C++ 프로그래밍을 위한 구체적 객체 모델의 작성 기법

김 태 균† · 임 채 덕†† · 송 영 기†† · 인 소 란††

요 약

객체 지향 패러다임의 확산으로 인하여 소프트웨어 개발을 위한 객체 모델의 사용이 일반화되고 있다. 소프트웨어 요구사항의 분석 및 설계 결과로 생성되는 객체 모델은 소프트웨어의 구현 시에 많은 도움이 된다. 특히 작성된 객체 모델이 구체적인 경우에는 자동적인 원시 코드의 생성도 가능하다. 따라서 시스템 분석가나 설계자는 분석 단계 초반기의 추상적인 객체 모델을 정제함으로써 구체적인 객체 모델을 유도하기 위해 많은 시간을 투입해야 한다. 그러나 추상적 객체 모델을 구체적 객체 모델로 정제하는 과정은 주로 설계자의 비정형적인 경험에 의하여 이루어지는 것이 현실이다.

본 논문에서는 OMT(Object Modeling Technique)의 객체 모델을 대상으로 추상적 모델의 구체화 기법을 논한다. 따라서 본문에서 제시되는 주된 내용은 객체 모델의 추상화 정도에 대한 정의와 모델의 변환 규칙에 대한 것이다. 이 변환 규칙은 정보 통신 서비스 개발 프로세스 모형화 개발 지원 도구의 일부분인 프로세스 모델러(Process Modeler)의 설계시에 적용되고 있으며 그 결과로 C++ 코드로 변환하기 쉬운 구체적 객체 모델을 얻을 수 있었다.

A Technique of Deriving Concrete Object Model for C++ Programming

Tae Gyun Kim† · Chae Deok Lim†† · Young Ki Song†† · Sho Ran In††

ABSTRACT

The usage of object models for the development of software has been growing due to the prevalence of the object oriented paradigm. The object models produced as results of requirements analysis and design activities are very beneficial to the implementation phase. It is even possible for source code to be generated automatically if object models are concrete enough. Therefore system analyzers and designers should make an effort to refine the abstract object model defined at an early stage in order to achieve a more concrete object model. In general, refining an abstract object model into a concrete model depends too much on the designer's informal experience.

In this paper, we present the refinement techniques required for concreting an abstract object model based on OMT(Object Modeling Technique)'s notation. We will discuss the definition of the abstraction level of an object

※본 연구는 전자통신연구소의 위탁과제 "정보 통신 서비스 개발 프로세스 모형화 개발 지원 도구 연구"의 지원을 받았음.

† 정 회 원:부산 외국어대학교 컴퓨터 공학과
†† 정 회 원:한국전자통신연구원 소프트웨어 공학 연구실
논문접수:1996년 7월 31일, 심사완료:1997년 1월 21일

model and the transformational rules of refinement. These transformational rules are currently applied to the design of a software tool, named Process Modeler, which is a major component of the software development process modeling system for ICS(Information Communication Service). Finally we can achieve a concrete object model which can easily be translated into C++ source code.

1. 서 론

최근 10여 년 간 중점적으로 이루어지고 있는 객체 지향 패러다임에 대한 연구는 소프트웨어 개발 방식에 있어서 많은 변화를 야기시키고 있다. 즉 객체 지향 패러다임의 적용에 따른 재사용[13, 16]의 일반화, 생산성의 향상이 실증적으로 증명되고 있다. 소프트웨어 공학적 관점에서 객체 지향과 관련되는 그 동안의 연구 동향은 첫째, Smalltalk, Ada, Objective-C, Eiffel, C++와 같은 객체 지향 언어에 대한 연구[1, 9, 5, 4, 17], 둘째, Booch의 OOD(Object Oriented Design)[7], Wasserman의 OOSD(Object Oriented Structured Design)[2], Coad와 Yourdon의 OOA(Object Oriented Analysis)[14], Wirfs-Brock의 RDD(Responsibility Driven Design)[15], Rumbaugh의 OMT[8]와 같은 방법론에 관한 연구, 그리고 객체 지향 기법을 지원하는 CASE(Computer Aided Software Engineering) 도구[11]에 관한 연구이다.

소프트웨어 개발 공정의 관점에서 객체 지향 패러다임이 제공하는 가장 큰 특징은 개발 프로세스가 반복적이고(iterative), 솔기 없는(seamless) 특징을 가지고 있다는 점이다. 객체 지향 소프트웨어 개발 공정의 반복적인 특성은 요구 분석, 설계, 구현, 테스팅의 각 단계가 프로젝트의 진화에 따라서 기능 수정과 기능 향상을 위해 쉽게 반복될 수 있음을 의미하며, 솔기 없는 특성은 전체 공정에 대한 일관된 개념의 적용을 통해서 각 단계에서 생성된 문서간의 변환이 쉬워졌음을 의미한다. 이러한 특성은 각 단계의 임무와 표기법이 이질적인 특성을 갖는 구조적 기법과 비교해 볼 때 프로젝트의 보수 유지가 매우 용이해 졌음을 의미한다. 객체 지향 방법론의 반복적인 특성은 나선형 생명 주기 모델(spiral life-cycle model)[3]을 가능하게 하며, 솔기 없는 특성은 역공학[6], 재구조화 등의 적용을 쉽게 한다.

객체 지향 개발 공정의 반복적이고 솔기 없는 특성의 분석 문서로부터 구현 프로그램까지의 일관적인 공정을 효과적으로 제어함에도 불구하고, 객체 지향 패러다임을 적용하고자하는 소프트웨어 개발자 입장에서 당면한 최대의 문제는 프로젝트 초기에 생성되는 추상적인 분석 모델을 어떻게 하면 구체적인 모델로 변형하는가에 대한 것이다. 본 논문에서는 객체 지향 언어 C++와 객체 지향 표기법인 OMT의 객체 모델을 대상으로 추상적 객체 모델을 구체적 객체 모델로 변환하는 기법을 제시한다. 이러한 변환 기법은 객체 지향 소프트웨어의 개발 시에 보편적인 지침서의 역할을 할 수 있으며, 특히 본 연구에서는 이 기법을 정보 통신 서비스 개발 프로세스 모형화 개발 지원 도구의 일부분인 프로세스 모델러의 개발 시에 적용하고 있다.

본 논문에서 제시하고자 하는 변환 기법과 관련된 연구로는 EER(Extended Entity-Relationship) 모델로부터 관계형 데이타베이스의 논리적 스키마를 정의하고자 하는 Teorey의 연구[18]와 OMT의 객체 모델을 관계형 데이타베이스의 스키마로 변환하는 연구[12]와 같이 데이타베이스로의 적용에 대한 연구가 주류를 이루고 있으며, 객체 모델을 객체 지향 프로그램으로 변환하고자 하는 연구는 아직 미흡하다. 본 논문의 변환 기법은 약 65 KDSI(Kilo Delivered Source Instructions)의 분량을 차지하는 객체 지향 CASE 도구인 OODesigner[19, 20]의 개발 과정과 재구조화 과정을 통하여 정립된 것으로 현재 실증적으로 적용되고 있다.

본 논문의 2 장에서는 객체 모델의 구성 요소에 대한 정의와 함께 추상적 객체 모델, 구체적 객체 모델을 정의한다. 3 장에서는 추상적 객체 모델을 구체적 객체 모델로 변환하는 기법을 기술한다. 4 장에서는 변환 기법의 적용 예를 보이며, 5 장에서는 결론과 함께 차후의 연구 방향에 대하여 논한다.

2. 정 의

본 장에서는 3 장에서 논의할 변환 기법의 대상이

되는 추상적 객체 모델과 구체적 객체 모델을 정의하며 아울러 변환 기법의 적용 시에 필요한 기본 클래스들에 대하여 논한다.

2.1 객체 모델의 정의

[정의 1] 객체 모델 OM은 튜플 〈CL, GE, AG, CO, AS〉로 정의된다. 여기에서 CL은 클래스의 집합, GE는 일반화(generalization)의 집합, AG는 집합화(aggregation)의 집합, CO는 협동 관계(collaboration)의 집합, AS는 결합 관계(association)의 집합을 말한다.

[정의 1]의 객체 모델에 포함되는 요소들은 기존에 사용되고 있는 객체 지향 기법들에 대한 분석 결과로써 추출된 핵심적인 요소들이다. 이 요소들 중에서 GE는 IS_A 관계를 의미하며, AG는 PART_OF 관계, CO는 USE_OF 관계, 그리고 AS는 객체간에 존재하는 일반적 관계를 의미한다. 이들 요소는 대부분 객체 지향과 관련된 분야에서 보편적으로 사용되는 요소이다. 그런데 본 논문에서 인용하고자 하는 OMT의 객체 모델에는 CO가 포함되어 있지 않다. CO은 CRC(Class Responsibility Collaboration) 모델[10]이나 RDD 모델에서 핵심적인 위치를 차지하는 요소로서 시스템의 모델링 시에 한 클래스가 다른 클래스의 서비스를 필요로 하는 경우를 표현하기 위해 사용된다. OMT의 객체 모델에 협동 관계를 추가하고자하는 시도는 [19]에서 이루어진 바가 있으며 CO에 대한 표기법을 위하여 화살표를 사용한다. 따라서 본 논문에서 사용할 객체 모델은 원래의 OMT 모델에 CO가 추가된 것이다.

[정의 2] 객체 지향 프로그래밍 언어 L_{obj}가 객체 모델 OM의 개별적인 요소 OM_i (여기에서 $OM_i \in$ {CL, GE, AG, CO, AS} 임)를 지원하는 문법과 어의를 제공할 때 OM_i는 L_{obj}에 의하여 직접적으로 표현가능(directly representable)하다고 하고 다음과 같이 표현한다.

$$OM_i \rightarrow L_{obj}$$

이 정의는 객체 모델의 개별 구성 요소들 중에서 어떠한 것이 프로그램으로 직접적으로 변환될 수 있는 가를 기술하기 위한 정의이다. 객체 모델을 구성하는 다섯 가지 구성 요소들 모두가 객체 지향 프로그램으로 직접 변환 가능한 것이 아니기 때문에 추상적인 객체 모델을 구체화 할 필요성이 생긴다.

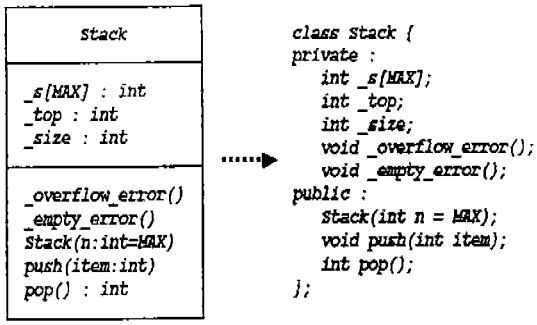
[가정 1] 객체 지향 프로그래밍 언어 C++를 L_{c++}라 할 때, 객체 모델의 구성 요소 중에서 CL, GE, AG, CO은 L_{c++}에 의하여 직접적으로 표현 가능하다. 즉, $CL \rightarrow L_{c++}$, $GE \rightarrow L_{c++}$, $AG \rightarrow L_{c++}$, $CO \rightarrow L_{c++}$이라고 가정한다.

(가정에 대한 이유) 객체 모델의 구성 요소 CL은 클래스의 이름과 데이타 멤버, 멤버 함수를 표현하기 위한 추상화 구조이다. C++의 클래스 선언문도 마찬가지로 클래스의 이름과 데이타 멤버, 멤버 함수를 기술하기 위한 문법 구조를 제공하고 있기 때문에 CL이 C++에 의해 직접적으로 표현 가능함은 쉽게 알 수 있다. 클래스간의 IS_A 관계를 나타내는 GE는 C++의 상속에 의하여 기술될 수 있으므로 직접적으로 표현 가능하다. 클래스간의 부품 관계를 나타내는 AG는 자동 변수(automatic variable)로 선언되는 데이타 멤버로 구현될 수 있거나 주 클래스의 생성자에서 부속 클래스의 기억 장소를 동적으로 할당하는 방식으로 구현될 수 있다. 마지막으로 클래스간의 서비스 사용 관계를 나타내는 CO는 클래스간의 함수 호출로서 구현될 수 있다. 이 경우에 서비스를 제공하는 협동자(collaborator)는 서비스를 필요로 하는 클래스에 포인터형을 갖는 데이타 멤버로 표현되거나 지역 변수로 사용된다.(끝)

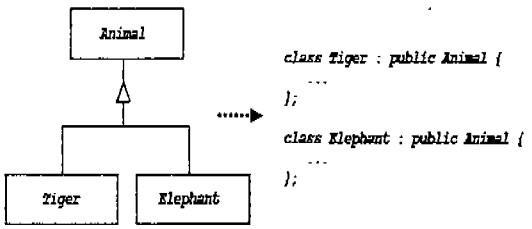
[가정 1]에 대한 구체적인 예는 다음과 같다. 우선 $CL \rightarrow L_{c++}$의 예로서 (그림 1)은 객체 모델로 기술된 클래스 Stack이 C++언어로 변환된 예를 보여 준다. 이 그림에서 알 수 있는 바와 같이 객체 도표에 명시된 클래스 정의가 C++언어와 논리적으로 일대일 대응을 이룰 수 있음을 알 수 있다.

다음으로 (그림 2)는 $GE \rightarrow L_{c++}$의 예이다. 이 그림에서 볼 수 있는 바와 같이 객체 모델에 표현된 일반화 표기법은 C++의 상속을 위한 문법에 의해 직접적으로 표현 가능하다.

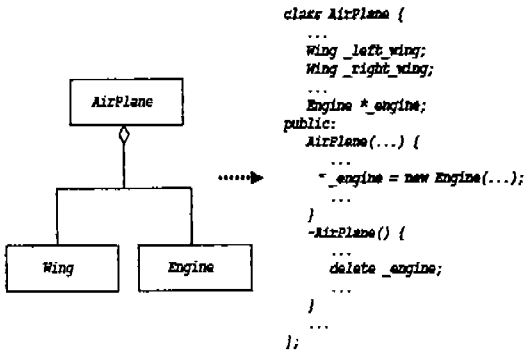
(그림 3)은 집합화의 변환 규칙을 보여 주는 예이다. 객체 모델의 집합화는 C++ 프로그래밍 시에 두



(그림 1) 클래스 정의의 변환
(Fig. 1) A transformation for class definition



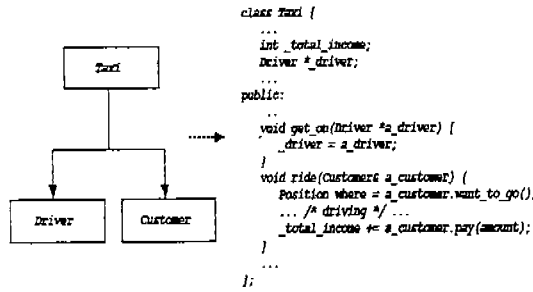
(그림 2) 일반화의 변환
(Fig. 2) A transformation for generalization



(그림 3) 집합화의 변환
(Fig. 3) A transformation for aggregation

가지 방식으로 표현 가능하다. 그 한 가지는 부속 객체를 주 객체의 데이타 멤버로 정의하되 자동 변수로 포함시키는 경우이며, 다른 하나는 부속 객체를 포인터 형의 변수로 정의하고 주 객체의 생성자에서 부속 객체를 위한 동적 할당을 실행하는 것이다. 예를 들어 (그림 3)에서 부속 객체 _left_wing과 _right_wing은 첫 번째 방식으로 정의되었으며, 부속 객체 _engine은 두 번째 방식으로 정의되었다. 이 두 가지 경우 모두 부속 객체의 수명(life time)이 주 객체와 일치한다는 공통점을 갖는데 이 점이 집합화의 본질적 특성이다. (그림 3)의 모델이 의미하는 내용은 "비행기는 날개와 엔진을 포함하고 있으며 이들 부속품의 수명은 비행기 전체의 수명과 일치한다"는 것이다.

마지막으로 (그림 4)는 협동 관계의 변환 규칙을 보여 주는 예이다. 협동 관계는 구조적 언어에서의 함수 호출 관계와 비슷한 경우로 어떠한 객체가 다른 객체의 서비스를 이용하는 관계를 갖는다. 그런데 이 관계가 집합화와 구별되어야 하는 점은 두 객체간의 수명이 일치하지 않는다는 점이다. 즉 (그림 4)에서와 같이 "택시"와 "운전사"의 관계가 협동 관계의 예인데, 이 경우 "택시"의 수명과 "운전사"의 수명은 상호 독립적이기 때문에 집합화와 달리 취급되어야 한다. 협동 관계가 C++ 프로그램으로 구현되는 방식은 두 가지인데, 그 하나는 사용자 객체의 데이타 멤버에 협동자 객체를 가리키는 포인터를 유지하도록 하는 것이며, 다른 하나는 사용자 객체의 멤버 함수 내에서 협동자 객체를 지역 변수로 정의하는 방식이다. 첫째 방식은 협동자 객체가 사용자 객체에 의하여 자주 접근되는 경우에 필요하며 둘째 방식은 자주 접근되지 않는 경우에 사용된다. (그림 4)에서 협동자 객체인 "운전사"는 "택시"에 의하여 자주 사용되므로 그 포인터가 데이타 멤버로 정의되어 있으며, "손님" 객체는 "택시"와 잠깐 동안 관계를 맺기 때문에 멤버 함수 ride()내에서만 사용되고 있다. 협동 관계를 이해하는 또 다른 관점은 이 관계가 일대일 대응



(그림 4) 협동 관계의 변환
(Fig. 4) A transformation for collaboration

을 이루는 결합 관계의 특수한 경우라는 것이다. 즉 결합 관계 중에서 IS_A 관계와 PART_OF 관계가 각각 일반화와 집합화 관계로 특화되었듯이 USE_OF 관계는 협동 관계로 특화된 것이다.

[정의 3] 구체적 객체 모델 OM_c는 그 모델을 구성하는 요소들이 모두 프로그래밍 언어 L_{obj}로 변환 가능한 모델이다. 즉 OM_c는 튜플 〈CL, GE, AG, CO〉로 정의된다.

[정의 3]은 본 논문에서 제시하고자 하는 변환 기법의 목표인 구체적 객체 모델을 정의하는 것으로 구체적 객체 모델은 클래스 정의, 일반화, 집합화, 협동 관계로만 표현되어야함을 의미한다. 이러한 구체적 객체 모델은 앞에서 논의한 바와 같이 객체 지향 언어로 쉽게 변환될 수 있다. 아울러 본 논문에서 제시하고자 하는 변환 기법의 대상인 추상적 객체 모델의 정의는 다음과 같다.

[정의 4] 추상적 객체 모델 OM_a는 그 모델을 구성하는 요소에 결합 관계가 반드시 포함된 모델이다. 이때 결합 관계는 클래스 객체를 원소로 하는 순서 튜플(ordered tuple)의 집합인 이진 결합(binary association) AS_2와 삼진 결합(ternary association) AS_3의 합집합으로 정의된다.

$AS_2 = \{(x, y) | x \in CL \wedge y \in CL \wedge$ x와 y의 관계가 GE, AG, CO의 의미를 갖지 않음$\}$

$AS_3 = \{(x, y, z) | x \in CL \wedge y \in CL \wedge z \in CL \wedge$ x와 y와 z의 관계가 GE, AG, CO의 의미를 갖지 않음$\}$

더욱이 이들 결합 관계에는 다중성(multiplicity), 링크 속성(link attribute), qualification의 추가 요소가 포함되며 이들은 OMT 표기법에 의해 표현된다.

논리적인 관점에서 결합 관계는 4차 이상의 관계를 가질 수 있다. 그러나 실제적인 객체 모델의 작성 시에 4차 이상의 결합 관계는 구현 관점에서 문제점을 갖기 때문에 모델링 되지 않으며 이진 관계나 삼진 관계로 환원되어야 한다. 따라서 본 논문에서는 4차 이상의 결합 관계는 무시한다.

객체-지향 분석 및 설계 시의 초반기에 이루어지는 대부분의 작업은 문제 영역에서 발견되는 각종 객체를 클래스로 추상화시키고 객체간의 관계를 문제 영역의 관점에서 결합 관계로 추상화시키는 것이다. 따라서 시스템 분석 초기에 만들어지는 객체 모델은 다수의 결합 관계를 포함하기 때문에 매우 추상적이다. 그런데 일반적인 결합 관계는 객체 지향 프로그램으로 직접적으로 변환될 수가 없기 때문에 구현적 관점을 고려한 설계 작업이 필요하다. 따라서 본 논문에서 추구하는 변환 기법은 결합 관계가 다수 포함되어 있는 추상적 객체 모델을 [정의 3]에서의 구성 요소만으로 표현되는 구체적 객체 모델로 변환하고자 하는 것이다. 이러한 변환의 주된 작업은 결합 관계의 다중성, 링크 속성, qualification을 고려하여 결합 관계를 집합화와 협동 관계로 변형하는 것이다. 이러한 과정을 통하여 작성되는 구체적 객체 모델은 추상적 객체 모델에 비하여 매우 쉽게 객체 지향 프로그램으로 표현될 수 있다.

2.2 기본 클래스의 정의

추상적 객체 모델의 구체화를 위해서는 문제 영역에 관계없이 항상 필요한 범용적 목적의 기본 클래스가 정의되어 있어야 한다. 이들 기본 클래스의 목적은 하나의 객체가 다수의 다른 객체와 연관을 맺고있는 상황을 표시하기 위해 사용되는 것으로 다형 리스트(polymorphic list)의 역할을 한다. 이들 기본 클래스에는 다중적인 관계를 구현적 관점에서 지원하기 위한 List 클래스와 집합 기능을 처리하는 Set 클래스, 다수의 객체를 처리하되 순서가 정렬되어 있는 것을 보장하는 OrderedList 클래스가 포함된다. 본 절에서는 이들에 대하여 기술한다.

2.2.1 List 클래스

List 클래스는 다수의 객체를 저장하기 위한 추상 클래스이다. List 클래스는 추상 클래스이기 때문에 응용 프로그램 상에서 직접적으로 인스턴시에이션(instantiation)되지 않으며 응용 목적에 맞는 다형 리스트의 구현을 위한 베이스 클래스(base class)로서만 사용된다. 예를 들어 List 클래스는 FigureList, PersonList, WindowList 등과 같이 응용 목적에 맞는 구체 클래스(concrete class)의 베이스 클래스로 사용된다. (그림 5)는 List 클래스의 명세(specification)이다.

```
typedef Object* AnyPointer;
class ListNode {
private : // data members
        AnyPointer _pointer;
        ListNode* _prev;
        ListNode* _succ;
public : // member functions
        ListNode();
        ~ListNode();
        ListNode(AnyPointer ptr);
        void insert(ListNode* node);
        void append(ListNode* node);
};

class List {
protected : // data members
        ListNode* _header;
        ListNode* _current;
        int _noflist;
        List();
        void insert(AnyPointer ptr);
        void append(AnyPointer ptr);
        Boolean inlist(AnyPointer ptr);
        void remove(AnyPointer ptr);
        AnyPointer init();
        AnyPointer next();
public:
        virtual ~List();
        int noflist();
        Boolean empty();
        void clear();
};
```

(그림 5) List 클래스의 명세
(Fig. 5) The specification for List class

본 논문에서 가정하는 추상 클래스인 List 클래스는 ListNode 클래스의 데이타 멤버에서 알 수 있듯이 이중 연결 리스트(doubly linked list)를 기반으로 작성된 것이다. List 클래스에서 정의된 대부분의 멤버 함수들은 일반적으로 잘 이해되고 있는 함수들이며 이중에서 특히 init() 함수와 next() 함수는 리스트의 멤버들을 traverse 하기 위해 사용되는 함수들이다. (그림 5)의 List 클래스는 추상 클래스라는 점에서 다음 특징을 갖는다. 우선 List 클래스에 정의되어 있는 대부분의 멤버 함수들이 protected 부분에 정의되어 있다. 이는 List 클래스가 구체적인 객체를 인스턴시에이션하지 않기 때문이며 List 클래스가 저장하는 객체의 형이 아직 구체화되어 있지 않기 때문이다. 따라서 이 List 클래스가 다루는 객체의 형은 AnyPointer로 가정되어 있으며 FigureList, PersonList와 같은 구체 클래스가 만들어질 때 리스트에 포함될 객체의 실제 형이 정해진다. 따라서 FigureList, PersonList와 같은 구체 클래스는 List 클래스의 객체 형에 대하여 type casting을 담당하는 방식으로 정의된다. 예를 들어, 그래픽 도구의 구현 시에 필요한 FigureList의 정의는 (그림 6)과 같이 이루어진다.

```
class FigureList : public List {
public : // member functions
        FigureList() : List() {}
        virtual ~FigureList() {}
        void insert(Figure* ptr) {
                List::insert((AnyPointer)ptr);
        }
        void append(Figure* ptr) {
                List::append((AnyPointer)ptr);
        }
        Boolean inlist(Figure* ptr) {
                return List::inlist((AnyPointer)ptr);
        }
        void remove(Figure* ptr) {
                List::remove((AnyPointer)ptr);
        }
        Figure* init() {
                return (Figure *)List::init();
        }
        Figure* next() {
                return (Figure *)List::next();
        }
};
```

(그림 6) FigureList 클래스의 정의
(Fig. 6) The definition of FigureList class

(그림 6)에서와 같이 응용 목적에 맞는 구체적인 다형 리스트들은 리스트가 처리하고자 하는 객체의 형에 대한 type casting작업을 담당하는 방식으로 쉽게 정의될 수 있다. 예를 들어 PersonList 클래스는 Person 형과 AnyPointer 형 사이의 type casting을 수행하도록 작성되며, WindowList 클래스는 Window 형과 AnyPointer 형 사이의 type casting을 수행하도록 작성된다.

본 논문에서와 같은 목적의 추상 클래스를 정의하기 위해서 C++의 템플리트(template) 기능을 이용할 수도 있다. 그러나 추상 클래스의 정의 시에 템플리트를 사용하게 되면 List 템플리트의 인스턴시에이션 결과인 List<Figure>, List<Person>, List<Window> 클

래스의 멤버 함수들이 List클래스의 멤버 함수와 일치해야하는 제약이 생기므로 소프트웨어의 유연성(flexibility)이 다소 떨어진다. 즉 본 논문의 방식으로 정의된 클래스에서는 FigureList::draw()와 같은 함수의 추가 정의가 용이하지만 템플리트를 사용하는 경우에는 그렇지 않다. 예를 들어 Rational 사에서 제작된 Rational Rose와 같은 CASE 도구는 C++ 코드의 자동 생성 시에 결합 관계의 처리를 위하여 UnboundedSetByValue나 UnboundedSetByReference와 같은 템플리트를 사용하고 있다. 그런데 이러한 템플리트를 사용하여 생성된 코드는 동적 바인딩과 다형 리스트의 적용이 어려울 뿐더러 복잡하고 비효율적인 클래스 인터페이스의 생성을 야기시킨다. 따라서 효율적인 클래스 인터페이스를 정의하기 위해서는 상속을 이용하여 컨테이너 클래스들을 만들어 주는 것이 바람직하다. 다음은 FigureList 객체에 포함된 모든 그림들을 그리는 멤버 함수 draw()의 예이다.

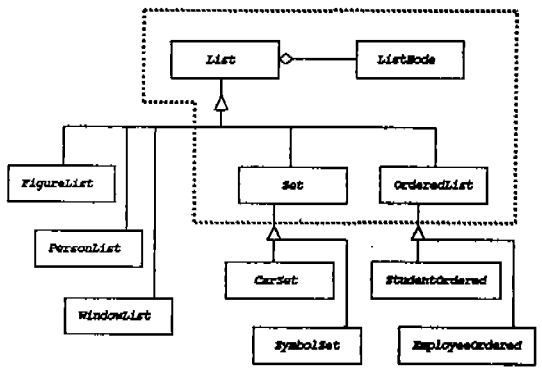
```
void FigureList::draw() {
    Figure *tmp=init();
    while(tmp != NIL) {
        tmp→draw();
        tmp=next();
    }
}
```

이 함수에서 알 수 있듯이 템플리트를 사용하지 않기 때문에 파생 클래스에 대한 멤버 함수의 추가가 용이함을 알 수 있다. 물론 본 논문의 방식으로 추상 클래스를 정의하게 되면 구체 클래스의 정의를 위하여 파일을 복사하거나 코딩을 하기 위한 부담이 늘어나지만 이러한 문제도 클래스의 복사 기능을 제공하는 OODesigner와 같은 CASE 도구를 이용하면 쉽게 완화될 수 있다.

2.2.2 기타의 범용 클래스

추상적 객체 모델의 구체화를 위하여 추가로 필요한 클래스는 Set 클래스와 OrderedList 클래스이다. 이들 클래스는 자료 구조상의 동일성으로 인하여 앞 절에서 논의한 List의 파생 클래스로 정의될 수 있다. 물론 Set과 OrderedList 클래스도 추상 클래스의 역할을 수행하기 때문에 대부분의 멤버 함수가 protected 부분에 정의되어야 하며 구체적 객체에 대한 형이 결정되지 않으므로 객체의 형이 AnyPointer로 취급되고 있다. List, Set, OrderedList 모두 컨테이너(container) 클래스로서의 역할을 함에도 불구하고 이들이 각각 구분되어야하는 점은 컨테이너에 객체를 입력할 때 서로 다른 제약 조건을 갖기 때문이다. 즉 Set 클래스의 insert() 멤버 함수는 객체를 리스트에 집어넣을 때 같은 객체가 중복되어 입력되는가를 검사하며 OrderedList 클래스의 insert() 멤버 함수는 객체를 리스트에 집어넣을 때 객체의 키(key) 값을 기준으로 정렬되어 입력되도록 한다. 예를 들어 다음은 이들 두 클래스의 insert() 멤버 함수 정의이다.

```
void Set::insert(AnyPointer ptr){
    if (List::inlist(ptr)==TRUE) return;
    List::insert(ptr);
}
void OrderedList::insert(AnyPointer ptr) {
    AnyPointer tmp=init();
    while(tmp !=NIL) {
        if (tmp→key()>ptr→key()) break;
        tmp=next();
    }
    ListNode *node=new ListNode(ptr);
    node→insert(_current);
}
```



(그림 7) 컨테이너 클래스들의 상속 트리
(Fig. 7) Inheritance tree for container classes

Set과 OrderedList 클래스의 명세부는 List 클래스의 명세부와 유사한데 Set 클래스의 경우에 집합 연산을 위한 멤버 함수들이 추가로 정의되며 OrderedList 클래스 경우에 인덱스 연산과 min(), max() 멤버 함수 등이 추가로 정의된다. (그림 7)은 본 절에서 논의한 클래스들의 상속 관계를 나타내는 것으로 점선 안에 표시된 클래스들은 기본적인 추상 클래스들이며 나머지는 응용 목적에 따라 정의될 수 있는 구체 클래스들이다.

3. 변환 기법

본 장에서는 결합 관계가 포함된 추상적 객체 모델을 구체화시키는 변환 기법에 대하여 논한다. 이러한 변환을 위하여 결합 관계에 대해서 고려되어야 할 사항은 다음과 같은 경우로 구분된다.

- 결합 관계가 이진 관계인가 혹은 삼진 관계인가?
- 결합 관계의 다중성은 어떻게 설정되었는가?
- 결합 관계에 링크 속성이 포함되어 있는가?
- 결합 관계에 qualification이 명시되어 있는가?
- 결합 관계에 순서나 집합에 대한 제약 조건이 명시되었는가?
- 두 객체가 결합 관계를 갖는 경우 어떠한 경로로 관계에 대한 탐색이 이루어지는가?

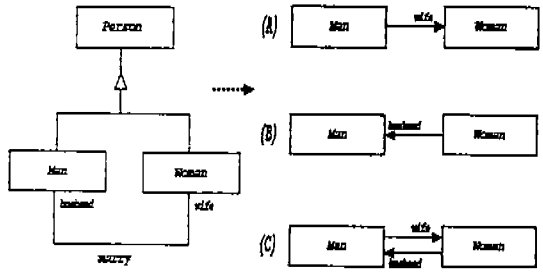
특히 마지막에 언급된 정보는 데이타 항로(data navigation path)와 관련되는 것으로 이 정보에 의해 협동 관계의 방향성이 결정된다. 앞에서 언급된 사항은 객체 모델의 변환 시에 컨테이너 클래스의 종류 및 채택 여부, 컨테이너 클래스에 대한 객체 입력과 삭제 시의 제약 조건, 협동 관계의 방향성 등을 결정하는 요소가 된다. 따라서 본 장에서는 이러한 경우에 대하여 결합 관계를 집합화와 협동 관계로 환원하는 방법에 대하여 기술한다.

3.1 이진 결합 관계의 변환

추상적 객체 모델에서 가장 자주 발생하는 결합 관계가 이진 결합 관계이다. 이는 시스템의 본질적인 특성에 기인한다. 즉 시스템은 객체를 vertex로 하고 객체 간의 관계를 edge로 하는 그래프로 이해될 수 있기 때문에 시스템에 대한 모델에는 결합 관계가 다수 인식되는 것이다. 결합 관계는 요구 사항 명세서에서 동사의 형태로 나타나는데, 예를 들어 "남자와 여자가 결혼하다", "회사가 사원을 고용하다", "비행기에 승객이 탑승하다" 등이 모델링 시에 결합 관계로 표현된다. 이러한 결합 관계는 매우 추상적이기 때문에 객체 지향 프로그램으로 직접 변환되기 어려우며 본 논문에서 제시하는 방식으로 구체화되어야 한다. 결합 관계의 변환을 위해서는 다중성, 링크 속성, qualification 등이 고려되어야 하는데 다음 절 부터는 이들 요소의 경우에 따른 변환 방식을 기술한다.

3.1.1 다중성이 1:1인 경우

본 절에서 다루는 다중성이 1:1인 경우는 한쪽 객체가 선택적인(optional) 경우를 포함한다. 즉, 0:1, 1:0인 경우에 대하여도 같은 변환 규칙이 적용된다. 다중성이 1:1인 경우의 변환 규칙은 결합 관계를 단순히 협동 관계로 변환하는 것이다. 결합 관계를 협동 관계로 변환하면 결합 관계에서 목적어의 역할을 하는 임무(role)가 주어 객체의 데이타 멤버에 포인터 형태로 추가된다. (그림 8)는 1:1 결혼 관계를 협동 관계로 변환한 예이다.

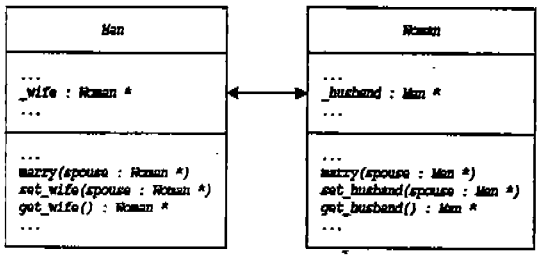


(그림 8) 1:1 결합 관계의 변환
(Fig. 8) A transformation for 1:1 association

(그림 8)에서와 같이 1:1 결합 관계의 변환은 데이타 항로에 따라 세 가지 방식으로 이루어질 수 있다. 데이타 항로는 결합 관계와 관련되는 정보를 얻기 위하여 어떠한 객체를 주로 접근하는가에 대한 정보이다. 데이타 항로는 응용 목적에 의존하게되기 때문에 (그림 8)의 세 가지 변환 중에서 어떠한 것을 선택하는가 하는 결정은 요구 사항의 분석 결과에 따라 다르다. 예를 들어 (그림 8)의 경우에 만약 "어떠한 남자

의 부인은 누구인가 ?"와 같은 질의가 자주 발생하는 경우에는 (A) 방식을 선택하는 것이 바람직하며, "어떠한 여자의 남편은 누구인가 ?"하는 질의가 자주 발생하는 경우에는 (B) 방식을 선택하는 것이 좋다. 만약 두 가지 형태의 질의가 고르게 발생하는 경우에는 (C) 방식을 선택하면 된다. (C) 방식이 가장 바람직한 질의 결과를 제공하기는 하지만 이 경우는 똑같은 정보를 중복 기록하는 문제점을 가지고 있으며 아울러 결합 정보의 입력과 삭제를 위한 부담이 상대적으로 크다. (그림 9)는 (C) 방식을 채택할 경우에 Man 클래스와 Woman 클래스에 필요한 자원을 보여주는 그림이다. 만약 남자 객체 a와 여자 객체 b가 결혼을 하는 경우에 "a→marry(b);" 실행문에 의하여 두 객체가 결혼하였다는 정보가 저장된다. 이때 Man 클래스의 marry() 함수는 다음과 같이 구현될 수 있다.

```
void Man::marry(Woman *spouse) {
        _wife=spouse;
        spouse→set_husband(this);
}
```



(그림 9) Man 클래스와 Woman 클래스의 정의
(Fig. 9) A specification for Man and Woman classes

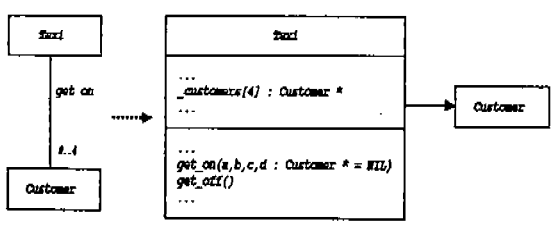
Woman 클래스의 marry() 함수도 비슷한 방식으로 구현될 수 있다. Man과 Woman 클래스의 marry() 함수 의미는 실질적으로 관계 정보를 입력하는 의미를 갖는다. 추상적 모델의 구체화 시에 고려되어야 하는 또 하나의 중요한 관점은 결합 정보의 삭제 방식에 관한 것이다. 만약 어떠한 부부가 이혼하는 경우에는 결혼 관계 정보가 삭제되어야 한다. 이러한 경우를 위하여 Man 클래스에 divorce()라는 멤버 함수를 추가하는 경우 이 함수의 구현은 다음 방식으로 이루어질 수 있다.

```
void Man::divorce() {
        _wife→set_husband(NIL);
        _wife=NIL;
}
```

본 절의 앞부분에서 언급하였듯이 1:1 결합 관계는 한쪽 객체가 선택적인 경우를 포함한다. 즉 한쪽 객체의 다중성이 0인 경우에도 본 절에서 논한 것과 똑같은 방식으로 변환된다. 단지 상대 객체를 가리키는 포인터 값이 NIL인 경우에 상대 객체의 다중성을 0으로 해석하면 된다. 예를 들어, 어떠한 남자 객체가 총각인 경우에 배우자를 가리키는 데이타 멤버 _wife는 NIL 값을 갖는다. 결론적으로 1:1 결합 관계는 협동 관계로 표현될 수 있으며 협동 관계를 고려한 각 클래스의 설계 시에 관계 정보의 입력과 삭제를 위한 멤버 함수의 추가가 필요하다.

3.1.2 다중성이 1:N인 경우 (N 값이 상수인 경우)

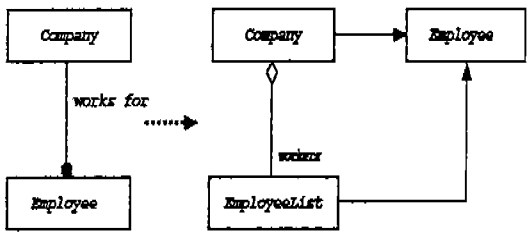
다중성이 1:N이면서 N 값이 상수인 경우의 변환은 다중성이 1:1인 경우와 같이 협동 관계로 변환 가능하다. 그런데 다중성이 1:N 경우에는 데이타 항로의 주된 객체가 다중성이 1인 객체로 정해진다. 따라서 다중성이 1인 주 객체의 데이타 멤버 부분에 포인터 배열을 이용하여 상대 객체의 포인터를 N 개 설정함으로써 변환을 이룰 수 있다. 예를 들어 (그림 10)은 다중성이 1:4인 관계로서 "택시에 최대 4 명의 손님이 탈 수 있다"는 관계를 변환한 예이다. 이 그림에서 Taxi 클래스에 포함된 get_on()과 get_off() 멤버 함수는 각각 관계 정보의 입력과 삭제를 담당하는 함수이다.



(그림 10) 1 : 4 결합 관계의 변환
(Fig. 10) A transformation for 1 : 4 association

3.1.3 다중성이 1:N인 경우 (N 값이 상수가 아닌 경우)

다중성이 1:N이면서 N의 값이 고정적이지 않은 경우에는 결합 관계를 변환하기 위하여 컨테이너 클래스를 필요로 한다. 이 경우에도 데이타 항로의 주된 객체가 다중성이 1인 객체로 정해지며, 다중성이 1인 주 객체의 데이타 멤버에 컨테이너 객체를 정의하고 상대 객체들에 대한 포인터들을 컨테이너 객체에 저장함으로써 결합 관계를 처리할 수 있다. 예를 들어 1:N 결합 관계는 (그림 11)과 같이 집합화와 협동 관계로 환원된다. (그림 11)은 "한 회사에서 다수의 사원이 일한다"는 관계를 변환한 것이다.



(그림 11) 1:N 결합 관계의 변환
(Fig. 11) A transformation for 1:N association

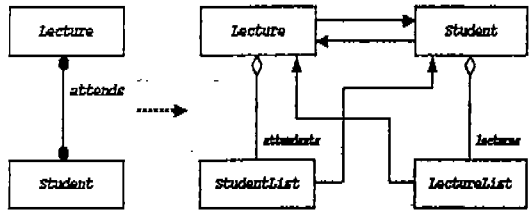
(그림 11)에서 EmployeeList 클래스는 응용 목적에 따라 정의되는 컨테이너 클래스로서 2.2 절에서 논의한 List 클래스로 부터 파생된 것이다. EmployeeList 클래스 객체는 다수의 사원을 저장하고 관리하는 임무를 맡는다. EmloyeeList 객체를 포함하는 Company 클래스의 개략적인 정의는 다음과 같다.

```
class Company {
        ...
        EmployeeList _workers;
public:
        ...
        void insert(Employee * employee) {
                _workers.insert(employee);
        }
        void remove(Employee * employee) {
                _workers.remove(employee);
        }
};
```

이 정의에서 insert()와 remove() 함수는 관계 정보의 입력과 삭제를 위한 멤버 함수의 역할을 하며 함수 본체에서 알 수 있다시피 부속 객체의 멤버 함수를 이용한 delegation 방식으로 구현될 수 있다. 이와 같은 1:N 결합 관계의 변환은 다중성이 선택적인 경우(즉, 1:0인 경우)를 내포하고 있다. 즉 어느 회사에 사원이 한 명도 없는 상황은 _workers 객체가 비어있도록 함으로써 표현할 수 있다.

3.1.4 다중성이 N:M인 경우

결합 관계의 다중성이 N:M인 경우는 다중성이 1:N인 경우의 확장된 방식으로 변환 가능하다. 즉 다중성이 1:N인 경우에는 데이타 항로의 주 객체가 한쪽 객체로만 한정되지만 N:M인 경우에는 양쪽 객체가 모두 데이타 항로의 주 객체가 될 수 있다. 따라서 N:M 결합 관계의 변환은 (그림 11)을 확장하여 이룰 수 있다. 즉, 두 개의 컨테이너 클래스를 사용하여 대칭적인 협동 관계를 맺도록 하면 된다.



(그림 12) N:M 결합 관계의 변환
(Fig. 12) A transformation for N:M association

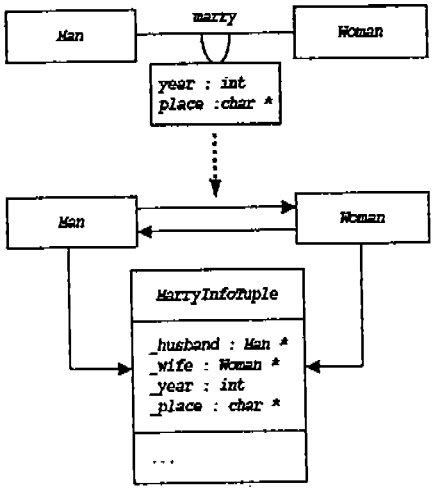
(그림 12)는 수강 과목과 학생간의 관계를 나타내는 것으로 이러한 변환의 예이다. 이 그림에서 결합 관계의 다중성이 N:M으로 설정된 이유는 한 과목에 다수의 학생이 수강할 수 있으며, 한 학생이 다수의 과목을 수강할 수 있기 때문이다. N:M 결합 관계의 변환 시에 주의해야할 사항은 관계 정보의 입력 및 삭제 시에 양쪽 객체의 컨테이너에 대하여 정보의 입력 및 삭제를 같이 수행해야 한다는 것이다. 예를 들어 Student 클래스의 insert(), remove() 멤버 함수는 다음 방식으로 구현될 수 있다.

```
void Student::insert(Lecture * a_lecture) {
        _lectures. insert(a_lecture);
```

```
        a_lecture->insert(this);
}
void Student::remove(Lecture *a_lecture) {
        _lectures. remove(a_lecture);
        a_lecture->remove(this);
}
```

3.2 링크 속성을 고려한 변환

링크 속성은 결합 관계의 인스턴스에서 추가되는 정보를 표현하기 위한 표기법이다. 예를 들어 "남자 a가 여자 b와 1996년 서울에서 결혼하였다"와 같은 정보를 표현하기 위해 결합 관계에 부수적으로 링크 속성이 추가된다. 링크 속성은 C++ 프로그램에 의하여 직접적으로 표현될 수 없기 때문에 구체적 객체 모델로 변환되어야 한다. 데이타 베이스에 대한 응용시에 링크 속성은 관계를 나타내는 테이블에 추가되는 속성으로 표현된다. 예를 들어 이진 결합 관계에 3개의 링크 속성이 정의되는 경우에 이 관계를 표현하기 위한 테이블의 튜플 크기는 다섯 개가 된다. 링크 속성을 반영하는 구체적 객체 모델도 데이타 베이스에 대한 응용에서와 마찬가지로 다수의 필드를 갖는 튜플을 표현할 수 있는 클래스를 정의함으로써 작성될 수 있다. 즉 이진 결합 관계에 n 개의 링크 속성이 추가되면 이 n +2 개의 데이타 멤버를 갖는 튜플 클래스를 정의함으로써 모델의 구체화가 가능하다. (그림 13)은 링크 속성이 존재하는 결혼 관계에 대한 변환 예이다.



(그림 13) 링크 속성의 변환
(Fig. 13) A transformation for link attribute

(그림 13)의 MarryInfoTuple 클래스의 주된 멤버 함수들은 데이타 멤버 값을 알아내기 위한 get_husband(), get_wife(), get_year(), get_place() 등의 엑세스 함수(access function)들이다. 이러한 변환 이후에 결혼 관계를 입력하기 위하여 사용될 수 있는 멤버 함수 marry()와 관계 삭제를 위한 멤버 함수 divorce()를 포함하는 Man 클래스의 개략적인 정의는 다음과 같다.

```
class Man {
        ...
        MarryInfoTuple *_marry_info;
public:
        ...
        void marry(Woman *spouse, int when, char *where) {
                _marry_info = new MarryInfoTuple
                                (this, spouse, when, where);
                spouse→set_marry_info(_marry_info);
        }
        void divorce() {
                _marry_info→get_wife()→set_marry_info(NIL);
                delete _marry_info; _marry_info = NIL;
        }
};
```

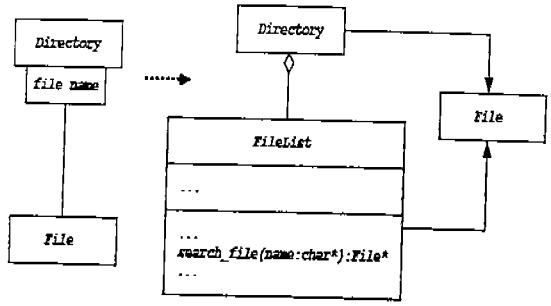
(그림 13)의 변환은 결합 관계의 다중성이 1:1인 경우이다. 만약 다중성이 N:M인 결합 관계에 링크 속성이 추가되는 경우에는 튜플 객체를 저장하는 리스트 클래스가 필요하다. 예를 들어 "다부 다처제"가 허용되는 결혼 관계의 경우에는 컨테이너 클래스인 MarryInfoTupleList 객체가 필요하며 앞에서 언급한 Man 클래스의 개략적인 정의는 다음과 같이 변경되어야 한다.

```
class Man {
        ...
                MarryInfoTupleList _marry_info_list;
public:
        ...
```

```
    void marry(Woman *spouse, int when, char *where) {
        MarryInfoTuple *a_tuple=new MarryInfo-
            Tuple(this, spouse, when, where);
        _marry_info_list. insert(a_tuple);
        spouse→insert(_marry_info);
    }
    void divorce(Woman *spouse) {
        MarryInfoTuple *found = _marry_info_list.
            search_tuple(this, spouse);
        _marry_info_list.remove(found);
        spouse→remove(found);
    }
};
```

3.3 Qualification을 고려한 변환

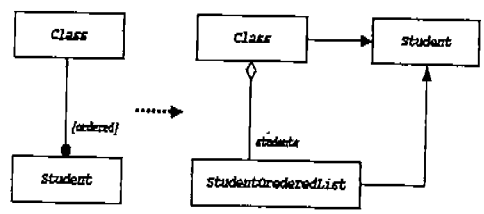
OMT 객체 모델에서 qualification의 사용 목적은 두 객체 간의 결합 관계에서 데이타 항로 상의 주 객체로 부터 상대 객체를 인식할 수 있는 키를 정의함으로써 1:N의 다중성을 1:1의 다중성으로 환원하기 위한 것이다. 예를 들어 디렉토리(directory)와 파일간의 관계에서 qualification의 사용 목적을 알 수 있다. 하나의 디렉토리에는 여러 개의 파일이 저장될 수 있기 때문에 디렉토리와 파일간의 결합 관계 다중성은 1:N이다. 그런데 한 디렉토리에 소속된 파일 이름들은 유일하기 때문에 이 관계는 파일 이름을 qualifier로 사용하면 (그림 14)의 왼쪽에 있는 모델처럼 1:1 결합 관계로 표현될 수 있다. 즉 이 모델의 qualification이 의미하는 내용은 "어떠한 디렉토리에서 특정한 파일 이름을 갖는 파일은 하나 뿐이다"라는 것이다. qualification은 데이타 베이스에 대한 응용 시에 탐색하고자 하는 객체의 키를 설정하기 위해 사용된다. 즉 (그림 14)의 경우 파일 객체를 위한 키는 디렉토리의 키와 파일 이름의 조합으로 결정된다. 그러나 객체 지향 프로그래밍 시에는 qualification이 차지하는 비중이 매우 적다. 즉 객체 지향 프로그래밍 시에는 다중성이 1:N에서 1:1으로 환원되는 것이 의미가 없으며 결국 1:N인 경우와 동일하게 다루어져야 한다. (그림 14)의 오른쪽 부분은 qualification을 구체화한 경우로 다중성이 1:N인 것과 같은 방식으로 변환되기 때문에 FileList라고 하는 컨테이너 클래스를 필요로 한다. 대신에 qualification에 대한 의미를 반영하기 위하여 디렉토리 객체의 컨테이너에서 파일을 탐색할 때 파일 이름을 키로 이용하는 멤버 함수가 FileList 클래스에 추가된다.



(그림 14) qualification의 변환
(Fig. 14) A transformation for qualification

3.4 기타 제약 조건을 고려한 변환

추상적 객체 모델의 결합 관계에는 정렬과 집합에 관한 제약 조건이 명시될 수 있다. 이러한 제약 조건을 고려하여 추상적 객체 모델을 구체화하기 위해서는 2.2.2 절에서 논한 추상 클래스인 OrderedList, Set으로 부터 파생된 컨테이너 클래스가 필요하다. (그림 15)는 "학급에 소속된 학생들은 학번에 의하여 순서대로 정렬되어 있다"는 관계를 변환한 것이다. 이러한 변환은 (그림 11)과 같이 다중성이 1:N인 경우의 변환과 유사하며 단지 컨테이너 클래스가 OrederedList로 부터 파생 것이라는 점만 다르다. 만약 결합 관계에 대한 제약 조건이 집합인 경우는 StudentSet과 같은 컨테이너가 사용된다.

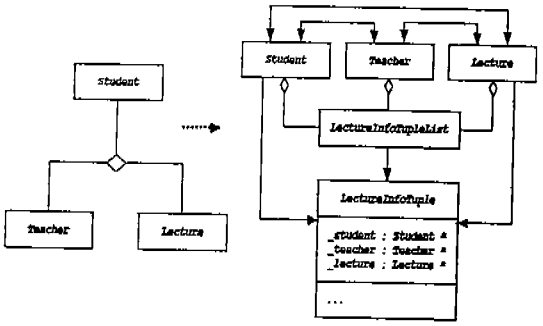


(그림 15) 제약 조건을 고려한 변환
(Fig. 15) A transformation concerning constraint

3.5 삼진 결합 관계의 변환

본 장에서 다룰 마지막 내용은 삼진 결합 관계의

변환에 관한 것이다. 객체 모델의 작성 시 삼진 결합 관계의 사용은 많은 주의를 요한다. 즉 삼진 결합 관계가 이진 결합 관계로 분해될 여지는 없는가를 확인해야 하며 삼진 결합 관계가 링크 속성을 포함하는 이진 결합 관계가 아닌가 하는 점을 검사하여야 한다. 삼진 결합 관계의 변환은 링크 속성을 고려한 변환과 동일하다. 즉 삼진 결합 관계를 표현하기 위해서는 3 개의 객체를 가리키는 포인터의 튜플이 필요하며 이러한 튜플은 링크 속성을 변환할 때의 튜플과 비슷한 구조를 갖는다. (그림 16)은 "학생 객체 a가 교수 객체 b로 부터 과목 c를 수강한다"는 관계를 변환한 것이다.



(그림 16) 삼진 결합 관계의 변환
(Fig. 16) A transformation for ternary association

(그림 16)의 변환은 다소 복잡해 보이지만 본질적으로 N:M 결합 관계에 링크 속성이 추가된 경우와 변환 방식이 동일하다. 아울러 응용 목적에 따라 데이타 항로의 주 객체가 특정한 객체로 한정되는 경우에는 변환된 모델이 더욱 단순해 질 수 있다. 예를 들어 (그림 16)에서 강의 객체를 통하여 수강 정보를 탐색할 필요가 없는 경우에는 Lecture 클래스에 컨테이너 클래스인 LectureInfoTupleList의 객체가 데이타 멤버로 포함되지 않아도 된다. (그림 16)의 삼진 결합 관계에 대한 정보 입력은 Student 클래스의 attend() 멤버 함수에 의하여 이루어지며 정보 삭제는 cancel() 멤버 함수에 의하여 이루어진다고 할 때 이들을 포함하는 Student 클래스의 개략적인 정의는 다음과 같다.

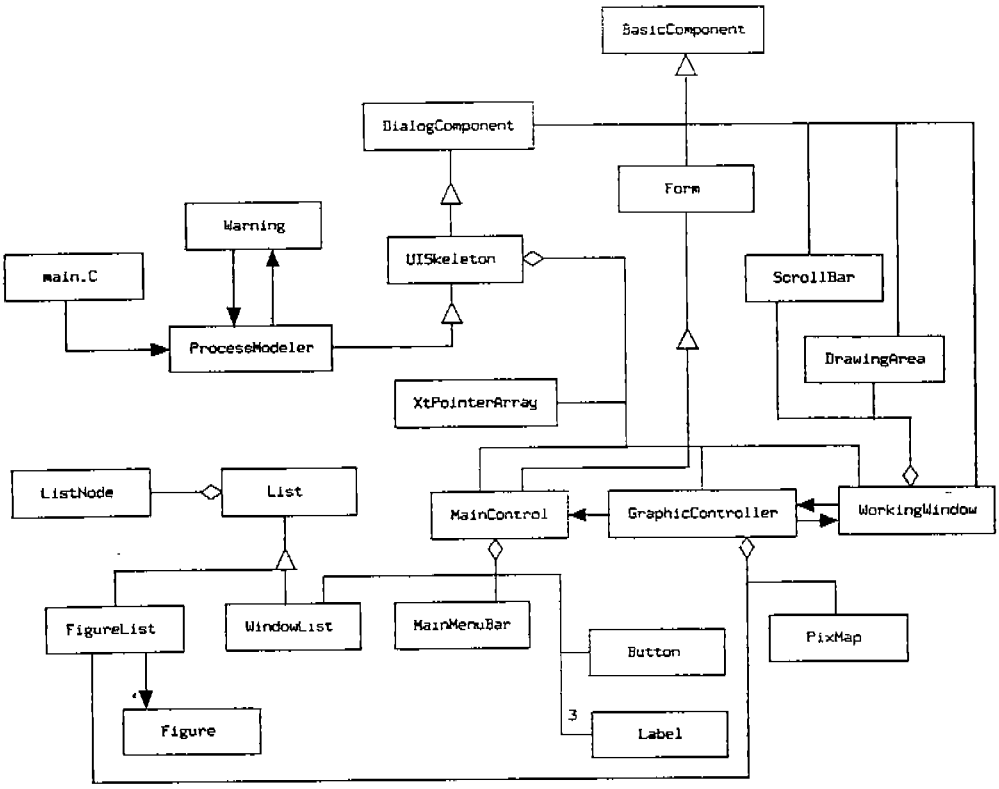
```
class Student {
        LectureInfoTupleList _lecture_info_list;
public:

        void attend(Teacher *teacher, Lecture *lecture) {
                LectureInfoTuple *a_tuple=new Lecture-
                        InfoTuple(this, teacher, lecture);
                _lecture_info_list.insert(a_tuple);
                teacher→insert(a_tuple);
                lecture→insert(a_tuple);
        }
        void cancel(Teacher *teacher, Lecture *lecture) {
                LectureInfoTuple *found=_lecture_info_list.
                        search_tuple(this, teacher, lecture);
                _lecture_info_list.remove(found);
                teacher→remove(found);
                lecture→remove(found);
        }
};
```

4. 변환 기법의 적용 예

본 논문에서 정의된 변환 기법에 의하여 작성되는 객체 모델은 결합 관계를 모두 집합화와 협동 관계로 환원한 것이기 때문에 C++ 프로그램으로 표현하는 것이 매우 용이하다. 이러한 변환 기법은 객체 지향 CASE 도구 OODesigner의 재구조화와 정보 통신 서비스 개발 프로세스 모형화 개발 지원 도구의 일부분인 프로세스 모델러의 설계 과정에 적용되고 있다. (그림 17)은 현재 설계가 진행 중인 프로세스 모델러의 객체 모델 중의 일부분으로 main module 부분을 나타내는 그림이다. 이 그림에서 알 수 있듯이 결합 관계가 전혀 표현되어 있지 않는데 그 이유는 모든 결합 관계가 집합화와 협동 관계로 환원되었기 때문이다. 이 모델에서 사용되고 있는 FigureList, WindowList 와 같은 컨테이너 클래스들은 1:N 결합 관계를 변환하기 위하여 정의된 클래스들이다.

(그림 17)에서 나타나는 핵심적인 클래스의 역할은 다음과 같다. 이 그림에 나타나는 모든 클래스들은 C++ 클래스로서 구현된다.

- main.C: main.C는 클래스가 아니라 C++ 파일 객체이다. main.C는 main() 함수를 포함하는 것으로



(그림 17) 프로세스 모델러를 위한 객체 모델의 주 모듈
(Fig. 17) The main module of object model for process modeler

ProcessModeler 객체를 인스턴시에이션시키고 이를 수행시킨다.

- ProcessModeler: 이 클래스는 실질적인 주 프로그램 역할을 하는 클래스로서 대부분의 전역 변수와 정적 함수에 대한 정의를 포함한다.

- UISkeleton: 이 클래스는 ProcessModeler의 기본적인 인터페이스의 속성과 기능을 담당하는 클래스로서 main menu 부분을 위한 MainControl 객체와 각 인터페이스에서의 논리적 기능을 담당하는 GraphicController 객체 그리고 작업 화면을 위한 WorkingWindow 객체로 구성되는 클래스이다. 이 클래스는 X 응용에서 독립적인 셸(shell)의 역할을 수행하는 것으로 재사용 지원을 위한 범용적 목적을 위하여 정의되었다.

- MainControl: 이 클래스는 각 모델러의 상단에 위치하는 주 메뉴와 여러 부속 요소들을 처리하기 위한 컨테이너의 역할을 하는 클래스이다.

- GraphicController: 이 클래스는 논리적인 작업을 위한 핵심적인 클래스로 데이타 자원을 관리하며 대부분의 상위 계층 콜백 함수(callback function)를 처리한다. 이 클래스의 가장 중요한 데이터 멤버는 FigureList로서 이 리스트를 이용하여 각종 표기법에 대한 처리를 수행할 수 있다.

- WorkingWindow: 이 클래스는 그림이 그려질 영역을 관리하는 클래스이다.

5. 결 론

OMT와 같은 표기법을 이용한 객체 지향 모델의 작성이 응용 소프트웨어의 구현을 위한 많은 도움을

제공함에도 불구하고 결합 관계를 포함하는 추상적 객체 모델은 시스템의 구현 시에 상당한 어려움을 야기시킨다. 본 논문에서는 이러한 어려움을 최소화하기 위한 방편으로 추상적 객체 모델을 구체적 객체 모델로 변환하는 기법을 제시하였다. 이를 위하여 본 논문에서는 추상적 객체 모델과 구체적 객체 모델에 대한 정의를 시도하였으며 결합 관계의 여러 상황을 고려한 변환 기법을 제안하였다.

결합 관계를 포함하는 추상적 객체 모델은 결합 관계의 다중성, 링크 속성, qualification, 제약 조건 등에 따라 다양한 방식으로 구체화 모델로 변환 가능하다. 이러한 변환 작업에 의하여 구체적 객체 모델에는 다양한 명칭의 컨테이너 클래스가 추가되어야 함을 알 수 있었으며 아울러 결합 관계 인스턴스의 정보 입력과 정보 삭제를 위한 멤버 함수의 보편적 기능을 파악할 수 있었다. 본 논문에서 제시된 변환 기법은 객체 지향 CASE 도구인 OODesigner 재구조화 과정과 소프트웨어 개발 프로세스 지원 도구인 프로세스 모델러의 설계 과정에서 적용되고 있으며 그 결과로 구체적인 객체 모델의 작성이 가능하였다.

차후의 연구 과제는 이러한 변환을 자동적으로 수행할 수 있는 지능형 CASE 도구를 개발하는 것이다. 이러한 CASE 도구는 시스템 설계자와의 대화를 통하여 객체 모델의 결합 관계를 집합화와 협동 관계로 환원시킬 수 있어야 한다. 아울러 이러한 변환 기법을 실제적으로 최대한 적용함으로써 좀 더 상세한 변환 기법으로 정제해야 할 것이다.

참 고 문 헌

[1] Adele Goldberg and David Robson, 'Smalltalk-80:The Language and its Implementation', Addison-Wesley, 1983.

[2] Anthony I. Wasserman, et al., "An Object-Oriented Structured Design Method for Code Generation," ACM SIGSOFT SE Notes, Vol. 14, No. 1, pp. 32-55, Jan. 1989.

[3] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," Software Engineering Notes, Vol. 11, No. 4, pp. 22-42, 1986.

[4] Bertrand Meyer, 'Object Oriented Software Construction', Prentice-Hall, 1988.

[5] Brad J. Cox, 'Object-Oriented Programming', Addison-Wesley, 1991.

[6] Elliot J. Chikofsky, James H. Gross II, "Reverse Engineering and Design Recovery:A Taxonomy," IEEE Software, Vol. 7, No. 1, pp. 13-17, Jan. 1990.

[7] Grady Booch, 'Object Oriented Design with Applications', Benjamin-Cummings, 1991.

[8] James Rumbaugh, et al., 'Object-Oriented Modeling and Design', Prentice-Hall, 1991.

[9] John G. P. Barnes, 'Programming in Ada', Addison-Wesley, 1989.

[10] Kent Beck, Ward Cunningham, "A Laboratory for Teaching Object Oriented Thinking," OOPSLA '89 Conference Proceedings, pp. 1-6, Oct. 1989.

[11] Matthias Jarke, "Strategies for Integrating CASE Environments," IEEE Software, Vol. 9, No. 2, pp. 54-61, Mar. 1992.

[12] Michael Blaha, et al., "Converting OO Models into RDBMS schema," IEEE Software, Vol. 11, No. 3, pp. 28-39, May 1994.

[13] Patrick A. V. Hall, "Software Components and Reuse-Getting More out of Your Code," Information and Software Technology, Vol. 29, No. 1, pp. 38-43, Feb. 1987.

[14] Peter Coad, Edward Yourdon, 'Object Oriented Analysis', Yourdon Press, 1990.

[15] Rebecca J. Wirfs-Brock, et al., 'Designing Object-Oriented Software', Prentice-Hall, 1990.

[16] Ruben Prieto-Diaz, "Status Report:Software Reusability," IEEE Software, Vol. 10, No. 3, pp. 61-66, May 1993.

[17] Stanley B. Lippman, 'C++ Primer', Addison-Wesley, 1989.

[18] Toby J. Teorey, et al., "A Logical Design Methodology for Relational Database Using the Extended Entity Relationship Model," ACM Computing Survey, Vol. 18, No. 2, pp. 197-222, June 1986.

[19] 김 태균, "C++ 언어에 대한 역공학 도구의 설계

및 구현", 정보과학회논문지(C), 제 1권, 제 2호, pp. 135-145, 1995

[20] 김태균, 우치수, "객체 지향 설계 도구의 설계 및 구현", 정보과학회논문지, 제 21권, 제 5호, pp. 909-921, 1994.

김 태 균

1985년 서울대학교 자연과학대학 졸업 (학사)
1987년 서울대학교 계산통계학과 (석사)
1995년 서울대학교 계산통계학과 (박사)
1988년~현재 부산외국어대학교 컴퓨터 공학과 부교수

임 채 덕

1989년 전남대학교 전산통계학과 졸업 (학사)
1989년~현재 한국전자통신연구원 선임연구원
관심분야: 분산 컴퓨팅, 정보통신, 브레인 컴퓨팅

송 영 기

1977년 서울대학교 계산통계학과 졸업 (학사)
1981년 한국과학기술원 전산학과 (석사)
1977년~1985년 국방과학연구소 선임연구원
1985년~현재 한국전자통신연구원 책임연구원
1993년 3월~1994년 3월 미국 텍사스 알링톤대학 방문연구원

인 소 란

1978년 홍익대학교 전자계산학과 졸업
1982년 홍익대학교 전자계산학과(석사)
1987년 정보처리기술사 취득(전자계산기 조직 응용 분야)
1991년 홍익대학교 전자계산학과(박사)
1978~현재 한국전자통신연구원 S/W공학연구실장 (책임연구원)