한다. 즉, 키워드 'implement'로 자신만의 인터페이스를 이미 명시하고 있는 클래스 구현부가 다른 클래스 구현부를 상속 받을 경우, 사용자의 의도는 오직 그 상위 클래스 구현부의 슬롯이나 메쏘드 코드의 재사용을 위해 상속 받는 것으로 볼 수 있다. 따라서, 상위 클래스 구현부의 인터페이스가 함께 상속 되지 않는것이 옳다.

만일 어떤 클래스 구현부가 상위 클래스 구현부에서 상속을 받고 implement 키워드가 없이 생성되었다면, 이것은 특정 인터페이스들을 기존의 구현부보다 구체적으로 구현하는 새로운 구현부를 정의하려는 의도로 해석할 수 있다. 따라서, 상위 클래스 구현부가 '구현'하는 인터페이스들을 새로운 클래스 구현부도 '구현'하는 것으로 간주한다.

인터페이스과 클래스 구현부의 유효한 바인딩

객체가 지정되거나 정의될 때, 분리된 인터페이스와 클래스 구현부가 모두 개입되어 결합되는 경우를 생각할 수 있다. 가령 인터페이스 'Image'와 클래스 구현부 'BitImage1', 'BitImage2'는 다음과 같은 객체 생성 구문에 사용될 수 있다.

```
Image * p = new BitImage1 ;
...
p = new BitImage2 ;
```

위의 인터페이스 포인터 'p'가 가리키는 객체는 인터페이스와 클래스 구현부 둘다에 동시에 관련되게 된다. 컴파일러는 이와 같은 구문에서 BitImage1이나 BitImage2가 Image를 '구현(implement)'하는 지를 검사하여, 만일 구현하지 않는 관계라면 오류가 발생한 것으로 간주하게 된다. '구현' 관계는 먼저 사용자가 클래스 구현부를 정의할 때 명시한 implement구문에 의해 직접 결정될 수 있다. 이것을 인터페이스와 클래스 구현부가 '바인딩' 되었다고 한다. 그런데, 사용자는 인터페이스와

2) 직접 하위(상위) 타입이란 사용자에 의해 명시된 관계를 말하며, 간접 하위(상위) 타입이란 사용자에 의해 명시되지는 않았으나, 이행성(transitivity)에 의해 추론되는 관계를 말한다

3) 상위 클래스 구현부란 클래스 구현부 계층 구조에서 상위 클래스를 일컬음.

클래스 구현부를 별도로 정의하고 있기 때문에, 바인딩된 인터페이스와 클래스 구현부가 내용적으로는 구현하는 관계가 아닐 수가 있다. 이 경우 클래스 구현부의 객체가 그 인터페이스 타입으로 사용되게 될 때, 타입 안전성이 파괴된다. 따라서 컴파일러는 클래스 구현부가 정의될 때 키워드 implement에 의해 명시된 인터페이스와 다음과 같은 관계가 있는지를 먼저 확인한다. 즉, 인터페이스에서 정의된 슬롯과 메쏘드 시그니처가 클래스 구현부에도 존재하여,

- 인터페이스에서 정의된 슬롯과 메쏘드 시그니처의 타입-명세가 클래스 구현부의 것과 동일 하거나,
- 인터페이스에서 정의된 슬롯과 메쏘드 시그니처의 타입-명세와 클래스 구현부에 있는 이들의 타입-명세가 또다시 인터페이스 -- 클래스 구현부 관계에 있을 경우

그 바인딩은 유효하다. 예를 들면, 다음과 같이 유효한 바인딩이 정의된다.

```
interface Color{ ... };
class       256Color{ ... implements Color; ... };

interface Image{ ... };
interface ColorImage : Image{
    ...
    Color color();
};

class 256ColorImage {
    ...
implements ColorImage;
    256Color color();
};
```

인터페이스 'ColorImage'는 인터페이스'Image'의 하위 인터페이스이고 클래스 '256ColorImage'는 명시적으로 인터페이스 'ColorImage'에 바인딩되어 있다. 'ColorImage'는 타입-명세 'Color'를 가지는 메쏘드 'color()'를 갖고있다. 또, '256ColorImage'의 선언에서 'color()'는 타입-명세를 '256Color'로 바꾸어 재정의하고 있는데, 이것은 '256Color'가 'Color'를 '구현'하기 때문에 가능한 것이다.

구현 관계

인터페이스와 클래스 구현부의 '구현'관계는 명시적인 바인딩 뿐만아니라, 인터페이스와 클래스 구현부 각각의 계층 구조에서 영향을 받아 유추하여 설정될 수 있는, 모든 의미있는 인터페이스와 구현의 관계들을 포함한다. 먼저, 인터페이스간 상속 관계를 고려한다면, 모든 클래스 구현부는 자신이 구현하는 인터페이스들의 상위 인터페이스[4]도 구현하고 있는 것으로 볼 수 있다. 예를 들어, 클래스 구현부 'M'이 'implement' 키워드를 통하여 인터페이스 'I'를 구현하고 있다면, 'I'의 상위 인터페이스도 구현한다고 볼 수 있다.

또, 특정 조건 아래서는, 모든 클래스 구현부는 그의 상위 클래스 구현부가 구현하는 인터페이스들도 구현하고 있는 것으로 간주할 수도 있다. 즉, 클래스 구현부 'T1'이 'T2'의 상위 클래스 구현부일 때, 만일 'T1'과 'T2' 사이에 또 다른 상위 클래스 구현부 'T3'가 계층 구조상에서 있어서 implement 키워드에 의해 인터페이스에 바인드 된 형태로 존재하는 경우를 제외하면 'T2'은 'T1'의 인터페이스를 '구현'한다고 볼 수 있다.

다음은 이와 같은 구현 관계를 설명해주는 예이다.

```
persistent class I1 {};
persistent class I2 : I1{};
persistent class I3 {};

class M1 { implements I2; ... }; // implements I1 and  I2
class M2 : public M1 { };          // implements I1 and  I2
class M3 : public M2{ implements I3; ... }; // implements I3
class M4 : public M3{ };                    // implements I3
```

인터페이스 'I1'과 'I2'는 상/하위 인터페이스 관계이다. 클래스 구현부 'M1'은 명시적인 바인딩에 의해 'I2'를 '구현'하고 있다. 'M1'에서 상속받은 클래스 구현부 'M2'은 'implement' 키워드가 없으므로 역시 'I2'를 구현하고 있다. 그러나, 클래스 구현부 'M3'도 기존의 'M2'를 상속하고 있기는 하지만, 'M2'가 구현하는 인터페이스 'I2'를 구현하지는 않는다. 대신 자신이 직접 'implement' 키워드로 명시한 인터페이스 'I3'을 구현하고 있다. 클래스 구현부 'M4'도 'M3'과 마찬가지로 인터페이스 'I3'을 구현한다. 즉, 'M4'는 'M2'의 인터페이스를 상속 받을 수도 있었지만, 명시적으로 인터페이스와 바인딩 된 'M3'이 계층 구조상에서 이들 사이에 있기 때문에 위의 조건을 만족하지 못하여 허용되지 않는다. 그림 2에서는 이러한 관계를 보여주고 있다.

4) 상위 인터페이스란 인터페이스 계층 구조에서 상위 클래스를 일컬음.

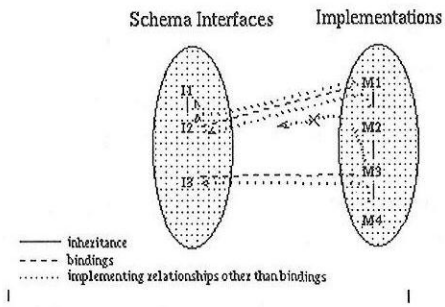

그림 2 구현 관계의 예

5. 결과적인 타입 공간

5.1 의미론적 도메인

이제, 몇 가지 정의를 써서 본 논문에서 제안하고 있는 모델을 정의한다. 먼저, 몇가지 기본적인 집합들을 정의함으로써 시작한다. int, float, char등 시스템이 정의하는 타입들의 집합은 '*B*'로 나타낸다. 인터페이스 이름의 집합은 '*DNI*'이며, 클래스 구현부의 이름의 집합은 '*DNM*'로 나타낸다. 슬롯 이름이나 메쏘드 이름의 집합은 '*DNSM*'으로 나타낸다. 각 클래스 구현부와 인터페이스는 고유한 이름을 가진다고 가정한다.

이제, 우리는 사용자 정의 인터페이스와 클래스 구현부를 정의한다.

정의 1 (인터페이스와 클래스 구현부의 집합) *사용자 정의 인터페이스 집합 I는 다음과 같은 쌍으로 정의된다 : $I = < IN, \upsilon >$. 단,*

$$IN \subseteq DNI$$
$$\upsilon : IN \rightarrow 2^{DNSM}$$

이 때, υ는 각 인터페이스 이름에서 슬롯이나 메쏘드의 집합으로 사상하는 함수이다.

사용자 정의 클래스 구현부 집합 M은 다음과 같은 튜플로 나타내어진다:$M = < IM, \delta, \theta >$. 단,

$$IM \subseteq DNM$$
$$\delta : IM \rightarrow 2^{DNSM}$$
$$\theta : IM \rightarrow 2^{DNSM}$$

이 때, δ는 각 클래스 구현부의 이름을 전용 슬롯이나 메쏘드의 집합으로 사상하는 함수이며, θ는 각 클래스 구현부의 이름을 공용 슬롯이나 메쏘드의 집합으로 사상하는 함수이다.

*IN*은 사용자 정의 인터페이스 이름의 집합이다. 그리고, 'υ(i)'에 의해, 인터페이스 '*i*'의 슬롯과 메쏘드가 구해진다. *IM*은 사용자 정의 클래스 구현부 이름의 집합이다. 클래스 구현부 '*m*'에 정의된 모든 슬롯과 메쏘드들은 '$\delta(m) \cup \theta(m)$'으로 구해진다. 본 논문에서는 분리되지 않은 상태의 일반 클래스는 클래스 구현부에 속한다고 가정한다. 또 본 절에서 부터는 $IN \cup IM$의 원소를 '클래스'라고 부른다.

이제, '타입-명세(type specification)'를 정의한다. 'REF'와 'FUNCTION'은 사용자 정의 심볼로서 타입-명세에 쓰인다. 타입-명세 자체는 다음과 같이 정의된다.

정의 2 (타입-명세(Type Specifications)) *집합 Ξ는 다음과 같이 재귀적으로 정의된다. :*

(i) 만일 $a \in B$ 이면 $a \in \Xi$ 이다.
(ii) 만일 $a \in (IN \cup IM)$ 이면 $a \in \Xi$ 이다.
(iii) 만일 $a \in \Xi$ 이면 ref(a) $\in \Xi$ 이다.
(iv) 만일 $a_0, \ldots, a_n \in \Xi$ 이면 function(n, $a_0, \ldots a_n$) $\in \Xi$ 이다.

단,

- *함수 'ref'는 $a_1 \in \Xi$ 에 대해 적용했을 때 '< REF, a_1 >'를 리턴한다.*
- *함수 'function'는, 모든 $a_i \in \Xi$ $(0 \leq i \leq n)$에 대해, $< n, a_0, \ldots, a_n >$에 적용했을 때 '< FUNCTION, n, $a_0, \ldots, a_n$ >'를 리턴한다. 이 경우, a_0 는 함수의 결과 타입-명세를 의미하며, $a_1, \ldots a_n$는 함수의 인자 타입-명세를 의미한다.*

Ξ는 변수나 슬롯, 메쏘드 등에 사용되는 타입-명세를 의미한다. 현재는 복합적 타입-명세(constructed type)에 타입 포인터와 함수 타입만을 허용하고 있다[5].

결과적으로, Ξ에 있는 각 타입-명세는, *IN*, *IM*, *B*와 같은 집합의 원소들과, '*ref*', '*function*'과 같은 연산들의 조합으로 만들어진다.

다음은, Ξ의 각 원소를 그의 구성 요소들에 사상시키는 함수 '$\mathcal{U}$'을 Ξ의 정의에 따라 귀납적으로 정의한 것이다. 즉, '$\mathcal{U}$'는 주어진 타입-명세에 사용된 인터페이스나 구현부 클래스들의 이름의 집합을 리턴한다.단, *function* 타입에 적용되는 경우, 결과 타입-명세만을 고려한다. 아울러, $\mathcal{U}$와는 반대로 , $ty \in \Xi$가 타입-명세중 '*function*'

5) 정의가 귀납법에 의해 반복적으로 번역될 때, 몇가지 제약이 존재한다. 예를 들어 함수의 포인터등은 허용되지 않는 것이 일반적이다. 그러나 본 논문에서는 다른 정의에 지장을 주는 중요한 사항이 아니므로 고려하지 않았다.

일 경우에 한하여, 그의 인자 타입으로 사상시켜주는 함수 '$\mathcal{W}$'를 정의한다.

정의 3 (타입-명세에 들어있는 이름들) *모든 $ty \in \Xi$에 대해 함수 $\mathcal{U} : \Xi \rightarrow 2^{IN \cup IM}$ 는 다음과 같이 ty에 대해 재귀적으로 정의된다.*

(i) 만일 $ty \in B$ 이면 $\mathcal{U}(ty)$은 $\emptyset$ 이다.

(ii) 만일 $ty \in (IN \cup IM)$이면 $\mathcal{U}(ty)$은 $\{ty\}$이다.

(iii) 만일 $a \in \Xi$ s.t. $ty = ref(a)$이면 $\mathcal{U}(ty)$ is $\{ \mathcal{U}(a) \}$이다.

(iv) 만일 어떤 $a_0,..., a_n \in \Xi$ 가 있어서 $ty = function(n, a_0,.. a_n)$이라면, $\mathcal{U}(ty)$ 은 $\{ \mathcal{U}(a_0) \}$ 이다.

그리고 모든 $ty' \in \Xi$에 대해, 함수 $\mathcal{W} : \Xi \rightarrow 2^{IN \cup IM}$ 은 다음과 같이 ty'에 대해 재귀적으로 정의된다.

(i) 만일 $a_0, a_1,..., a_n \in \Xi$에 대해, $ty' = function(n, a_0,.. a_n)$ 이라면, $\mathcal{W}(ty')$ 는 $\{\mathcal{U}(a_1), ... \mathcal{U}(a_n)\}$이다.

(ii) 그 외에는, $\mathcal{W}(ty')$는 $\emptyset$이다.

이제 주어진 인터페이스나 클래스 구현부에 들어있는 슬롯과 메쏘드의 타입-명세에 대해 기술한다. 메쏘드는 오버로딩(overloading) 때문에 한 개 이상의 타입-명세를 가질 수가 있다. 함수 $\mathcal{P} : (IN \cup IM) \times \times DNSM \rightarrow 2^{\Xi}$ 는 인터페이스나 클래스 구현부에 들어있는 슬롯이나 메쏘드 이름을 클래스 정의시 명시된 타입의 집합으로 사상한다. 즉, 인터페이스나 구현부 클래스 'm'에 속하는 메쏘드 'f'의 타입들은 '$\mathcal{P}$(m,f)'로 나타낼 수 있다. 또, 인터페이스나 구현부 클래스 'm'의 메쏘드 'f'의 타입 정의들에 나타난 인터페이스나 구현부 클래스 이름의 집합은, '$\mathcal{U}(t) \cup \mathcal{W}(t) \mid t \in \mathcal{P}(m,f)$'로 나타낼 수 있다.

이제, 주어진 인터페이스의 슬롯과 메쏘드 선언 집합을 기술하는 함수 'π_I'를 정의한다. 'π_M' 역시 구현부 클래스에 대해 동일한 방식으로 정의된다.

정의 4 (인터페이스와 구현부 클래스의 슬롯과 메쏘드의 집합) *함수 π_I: $IN \rightarrow (IN \cup IM) \times \Xi$와 함수 π_M: $IM \rightarrow (IN \cup IM) \times \Xi \times$ {'PUBLIC', 'PRIVATE'} 는 다음과 같은 등식으로 정의된다. (1) 모든 $i \in IN$에 대해, $\pi_I(i) = \{ < sm, t > \mid sm \in \mathcal{U}(i)$ 이고 $t \in \mathcal{P}(i,sm) \}$, (2) 모든 $m \in IM$에 대해, $\pi_M(m) = \{ < sm, t,$ 'PUBLIC' $> \mid sm \in \delta(m)$ 이고 $t \in \mathcal{P}(m,sm) \} \cup \{ < sm, t,$ 'PRIVATE' $> \mid sm \in \delta(m)$ 이고 $t \in \setminus \mathcal{P}(m,sm) \}$.*

그리고, 이진 관계(binary relationship) 집합을 반사성(reflexive)이고, 이행성(transitive)를 보존하는 울타리(closure)로 사상하는 함수를 'Ω'라고 하자. π_I, π_M와 Ω를 써서, '$\angle_I$'와 '$\angle_M$'로 나타내어지는 인터페이스와 클래스 구현부의 계층 구조를 정의할 수 있다.

$\angle_I$의 정의는 접근 명세자(access specifier)를 제외하면, $\angle_M$의 정의와 거의 같으므로 여기서는 $\angle_M$의 정의만을 소개한다.

정의 5 (구현부 클래스의 상속) *$\angle_M \subseteq IM \times IM$는 사용자 정의 상속 관계의 부분집합으로, 다음과 같은 성질을 만족한다: 만일 $< m_1, m_2 > \in \angle_M$이면, 모든 $d_2 = < sm_2, t_2, pr_2 > \in \pi_M(m_2)$에 대해, 대응되는 $d_1 = < sm_1, t_1, pr_1 > \in \pi_M(m_1)$가 있어서,*

(i) $d_1 = d_2$ 이거나,

(ii) $sm_1 = sm_2$, $pr_1 = pr_2$이고

- *모든 $x_2 \in \mathcal{U}(t_2)$에 대해, $x_1 \in \mathcal{U}(t_1)$가 존재하여, $< x_1, x_2 > \in \Omega(\angle_I) \cup \Omega(\angle_M)$, $t_2[x_1/x_2] = t_1$이고,*
- *모든 $x'_2 \in \mathcal{W}(t_2)$에 대해, $x'_1 \in \mathcal{W}(t_1)$가 존재하여, $< x'_2, x'_1 > \in \Omega(\angle_I) \cup \Omega(\angle_M)$, 이고 $t_2[x'_1/x'_2] = t_1$이다.*

위 정의의 두번째 조건은 상속시 재정의를 허용하기 위한 반변환 조건을 기술하고 있다. 'PRIVATE' 과 'PUBLIC'과 같은 접근 제어 명세는 재정의될 수 없다. '$e[x/y]$'는, 다른 부분이 모두 'e'와 같지만 모든 'y' 의 자리에 'x'를 대입한 것을 의미한다. 이제 부터 $\Omega(\angle_M)$를 '$\angle_M$*'로 표기하고, $\Omega(\angle_I)$을 '$\angle_I$*'로 표기한다.

인터페이스와 클래스 구현부의 바인딩이 유효한지를 나타내는 것을 정의하기 위해 먼저, 세가지 관계를 인자로 하고 이러한 관계들의 복합 연산(combination)의 변형을 그 결과로 하는 함수를 정의한다.

정의 6(Extended Relationship) *주어진 세개의 이진 관계 집합 $\rho_1 \subseteq X \times X$, $\rho_2 \subseteq Y \times Y$, $\sigma \subseteq X \times Y$에 대해, $\Upsilon(\rho_1, \rho_2, \sigma)$: $X \times Y$는 다음과 같이 정의된다.*

(i) 만일 $< a, b > \in \sigma$이면, $< a, b > \in \Upsilon(\rho_1, \rho_2, \sigma)$이다.

(ii) 만일 $< b, a > \in \Omega(\rho_1)$이고, $< b, c > \in \Upsilon(\rho_1, \rho_2, \sigma)$이면, $< a, c > \in \Upsilon(\rho_1, \rho_2, \sigma)$

(iii) 만일 $< b, c > \in \Upsilon(\rho_1, \rho_2, \sigma)$, $< d, c > \in \Omega(\rho_2)$이고 어떠한 $k \in Y$ 도 네가지 조건 (1) $k \neq c$, (2)$< d, k> \in \Omega(\rho_2)$, (3)$< k, c> \in \Omega(\rho_2)$, (4) $\exists w$이고, $< w, k > \in \sigma$을 모두 만족하지 않는다면, $< b, d > \in \Upsilon(\rho_1, \rho_2, \sigma)$이다.

ρ_1와 ρ_2는 각 분리된 도메인에 존재하는 관계들을 나타낸다. σ는 이러한 두 개의 도메인을 연결해준다. $\Upsilon(\rho_1, \rho_2, \sigma)$의 각 원소들은 인자로 받은 관계들에 각각 속해있는 세 개의 경로에 의해 결정된다. $\Upsilon(\rho_1, \rho_2, \sigma)$은 바인딩 및 구현 관계를 기술하기 위해 쓰인다. 즉, '구현' 관계도 $\Upsilon(\rho_1, \rho_2, \sigma)$의 형태로 나타내어질 수 있는데, 이 때, σ는 바인딩, ρ_1와 ρ_2는 인터페이스 계층 구조와 클래스 구현부의 계층 구조와 각각 비슷한 개념이 된다.

다음은 Υ을 사용하여 사용자 정의 명시적 바인딩을 재귀적으로 정의한 것이다.

정의 7(바인딩의 집합) *$\triangleright \subseteq IN \times IM$ 은 사용자 정의 바인딩의 부분 집합이며, 다음과 같은 성질을 가진다. 만일 $< i, m > \in \triangleright$이면, 모든 $d_1 = < sm_1, t_1 > \in \pi_I(i)$에 대해, 대응되는 $d_2 = < sm_2, t_2, pr > \in \pi_M(m)$이 존재하여 $sm_1 = sm_2$, pr = 'PUBLIC' 이고,*

(i) $t_1 = t_2$ 거나

(ii) - 모든 $x_1 \in \mathcal{U}(t_1)$에 대해 $x_2 \in \mathcal{U}(t_2)$가 존재하여 $< x_1, x_2 > \in \Upsilon(\angle_I, \angle_M, \triangleright)$, 이고 $t_1[x_2/x_1] = t_2$, 이고,

- 모든 $x'_1 \in \mathcal{W}(t_1)$, $x'_2 \in \mathcal{W}(t_2)$가 있어서, $x'_1 = x'_2$이다.

바인딩과 여기서 유추된 관계들은 $\Upsilon(\angle_I, \angle_M, \triangleright)$로 정의될 수 있다. 이제부터는 간단히 $\Upsilon(\angle_I, \angle_M, \triangleright)$를 '$\triangleright*$'로 나타낸다.

그런데, 앞서 설명되었듯이 모든 하위 클래스 구현부가 상위 클래스 구현부의 인터페이스를 '구현'하는 것은 아니다. 따라서, 다음과 같이 $\angle_M$을 제한함으로써, 새로운 관계 '$\angle_{M+}$'를 정의한다.

정의 8 (제한된, 클래스 구현부의 상속) *모든 $< m_1, m_2 > \in \angle_M$에 대해, 만일 $m_3 \neq m_2$ 이고, $< m_1, m_3 > \in \angle_M$ 이고, 어떤 w가 있어서 $< w, m_3 > \in \triangleright$인 m_3가 존재하지 않는다면, $< m_1, m_2 >$가 $\angle_{M+}$에 속한다.*

즉, $\mathcal{U}(\angle_{M+})$는 클래스 구현부의 상속 관계중 인터페이스도 함께 상속받는 것들만을 모은 것을 의미한다. 이제, $\mathcal{U}(\angle_{M+})$를 '$\angle_{M+}$*'로 간단히 칭하기로 한다. 그런데, $\triangleright$*는 그 정의에 $\angle_M$대신 $\angle_{M+}$을 사용해서 새로 정의될 수 있다. 이렇게 새로 정의된 관계는 본 모델이 결과적으로 정의하는 사용자 정의 바인딩을 의미하게 되는데, 이를 '$\triangleright_+$'로 부른다. 결과적으로는 $\Upsilon(\angle_I, \angle_{M+}, \triangleright_+)$를 나타내는 관계 '$\triangleright_+*$'를 생각해 볼 수 있으며, 이것이 바로 인터페이스와 클래스 구현부의 '구현' 관계를 의미하게 된다.

5.2 타입 규칙

표 1은 본 절에서 정의되는 타입 시스템을 위한 간략한 언어 구문이다. **Typ**는 기본 타입, 클래스 이름, 그리고 이들의 **ref** 타입을 의미한다. **RawTyp**은 **Typ**과 비슷하지만, **Typ**의 클래스 이름 대신 그 클래스의 슬롯과 메쏘드의 리스트를 구체적으로 가진다. 여기서 'T'를 클래스 정의를 위한 환경(environment)으로 정의한다. 즉, 함수 'T'는 IN과 IM의 원소를 인자로 받아, **RawTyp**을 결과로 리턴한다. 환경은 표 2에 기술되어 있다.

Num ∋ n	::=	0 | 1 |
Bool ∋ b	::=	true | false
Pr ∋ pr	::=	PRIVATE | PUBLIC
Exp ∋ e	::=	v | n | b |
		if (e) e else e |
		v : τ := e |
		e(e) | a.x | s.x := e |
		new e | ref(e) | !e | λx.e | e;e |
		persistent class v : v, . . . ,v [$l_1 : \tau_1, \ldots, l_n : \tau_n$] |
		class v : v, . . , v [implements v]
		public [$l_{pub1} : \tau_{pub1}, \ldots, l_{pubn} : \tau_{pubn}$]
		private [$l_{pr1} : \tau_{pr1}, \ldots, l_{prn} : \tau_{prn}$] |
Typ ∋ τ	::=	**Bool** | **Num** |
		IN | *IM* |
		ref(τ) |
		$\tau_1, \tau_2 \ldots \tau_n \rightarrow \tau_0$
RawTyp ∋ x	::=	**Bool** | **Num** |
		< [$l_{pr1} : \tau_{pr1}, \ldots, l_{prn} : \tau_{prn}$], [$l_{pub1} : \tau_{pub1}, \ldots, l_{pubn} : \tau_{pubn}$] > | ref(τ) |
		$\tau_1, \tau_2 \ldots \tau_n \rightarrow \tau_0$

표 1 구문

$$[\text{Env I}]\ \frac{\text{persistent class v} : v_1,\ldots,v_x\ [\ l_1 : \tau_1,\ldots,\ l_n : \tau_n]}{\vdash < \text{v, persistent class v} : v_1, \ldots, v_x\ [l_1 : \tau_1, \ldots, l_n : \tau_n] > \in \Gamma}$$

$$[\text{Env II}]\ \frac{\begin{array}{l}\text{class v} : v_1,\ldots,\ v_x \text{ implements v} \\ \text{public [} l_{pub1} : \tau_{pub1}, \ldots, l_{pubn} : \tau_{pubn}\text{] private [} l_{pr1}: \tau_{pr1}, \ldots, l_{prn}: \tau_{prn}\text{]}\end{array}}{\begin{array}{l}\vdash < \text{v, class } v_1, \ldots, v_x \text{ implements v} \\ \text{public [} l_{pub1} : \tau_{pub1}, \ldots, l_{pubn} : \tau_{pubn}\text{] private [} l_{pr1}: \tau_{pr1}, \ldots, l_{prn}: \tau_{prn}\text{] } > \in \Gamma\end{array}}$$

표 2 환경(Environment)

그리고, 주어진 클래스 이름과 슬롯 이름에 대해, 함수 'ν' 는 대응되는 **Typ** 타입을 리턴한다.

정의 9 (함수 ν) *함수 ν : IN × DNSM → **Typ** 는 다음과 같이 정의된다. 만일 모든 υ ∈ IN, s ∈ DNSM에 대해 Γ(υ) = ... [... l_i : τ_i...] ... and, l_i = s 인 i가 존재한다면, ν(i, s) = τ_i이다.그렇지 않으면, ν(i, s) =∅이다.*

이제, 앞 절에서 정의한 여러 관계와 성질들을 사용하기 위해 환경 'Σ' = < *I*, *M*, $\angle_I$*, $\angle_{M+}$*, $\rhd_{+}$* >를 정의한다. 표 3과 표 4는 Σ, Γ and ν를 기반으로 하는 타입 규칙을 나열한 것이다. 이러한 타입 규칙들은 기존의 타입 형식화[15,20]에 [I-Sub], [M-Sub], [I-M-Implementing]와 같은 새로운 규칙을 첨가하고 확장한 것들이다.

$$[\text{Refl}]\ \frac{}{\vdash \tau \le \tau}$$

$$[\text{Func}]\ \frac{\Sigma \vdash \tau_0 \le \tau_0',\ \tau_1' \le \tau_1,\ \ldots\ \tau_i' \le \tau_i,\ \tau_n' \le \tau_n}{\Sigma \vdash \tau_1,\ldots\ \tau_n \text{ -> } \tau_0 \ \le\ \tau_1',\ldots\ \tau_n' \text{ -> } \tau_0'}$$

$$[\text{Trans}]\ \frac{\Sigma \vdash \tau \le \tau' \ \text{ and } \ \tau' \le \tau''}{\Sigma \vdash \tau \le \tau''}$$

$$[\text{I-Sub}]\ \frac{\Sigma \vdash i_1\ \angle_I{}^*\ i_2}{\Sigma \vdash i_1 \le i_2}$$

$$[\text{M-Sub}]\ \frac{\Sigma \vdash m_1\ \angle_{M+}{}^*\ m_2}{\Sigma \vdash m_1 \le m_2}$$

$$[\text{I-M-implementing}]\ \frac{\Sigma \vdash i\ \rhd_{+}{}^*\ m}{\Sigma \vdash i \le m}$$

표 3 서브 타입 규칙

$$[\text{Var}]\ \frac{v \in \text{dom}(\Gamma)}{\Sigma;\ \Gamma \vdash v : \Gamma(v)}$$

$$[\text{Conditional}]\ \frac{\Sigma;\ \Gamma \vdash e_1 : \text{Bool},\ e_2 : \tau_1,\ e_3 : \tau_2}{\Sigma;\Gamma \vdash \text{if}\ \ (e_1)\ \ e_2\ \ \text{else}\ \ e_3\ \ : \tau_2}$$

$$[\text{New}]\ \frac{\Sigma;\Gamma \vdash e : \tau}{\Sigma;\Gamma \vdash \text{new}\ \ e : \text{ref}(\tau)}$$

$$[\text{Slot}]\ \frac{\Sigma;\Gamma \vdash e\ .\ \tau_1,\ \tau_1 \in (I \cup M),\ \nu(\tau_1,\ l) = \tau_2}{\Sigma;\Gamma \vdash e.l : \tau_2}$$

$$[\text{Message}]\ \frac{\Sigma;\Gamma \vdash e_1 : \tau_1,\ \ e_2 : \tau_2,\ \ \tau_1 = \tau' \text{ -> } \tau'',\ \ \tau_2 \le \tau'}{\Sigma;\Gamma \vdash e_1(e_2)\ \ \ \tau''}$$

$$[\text{Ref}]\ \frac{\Sigma;\Gamma \vdash e : \tau}{\Sigma;\Gamma \vdash \text{ref}(e) : \text{ref}(\tau)}$$

$$[\text{Deref}]\ \frac{\Sigma;\Gamma \vdash e : \text{ref}(\tau)}{\Sigma;\Gamma \vdash\ \ !\,e : \tau}$$

표 4 표현식(expression) 타입 규칙

6. 관련 연구와의 비교

인터페이스와 클래스 구현부가 분리된 모델에서는 기본적으로 (1) 인터페이스--인터페이스 관계(인터페이스 계층 구조), (2) 클래스 구현부--클래스 구현부 관계(클래스 구현부 계층 구조), (3) 인터페이스--클래스 구현부 관계(구현 관계) 등의 세가지 관계가 존재하게 된다. 이러한각 관계들을 지원하는지 여부와, 지원하는 경우 각 관계를 결정 짓는 방법에 따라 각 분리 모델의 성격이 결정된다.

현재까지 등장한 인터페이스와 클래스 구현부의 분리에 관한 연구들은 크게 타입 이론에서 출발한 것과 그 밖의 접근 방법으로 나누어 볼 수가 있다. 먼저 타입 이론에서 출발한 연구들[4,9,13]은 [8,18]에서 밝혀진 것과 같이, 서브 타입 클래스 구현부의 상속에 의해 식별되는 것이 항상 옳지만은 않다는 사실에 기인한다. 따라서, 이들은 타입을 위해 인터페이스를 분리해내어, 기존의 타입 이론[16,20,21]들을 인터페이스 집합에 한정하여 적용시키는 것만을 목적으로 하였다.

특히, 이러한 모델들은, 인터페이스 간의 상속을, 구조-일치 기법(structural equivalence)[22]에 따라 정의한다[4,9,13]. 즉, 타입을 사용자가 주는 정보나 이름이 아닌, 컴파일러에 의해 내용을 비교함으로써 식별하는 방식이다. 예를 들어, 앞서 2절의 예에 이러한 구조 일치 기법이 적용된다면, 인터페이스인 'Collection'과 'Set', 'Bag'의 관계가 사용자의 명시에 의해 정의되는 것이 아니라, 컴파일러가 'insert', 'delete'등의 메쏘드를 공통적으로 가지고 있음을 비교하여, 정해지게 된다. 순응성(conformance)[13]이라 불리는 인터페이스-클래스 구현부 관계도, 본 논문의 '구현' 관계와는 달리 사용자가 지정해 줄 수 있는 것이 아니라, 컴파일러가 전적으로 결

정한다. 그러나, 이러한 구조 일치 기법은, 비슷한 메쏘드나 슬롯을 가지고 있기만 하면, 인터페이스나 클래스 구현부들 간에 의도하지 않은 관계가 사용자가 모르는 사이에 정의될 수가 있다는 단점이 있다. 즉, 예를 들면 현금 카드를 판독하는 기계를 나타내는 'Card-Machine'이라는 인터페이스가 있어서, 'insert', 'delete'를 메쏘드로 가지고 있는 경우, 인터페이스 'Set'이나 'Collection'과 실제로는 무관하더라도, 이들과 계층 구조를 이루는 것으로 인식되게 된다. 또, 일일이 슬롯과 메쏘드 시그니쳐를 비교해야하기 때문에, 성능도 다른 기법들에 비해 좋지않다고 알려져 있다[22]. 그리고, 이러한 모델들 중에서는 클래스 구현부 간의 계층 구조도 지원 되지 않는 등 클래스 구현부에 대해서는 별 고려를 하지 않는 것도 있다[13].

또, 타입의 의미를 인터페이스에만 부여하고, 클래스 구현부는 타입 시스템에서 제외시키고 있기 때문에[5], 클래스 구현부들이나 인터페이스-구현부가 분리되지 않은 일반 클래스들이 타입 도메인이 될 수 없다는 단점이 있다. 이러한 측면은 이들 분리 모델이 일반적인 객체지향 프로그래밍 언어들에 적용되어 확장되는 경우, 이미 타입 시스템의 일부로 참여하고 있는 기존의 일반 클래스들을 변형없이 재사용하기 어렵게 한다.

인터페이스와 클래스 구현부의 분리 모델의 또 다른 부류는 타입 이론과는 무관하게 개발된 것들이다. 분산 환경을 목적으로 하여 개발된 Abel Group의 C++ 확장 모델[12]은 극히 초보적이지만 사용자가 의도한 인터페이스 및 클래스 구현부 계층 구조와 인터페이스-클래스 구현부 구현 관계를 제공하는 C++ 확장 모델을 소개하고 있다. 그러나, 이 방식은 인터페이스 분리를 위해 개발된 특정 프로그래밍 언어 모델이라기 보다는, 두 계층 구조의 분리와 결합을 직관적으로 C++의 다중 상속에 그대로 사상시키는 방식을 기술한 것이다. 따라서, 서로 다른 종류의 계층 구조들이 하나의 C++ 계층 구조에 얽혀 있기 때문에 부수 작용(side-effect)등에 의해 원하지 않는 결과를 초래할 위험이 있는데, 모델의 의미나 타입 공간에 대한 명확한 기술이 없다. 이 결과로, 이 모델이 취한 접근 방식은, 부수 결과(side-effect)에 의해 사용자가 의도하지 않은 인터페이스와 클래스 구현부의 구현 관계가 삽입될 수 있다고 밝혀진 바 있다[14].

분산 환경 지원을 위한 CORBA[6] 모델과 객체지향 데이타베이스를 위한 ODMG 93[7] 모델도 인터페이스와 클래스 구현부 분리에 관해 언급하고 있다. CORBA[6]은 클래스 구현이나 구체적인 프로그래밍 언어에 의존하지 않는 IDL(interface definition language)을 사용함으로써 여러 프로그램이 공유할 수 있도록 한다. ODMG93[7]에서는 데이타베이스 관리자가 특정 스키마 클래스의 인터페이스에 대해 원하는 클래스 구현부를 지정하는 것을 허용하고 있다. 그러나 이들은 특정 순간에는 한 클래스당 하나의 클래스 구현부를 지정하도록 하고 있거나, 타입 재정의를 허용하지 않고 있기 때문에 본 모델이나 타입 이론을 기반으로하는 다른 접근 방식들에 비해 상당히 제한적이다.

기존의 객체지향 프로그래밍 언어의 형식화된 타입 이론들[20,21,24]은 인터페이스 분리에 촛점을 맞추기에 필요 이상으로 비대하기 때문에, 본 모델이 결과적으로 의도하고 있는 다중 클래스 공간(인터페이스와 클래스 구현부)을 기술하기에는 적절하지 못하다고 판단하였다. 그러나, 본 논문에서 제안한 타입 규칙들중 기본적인 것들의 대부분은, 일반적으로 통용되는 정의들을 그대로 사용하였다.

7. 결 론

본 논문에서는, 클래스의 인터페이스와 클래스 구현부의 분리에 관해 고찰하고, 새로운 모델을 제안하였다. 이를 위하여 기존의 프로그래밍 언어에서 볼수 있는 일반적인 클래스 개념 외에 인터페이스, 클래스 구현부를 도입하였다.

그리고, 인터페이스와 구현부 각각의 계층 구조를 허용하며, 이들을 전체 클래스의 계층 구조에서 보존되도록 하였으며, 인터페이스와 구현부 간의 '구현' 관계를 정의하여, 인터페이스와 클래스 구현부 및 이들을 결합시킨 클래스 모두가 타입으로 사용되는 경우에도 적용할 수 있는 타입 시스템을 정의하였다. 따라서, 새로운 언어를 만들 때 뿐 아니라 기존의 객체지향 언어를 확장하는 데에도 적용될 수 있으며, 결과적으로는 의도하지 않은 부수작용이 없이 정확히 사용자가 원하는 결과 타입 시스템이 생성되게 된다.

본 논문의 모델은 일반 프로그래밍뿐 아니라, 인터페이스부분의 공유가 필수적인, 데이타베이스 프로그래밍, 분산 프로그래밍에도 유용하게 적용될 수 있다. 특히, 현재는 ODMG93[7] 기반의 객체지향 데이타베이스 관리 시스템인 SOP(SNU OODBMS Platform)[6]의 C++기반 데이타베이스 프로그래밍 언어에 본 모델을 적용하여 구현하고 있다[23,25].

앞으로는 본 모델을 여러 형태의 메쏘드 오버로딩

6) 서울대학교에서 1993년부터 시작하여 1995년에 개발된, ODMG-93 모델을 기반으로하는 객체지향 DBMS.

(overloading)과 추출(dispatch) 방식을 포괄할 수 있도록 확장할 계획이다.

참 고 문 헌

[1] Bjarne Stroustrup, editor. *The C++ programming language second edition.* Addison-Wesley Publishing Company, Inc., April 1991.

[2] Jean-Marc Jezequel. *Object-Oriented Software Engineering with Eiffiel.* Addison-Wesley Publishing Company, Inc., 1996.

[3] A. Goldberg and D. Robson. *Smalltalk-80 : The Language and Its Implementation.* Addison-Wesley Publishing Company, Inc., 1983.

[4] G. Baumgartner and V. F. Russo. "Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism," Technical Report CSD-TR-93-059, Purdue University, September 1993.

[5] Luca Cardelli, James Donahue, and Lucille Glassman. "Modula-3 Language Definition," *ACM SIGPLAN Notice*, Vol.8, No.29, pp. 15--42, August 1992.

[6] DEC, HP, HyperDesk, NCR, Object Design, and SunSoft. *The Common Object Request Broker : Architecture and Specification.* OMG group, 1991.

[7] R. G. G. Cattell. *Object Database Standard : ODMG-93.* OMG group, 1993.

[8] William R. Cook. "Interfaces and Specifications for the Smalltalk-80 Collection Classes," In *Proc. of the ACM OOPSLA Conf.*, pp. 1--15, 1992.

[9] Elana D. Granston and V. F. Russo. "Signature-Based Polymorphism for C++," In *Proceeding of Usenix C++ conference*, pp. 65--79, 1991.

[10] A. Nico Habermann and Dewayne E. Perry. *Ada for Experienced Programmers.* Addison-Wesley Publishing Company, Inc., 1983.

[11] Dinesh Katiyar, David Luckham, John Michell, and Sigurd. "Polmorphism and Subtyping in interface : SIGPLAN Notice No.8 Vol.29," In *ACM Workshop on Interface Definition Languages*, pp 22--34, January 1994.

[12] Bruce Martin. "The Separation of Interface and Implementation in C++," In *Proceeding of Usenix C++ conference*, pp. 51--63, 1991.

[13] R. K. Raj and et al. "The Emerald Approach to programming," Technical Report 88-11-01, University of Washington, November 1989.

[14] E.S. Cho. "A Semantics of the Separation of Interface and Implementation in C++," In *Proc. of Int'l conference on COMPSAC*, 1996.

[15] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. "Application of OOP Type Theory: State, Decidability, Integration," In *Proc. of the ACM OOPSLA Conf.*, 1994.

[16] Luca Cardelli. "A Semantics of Multiple Inheritance," In *Computer Science Lecture Note:173.* Springer-Verlag, 1984.

[17] Giuseppe Castagna. "Covariance and Contravariance : Conflict without a Cause," *ACM Transactions on Programming Languages and Systems*, Vol.17, No.3, pp. 431--447, 1995.

[18] William R. Cook. "Inheritance Is Not Subtyping," In *Proc. of SIGPLAN Conf. on Principle of Programming Languages*, pp. 125--135, 1990.

[19] David Gelertner and Suresh Jagannathan. *Programming Linguistics.* MIT Press, 1990.

[20] Roberto M. Amadio and Luca Cardelli. "Subtyping Recursive Type," *ACM Transactions on Programming Languages and Systems*, Vol.15, No.4, pp. 575--631, September 1993.

[21] Atsushi Ohori. "A Polymorphic Record Calculus and Its Compilation," *ACM Transactions on Programming Languages and Systems*, Vol.17, No.6, 1995.

[22] R.C.H Connor, A.L. Brown, Q.I Cutts, and A. Dearle. "Type Equivalence Checking in Persistent Object Systems," In *The Fourth International Workshop on Persistent Object Systems*, pp. 154--167, 1990.

[23] E.S. Cho. "A Noble Integration Mechanism of OOPLs and OODBMSs : LOD*," In *IEEE Database Engineering*, 1997. *submitted.*

[24] Gail Anne Mitchell. "*Extensible Query Processing in an Object-Oriented Database,*" PhD thesis, Brown University, May 1993.

[25] 안정호 외, "Soprano: 객체 저장 시스템의 설계및 구현", 정보과학회 논문지(C), 제 2권 제 3호, 1996

조 은 선

1991년 2월 서울대학교 계산통계학과 졸업. 1993년 2월 서울대학교 계산통계학과 졸업(이학 석사). 1993년 3월 ~ 현재 서울대학교 전산과학과 박사과정. 관심 분야 객체지향 프로그래밍 언어, 객체지향 데이타베이스

한 상 영

1972년 서울대학교 공과대학 응용수학과 졸업(공학사). 1977년 서울대학교 대학원 계산통계학과(이학석사). 1977년 3월 ~ 1978년 3월 울산대학교 공과대학 전임강사. 1984년 3월 ~ 현재 서울대학교 자연과학대학 전산과학과 교수. 관심분야는 병렬처리임.

김 형 주

제 24 권 제 1 호(B) 참조