해 설

객체지향 프로그래밍 언어의 특성과 그의 비교

하 수 철†

❖ 목 차 ❖

1. 서 론
2. 기본 개념
3. 객체지향 프로그래밍 언어의 종류
4. 객체지향 프로그래밍 언어의 논쟁점
5. 결 론

1. 서 론

객체지향 프로그래밍 언어 (OOPL: Object-Oriented Programming Language)는 1960년대 Simula 프로젝트로 시작되었다. 이 프로젝트의 중심 개념 중의 하나는 객체의 개념을 언어에 통합시키는 것이었다. 이 객체는 실세계 객체와 마찬가지로 어떤 성질을 가진 엔티티(entity)이며, 고유의 방법으로 사건(event)에 반응하는 능력을 가진다는 개념이다. 또, 프로그램은 일련의 객체로 이루어지며, 서로 다른 객체에 반응함으로써 동적으로 반응할 수 있다는 사상이다. 이 개념들이 Simula67[1]에 통합되었다. Simula67은 1970년대에 두 가지 다른 방향의 영향을 주게 된다. 첫 번째는 추상 데이타 형(abstract data type)기법의 개발이고, 또 다른 하나는 객체지향 패러다임 자체의 개발이다. 객체지향 패러다임의 대표적인 개발 방향은 Smalltalk-80[2]에서 정점을 이룬 Dynabook 프로젝트이었다.

1970년대 초에는 객체지향이라 불리는 프로그래밍 언어는 Smalltalk뿐이었고, 이에 관심 있는 사람들의 모든 초점도 OOP에 관해서 였다. 1980년대에 이르러 OOPL의 폭발적인 성장이 있었는데, 이 언어들의 대다수는 Smalltalk 패러다임에 입각하여 정의되었다. 또한 이때에는 객체지향 설계(OOD), 객체지향 요구사항분석(OORA), 객체지향 도메인분석(OODA), 객체지향 데이타베이스 시스템(OODBS)등이 정형화되기 시작하였으며, OOPL의 이해와 사용이 증가함에 따라 객체지향 소프트웨어 공학(OOSE)에 대한 이해가 일반화되기 시작하였다. 1980년대 말에는 OOPL에 대한 증가된 관심뿐만 아니라 객체 지향화의 정도로 주어진 언어를 평가하고, 분류하고, 판별하는 시도가 많아졌다[4,5]. 또한 기존의 프로그래밍 언어에 객체 지향적 특성을 확장하여 제안하기도 하였다[6, 7].

OOP의 주요 목적은 소프트웨어 확장성과 재사용성을 증가시킴으로써 프로그래머 생산성을 향상시키고, 소프트웨어 유지보수의 복잡도와 비용을 제어하는데 있다. 또한 OOP는 추상 데이타형, 클래스, 형 계층구조, 상속성, 다형식 등과 같은 주요개념을 기반으로 하고 있다.

† 종신회원 : 대전대학교 컴퓨터 공학과 교수

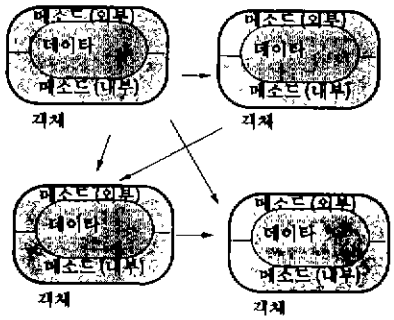
본 논문의 구성은 다음과 같다. 먼저 2장에서는 본 논문에서 분석할 OOPL이 가지고 있는 기본 개념에 대해 논의한다. 3장에서는 OOPL의 분류법, 종류 및 특징을 기술하며, 4장에서는 OOPL에 대한 설계, 구현, 프로그래밍 측면에서의 논쟁점에 관하여 논한다. 5장에서는 본 논문의 결론으로 OOPL의 선택기준을 언급하고 그들의 비교를 한다.

2. 기본 개념

2.1 객 체

객체(object)는 데이타(속성 또는 성질)와 기능적 논리(메소드)들의 집단이다. 데이타는 객체의 상태를 정의하며 메소드(method)는 객체의 행위를 정의한다. 메소드에는 두 가지 종류가 있는데, 그 하나는 객체간의 통신 수단이 되는 인터페이스 메소드이며, 또 다른 하나는 객체행위를 만들어 내지만 객체 외부에서는 접근할 수 없는 내부 메소드이다. 객체의 사용자는 인터페이스 메소드에 관해 알 수 있으며, 내부 메소드는 객체의 설계자에게만 알려진다. 이것은 소프트웨어 모듈에 대한 몇 가지 가정을 갖는다. 즉, 사용자는 인터페이스 메소드에 관한 가정만 하는데, 예를 들어, 무슨 메소드를 사용할 수 있는가? 이들을 통해 통과되는 정보는 무엇인가? 또한 어떤 정보가 호출한 곳으로 반환되는가 등이다.

객체, 메소드, 데이타 그리고 외부와의 관계를 (그림1)에 보인다.



(그림 1) 객체, 메소드, 데이타

객체의 메소드를 호출하는 것을 "객체에 메시지(message)를 보낸다"라 하기도 한다.

각 객체는 주체성(identity)에 의해 유일하게 식별되는데, 이 객체 주체성은 재래식 프로그래밍 언어의 포인터, 운영체제에서의 화일 명의 개념을 확장한 개념이다. 객체의 주체성은 객체의 상태와는 별개로서, 두개의 객체가 같은 상태를 가질 수 있으나 주체성은 서로 다르다.

객체지향 시스템은 객체가 속성을 가지도록 하는데 그 속성자체가 객체자신이다. 이것은 추상 데이타 형(abstract data type)을 통해 달성된다. 추상 데이타 형은 비록 사용자에게 물리적 정보의 저장소를 숨기지만, 사용자가 객체의 속성들을 조작하기 위해 호출할 수 있는 메소드의 공용 집합을 제공한다. 이것은 명확히 정의된 모듈을 장려하고 소프트웨어 시스템에서 전체 복잡도를 경감시킬 수 있는 매우 강력한 기법이다.

데이타와 메소드를 결합시키는 행동을 캡슐화(encapsulation)라 부르며, 어떠한 객체도 다른 객체의 상태를 직접 읽거나 변경시킬 수 없다. 객체의 상태를 접근하는 것은 인터페이스 메소드를 통해서 달성된다. 그렇지만 이때의 접근은 객체 자신이 그것을 허용할 때에만 가능하다. 이러한 데이타 은폐(information hiding) 또는 데이타 추상화는 객체간의 결합성(coupling)을 약화시키는 효과를 갖는다.

2.2 클래스

다수의 유사한 객체들을 일반화된 서술로 같이 묶어 기술할 수 있다. 이것을 클래스(class)라 한다. 각 객체는 클래스의 구체예(instance)가 된다. 클래스는 객체에 의해 사용되는 메소드를 정의하며, 객체가 내포할 수 있는 상태들을 정의하는 데이타 형을 선언한다. 즉, 클래스는 클래스형의 구체예로서 객체들을 생성하기 위해 사용되는 청사진이 된다.

각각의 객체들은 그들 자신의 요구를 만족하기 위해 그들 데이타(instance variable이라고도 부

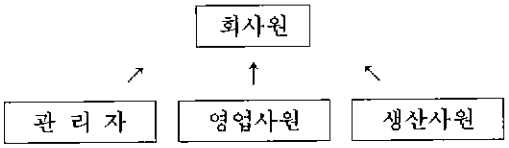
름)를 활용하는 반면, 클래스는 데이타를 정의하기 위해 사용된다. 따라서 대부분의 경우 클래스는 컴파일 시간 개념이다. 객체와 연관된 코드는 클래스에 상주하나, 데이타는 클래스의 구체예, 즉 객체에 상주한다. 이러한 방식으로 같은 형의 객체는 각 객체 구체예에 대한 코드를 중복하는 것이 아니라 클래스의 메소드를 공유하는 것이다.

2.3 상속성

객체들이 서로 같지는 않지만 유사한 속성과 메소드를 가질 경우, 이들 성질을 상속하는 능력이 있다면 유용할 것이다. 상속성(inheritance)은 클래스들 간의 데이타와 메소드의 공유를 허용하는 OOPL의 주요 메커니즘이다. 예를들어 클래스 B가 클래스를 선언할 때 다른 클래스 A의 전부 또는 일부인 구체예 변수와 메소드를 상속받을 수 있다. 이때 B는 A의 부속클래스(subclass)이고, A는 B의 슈퍼 클래스(superclass)가 된다. 상속성은 다음과 같은 몇 가지 형태를 가진다.

2.3.1 단일 상속성

단일 상속성(single inheritance)은 부모-자식 관계성이 부속 클래스와 슈퍼 클래스간에 유지되는 경우이다. 예를 들면 (그림 2)와 같다.

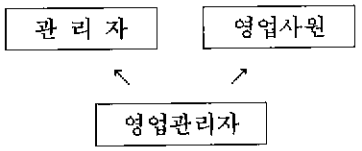


(그림 2) 단일 상속성

2.3.2 다중 상속성

다중 상속성(multiple inheritance)은 한 클래스가 하나 이상의 바로 위 슈퍼클래스로부터 상속받는 것을 허용한다.

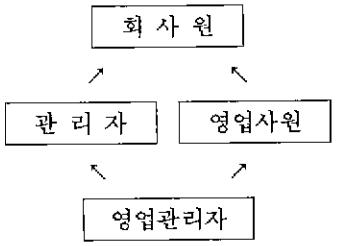
다중 상속성을 정의하는 규칙은 발생할지 모를 혼란을 다루어야 하는데, 그 혼란으로는 같은 이름이 두개의 다른 부모에서 다른 속성 또는 메소드를 표현하기 위해 사용될 수 있다는 혼란과 이들 속성 또는 메소드들이 같은 자식 클래스로 상속될 수 있다는 것이다. 다중 상속성이 객체 지향적 생각을 구성하기 위한 강력한 메커니즘을 제공하지만 복잡성을 증가시킬 수 있다.



(그림 3) 다중 상속성

2.3.3 반복 상속성

반복되는 상속성(repeated inheritance)은 같은 조부모로부터 상속받는 두 부모로부터 또 다시 상속받는 자식 클래스에서 발생한다. 따라서 자식은 잠재적으로 조부모로부터 시작하여 모든 것을 두번 상속받게 된다.

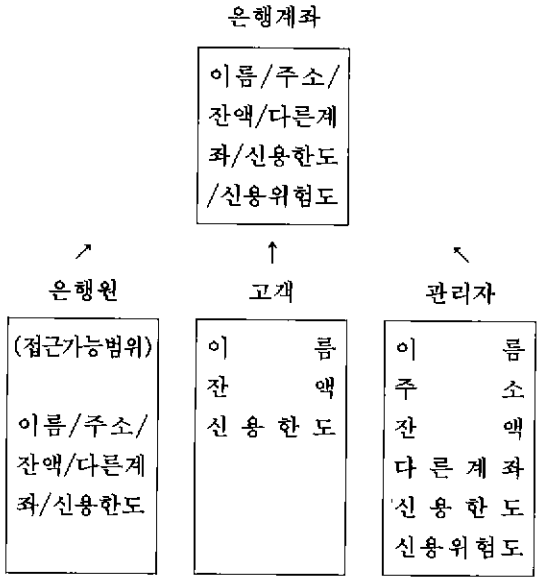


(그림 4) 반복 상속성

부속 클래스(예에서 영업관리자)는 슈퍼 클래스(예에서 관리자)로부터 메소드와 속성을 상속받는데, 부속클래스는 슈퍼클래스 자신이 추가적인 메소드와 속성을 정의한 것과 마찬가지로 슈퍼클래스로부터 상속받은 메소드나 속성을 재정의 할 수 있다.

2.3.4 선택적 상속성

선택적 상속성(selective inheritance)에서 부속 클래스는 슈퍼클래스로부터 메소드의 일부만을 상속받는다. 이 특성은 제한된 정보의 접근만을 사용자가 허용할 때 유용하다.



(그림 5) 선택적 상속성

2.4 메시지 전달과 바인딩

객체는 메시지를 전달함으로써 다른 객체들과 통신을 한다. 메시지는 목적지를 식별하는 이름을 가지고 있을 뿐만 아니라 인수나 매개변수를 갖기도 한다. 이것은 전통적인 함수 호출 또는 프로시저 호출과 유사하다. 그렇지만 메시지는 전통적인 함수 호출에 관련된 것보다 더 복잡한 송신자(sender)로부터 수신자(receiver)까지의 경로를 포함할 수 있다. C나 Pascal과 같은 전통적인 프로그래밍 언어는 송신자가 수신자의 주소를 안다. 따라서 메시지의 바인딩(binding)은 컴파일 시간에 달성된다. 모든 바인딩을 컴파일시간에 완성하는 것은 일관성 대조를 쉽게 하고 형(type)과 관련된 문제에 대해 사용자의 주의를 환기시키는데 도움을 준다.

반면, OOPL에서 하나의 객체로부터 다른 객체로의 메시지 전달은 송신자가 수신자의 주소를 안다는 가정이 항상 가능하지 않기 때문에 더 복잡한 과정이 필요하다. 이 과정은 동적 바인딩(dynamic binding)에 의해 수행된다. 동적 바인딩은 운행(run)시간까지 메시지 이름과 함수와의 연관 관계를 늦춘다고 볼 수 있으며, 이것은 운행시간 효율성의 비용으로 컴파일의 속도를 증가시키 고 재컴파일 없이 근본적인 변경이 가능해야만 언어의 융통성을 향상시킬 수 있다. 메시지가 발송될 때, 송신 객체는 처리를 멈추고 제어를 수신 객체에 전달한다. 수신자가 처리를 완성하면 원래 송신자에게 반응을 보내게 된다. 이것을 동기적 반응(synchronous response)이라 한다. 한편, 다수의 제어 묶음이 있는 곳에서의 반응은 비동기적(asynchronous)이며, 메시지는 처리를 위한 수신자의 대기행렬(queue)에 들어가게 된다. 이때 송신 객체를 클라이언트(client), 수신 객체를 서버(server)라 부르기도 한다.

2.5 다형식

다형식(polymorphism)은 서로 다른 형을 가정하거나 서로 다른 형의 객체를 조작할 수 있는 어떤 객체의 능력이다. 다형식은 같은 코드를 서로 다른 구체예에 재활용되게 하므로 소프트웨어에 대한 적은 변경으로도 그 능력이 증진되게 한다. 다형식에는 다음과 같은 3가지 종류가 있다.

2.5.1 포함 다형식

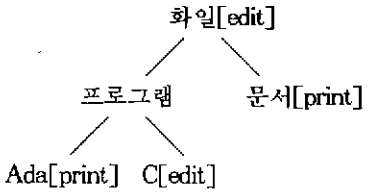
포함 다형식(inclusion polymorphism)은 Eiffel [8], Trellis[9], C++[10]와 같은 강한 형(strongly type)언어에 적용된다. 포함 다형식은 형이론(type theory)의 개념으로 구현되어, 단일 이름이 어떤 일반적인 슈퍼 클래스에 의해 관련되는 여러 다른 클래스들의 객체를 표현할 수 있다. 그러므로 이 형식의 사용이 상속성의 핵심이 된다.

예를 들어 클래스 Dog가 클래스 Animal로부터 상속되었다 하자. 만일 언어가 Dog을 Animal의 부속형(subtype)으로 간주하게 되면, 컴파일러는 Dog의 구체예를 Animal의 구체예가 기대되는 곳 어디에서도 가능하도록 할 것이다. 이것은 동적 형 체계(dynamic typing)가 된다. 식별자의 정적 형은 식별자의 초기에 선언된 형이다. 만일 운행시간에 엔티티가 다른 형의 객체와 연관되게 된

다면, 이 형은 동적 형으로 참조되는 것이다.

2.5.2 연산 다형식

연산 다형식 (operation polymorphism)에서는 부속 클래스에 의한 아무런 관계성도 지니지 않은 다른 클래스들의 객체들 간에 행동이 공유된다.



(그림 6) 포함 및 연산 다형식

이 예에서 형 "화일"의 객체는 "문서" 또는 "프로그램" 화일 중의 하나이고, "프로그램" 화일은 "Ada" 또는 "C" 프로그램 중의 하나이다. 부속 형체계(subtyping)를 통해 edit는 "문서" 화일과 "Ada" 또는 "C"프로그램에 적용된다. 이것이 포함 다형식의 개념이다. 한편, print연산은 "Ada"나 "문서"에는 정의되어 있지만 이 메소드에 대한 두 화일 간에는 부속형 체계의 개념이 존재하지 않는다. 따라서 print연산은 각 호출의 문맥에 따라 해석된다. 이 개념이 연산 다형식이다. 물론 함수 오버로딩(overloading)도 연산 다형식의 한 형식이다.

2.5.3 매개 변수 다형식

매개 변수 다형식(parametric polymorphism)은 포괄성(genericity)이라고 부르기도 한다. 이것은 포괄형(또는 클래스)선언에서 매개변수로서 형을 사용한다. 예를 들면 객체의 다양한 형의 큐를 다루는 클래스가 필요한 경우, 다음 코드는 이러한 구현을 설명한다.

```
class queue[t_obj]
private :
        que-arr: array[1..10] of t_obj ;
public :
        add_item(new_obj: t_obj);
end class
```

매개변수 t_obj는 형 t_obj 상에 매개변수화된 클래스로서 이 형은 응용부문에서 큐가 사용될 때 어떠한 형으로도 대치될 수 있다. 이 정의는 기본적으로 형판(template) 개념이다. 이러한 매개변수 다형식은 코드 공유라는 중요한 이점이 있다. 이것을 지원하지 않는 언어인 경우 사용자는 요구된 큐의 각각 다른 형식에 대해 코드를 중복시켜야 한다.

2.6 클래스 라이브러리, 프레임워크

클래스 라이브러리(class library)에는 일반적으로 유용한 프로그래밍 모듈들이 들어 있으며, 배열, 사전, 테이블 등의 재사용 가능한 자료구조와 선형 운행(traversal), 해싱 등의 메소드들이 그 내용이 된다. 응용부문을 개발하는 것은 클래스 라이브러리를 개발하는 것과 응용부문 구축에 성분요소가 되는 클래스를 사용하는 것으로 달성된다.

프레임워크(framework)는 응용부문의 특정 부류에 대해 특별히 조율된 특정 정의역 클래스 라이브러리로서 응용 부문을 구축하는데 클래스 라이브러리 보다 더 빠르고 용이하다.

2.7 지속성

지속성(persistence)은 객체의 영속성을 말하며, 컴퓨터 메모리에서 접근가능함을 유지하고 공간이 할당되는 시간의 양에 대한 개념이다. 객체가 더 이상 필요 없을 때 할당된 메모리 공간은 소멸되며 회수된다. 메모리 공간의 자동적인 회수를 쓰레기 수집(garbage collection)이라 한다. 지속성 객체(persistent object)는 명시적으로 삭제할 때까지 메모리에 존재하는 객체이다.

3. 객체지향 프로그래밍 언어의 종류

3.1 분류법

객체지향 프로그래밍 언어(OOPL)라는 용어는 사람에 따라 다른 의미로 해석하며 객체지향성의 정도에 따라 변화성이 있다. Wegner[11]는 다음과 같이 객체 지향성을 분류하고 있다.

- 객체기반(object-based) 언어 : 객체를 지원하는 모든 언어 집합
- 클래스기반(class-based) 언어 : 한 클래스에 속하는 모든 객체를 요구하는 부분집합
- 객체지향(object-oriented) 언어 : 상속성을 지원하는 클래스를 요구하는 부분집합

객체 기반 언어는 객체의 기능성을 지원하지만 관리를 지원하지는 않는다. 클래스 기반 언어는 객체관리를 지원하나 클래스의 관리를 지원하지는 않는다. 객체 지향 언어는 객체 기능성, 클래스에 의한 객체 관리, 그리고 상속성에 의한 클래스 관리를 지원한다. 이를 공식화 해보면,

- 객체기반=캡슐화+객체 주체성
- 클래스기반=객체기반+집합 추상화
- 객체기반=클래스기반+상속성+자신재귀성

이라 할 수 있다.

OOPL은 순수 OOPL과 혼합형 OOPL로 나눌 수 있다. 순수 OOPL은 패러다임을 지원할 뿐만 아니라 강요하는 것이며, 혼합형 OOPL은 패러다임을 강요하지는 않는다. 〈표 1〉은 현재 사용중인 주요 3세대 언어의 분류 표이다.

〈표 1〉 3세대 언어의 분류

구조화	객체기반	클래스기반	순수객체 지향	혼합객체 지향
C	Ada	CLU	Smalltalk	C++
Pascal			Eiffel	Objective-C
COBOL			Simula	CLOS
			Trellis	Object Pascal
			Actor	Object COBOL

3.2 언어의 종류 및 특징

3.2.1 Simula

Simula67은 범용 프로그래밍에 클래스와 상속성의 개념을 처음으로 도입한 언어이다[1]. Simula67의 뿌리가 ALGOL에 있지만 주로 시뮬레이션 언어로 고안되었다. 즉 Simula 객체는 독립된 존재이며 시뮬레이션을 하는 동안 다른 객체들과 교신한다. 다중 상속성은 제공하지 않는다. 정보 은폐는 아래 방향으로 상속되는 것을 방지하는 메카니즘에 의해 달성된다. 다형식은 오버로딩의 형태로 지원된다. 형 대조는 컴파일 시간에 통계적으로 수행되거나, 속성이 "virtual"로 선언되면 런 시간에 수행될 수 있다. 운영 중에 시스템은 시간마다 쓰레기 수집을 실행하게 된다.

3.2.2 Smalltalk

Smalltalk는 A. Kay, A. Goldberg, D. Ingalls 등의 연구자의 결과로 Xerox PARC 에서 개발되었다. 사실 Smalltalk는 단순한 언어가 아니라 편집기, 클래스 라이브러리 브라우저(browser) 및 많은 4GL의 특성을 가진 완전한 프로그래밍 환경이다.

Smalltalk에 의해 도입된 중요한 개념 중의 하나로 메타 클래스(metaclass)가 있다. 메타 클래스란 구체예로 될 수 없는 클래스의 클래스이며 훨씬 구체적인 개념으로 상속될 수 있는 추상개념을 표현한다. 메타 클래스는 클래스 구축을 위한 형판으로 정의한다. Smalltalk에서 각 클래스는 메타 클래스의 오직 하나의 구체예이며 메타 클래스는 클래스가 존재할 때 생성된다. 각 메타 클래스는 클래스 Metaclass의 구체예이며 클래스 Class의 부속 클래스들이다. 클래스 메소드는 실제로 메타 클래스에 저장된다. 이러한 개념은 이 언어를 초보자가 이해하기 어렵게 만드는 점이다. 이 언어는 아무런 내장(built-in)형이 없는데 이것은 LISP유의 언어로부터 영향을 받았다고

볼 수 있다. 또한, 내장 제어구조를 가지지 않지만 이러한 구조는 메시지 전달로부터 수행된다. 예를 들면 대괄호로 묶은 수식 – 블록이라 부름 – 은 수식의 평가를 연기한다. 대부분의 제어 구조는 블록을 인수로 삼는 객체에 대한 메시지로 구현된다.

Smalltalk-80으로 판매되던 시스템은 현재 ObjectWorks\Smalltalk로 불리며, 여기에는 대략 7400 메소드를 가진 350 클래스를 갖추고 있다. X 윈도하의 UNIX 워크스테이션과 Windows의 PC에서도 사용가능하다. Smalltalk-V는 대략 2000개의 메소드를 가진 100클래스를 갖추고 있다[12].

ObjectWorks\Smalltalk와 Smalltalk-V는 전혀 다른 클래스 라이브러리를 가지며 사용자 인터페이스를 사용하는 코드는 서로 이식성이 없다.

3.2.3 C++

C++는 C언어의 슈퍼 집합으로 1980년 중반 AT&T 벨 연구소의 B. Stroustrup이 개발한 언어로서, 추상화, 상속성, 자가 참조, 정적 및 동적 바인딩을 제공한다. Smalltalk와 달리 형체계는 단일 루트 트리 구조로서 구성되는 것이 아니라 상대적으로 작은 트리들의 집단으로 구성된다. 그러므로 C++는 Smalltalk에 의해 강요되는 협소하고 깊은 구조 – 모든 것이 Object의 부속 클래스 – 와 비교해서 광범위하고 얕은 클래스 구조를 권고한다.

C++에서 메소드는 멤버 함수(member function)라 부르며, 메시지 전달은 함수 호출에 대응한다. C++의 기발한 특성 중의 하나는 멤버가 아닌 다수 클래스들의 전용부(private part)에 대해 함수 접근 권한을 부여하는 능력이다. 이것은 함수를 이들 클래스의 "friend"로 선언함으로써 달성된다.

상속성은 부속 클래스의 정의를 통해 구현된다. C++에서 부속 클래스는 유도된 클래스(derived class)라 한다. 유도된 클래스들은 공용(public)으로 정의된 클래스에 따라 기본 클래스(base class)의 모든 성질을 상속받는다. 버전 2.0 이후부터 다중 상속 및 예외 처리 메커니즘이 가능하게 되었다[13]. 동적 바인딩은 기본 함수에서 정의되나 유도된 클래스에서 재정의 될 수 있는 가상 함수(virtual function)에 의해 구현된다.

3.2.4 Eiffel

Eiffel은 완전성, 강건성, 이식성 및 효율성의 논쟁을 의식적으로 불러일으킨 언어이다. Smalltalk와 달리 Eiffel에서 클래스는 객체가 아니다. 이것은 정적 형 체계가 운행 시간 오류를 감소시키고 효율성을 증가시키도록 한다. 다른 언어들에서 메소드라는 용어는 루틴(routine)이라 부르며 클래스는 루틴과 속성 모두를 캡슐화하는데, 양자를 "feature"로 모아 참조한다. Feature는 C++나 Ada에서의 전용(private) 또는 공용(public, export)이 될 수 있다.

Eiffel의 중요한 특성은 객체의 연산은 "assertion"을 작성하고 그에 복종해야 한다는 정형적 성질을 지정할 수 있는 능력이다. Assertion은 사전 조건, 사후 조건 또는 불변 조건을 표현할 수 있다. 사전 조건은 메소드가 호출될 때마다 대조되며, 사후 조건은 메소드가 종료하거나 값을 반환할 때 사실이도록 보장된다. 불변 조건은 일단 객체가 생성되거나 메소드가 호출되면 항상 사실이다.

Ada에서처럼 형을 표현하기 위해 매개변수를 가진 포괄 클래스 개념이 있는데, Ada와는 달리 상속성이 전폭적으로 지원된다. 다중 상속성이 지원되나 두 부모로부터 상속된 속성 또는 메소드가 후계에서 재명명(rename)되어야 하기 때문에 상속 충돌은 다루어지지 않는다.

응용부문은 효율성과 이식성을 위해 C로 컴파일 되며 Ada처럼 프로그램 기술언어(PDL) 또는 설계 언어로 사용될 수 있다.

3.2.5 Objective-C

Objective-C[6,14]는 ANSI C의 슈퍼집합과 Smalltalk-80의 주요 부분을 갖춘 혼합형 OOPL

로서 new 자료형, 객체 식별자, new 연산, 메시지 전달을 도입하였다. Objective-C는 C 컴파일러에 적합한 형태로 문장을 전환하는 전처리기 역할을 한다. 저장장소는 "factory" 객체에 의해 관리되는데 이 객체는 클래스를 표현하며 컴파일 시간에 생성된다. Factory 객체는 모두가 운행 시간에 클래스의 구체예를 생성하는 "new"라는 메소드를 가진다.

3.2.6 CLOS

CLOS(Common Lisp Object System)는 LOOPS라는 초기 언어에 기반을 두고 위원회에 의해 만들어졌으며 LISP유의 구문을 가진 혼합형 OOPL이다[15,16]. CLOS는 Smalltalk처럼 점진적인 양식을 제공하며, 빠른 시제품화를 수행하는데 유용하도록 약한 형 체계를 가진 인터프리팅 언어와 유사하다.

AI 시스템에서 객체에 가장 근접하는 개념은 프레임(frame)으로 객체의 개념을 포착하기 위한 구조적 추상화이다. 프레임은 속성 또는 클래스 변수와 동등한 슬롯(slot)의 확장 가능한 목록으로 이루어진다. 슬롯은 상태와 프로세스 기술 모두를 포함하고 메소드는 특정 슬롯에 부착된다. 상태를 갖고 있는 슬롯은 탐색 암묵적인 값, 자극 등을 결정하기 위한 메소드인 facet을 가질 수 있다. CLOS에서 각 객체는 유일한 식별자를 가지며, 슬롯은 전체 객체의 상태를 표현하는 다른 객체에 대한 포인터를 갖는다. 모든 구체예 객체는 한 클래스에 속하고, 모든 클래스는 하나의 형이다. 슬롯은 제한된 형 또는 자유형이 될 수 있다. 즉, 슬롯이 지정된 형의 값을 택하도록 하거나 형에 관계없이 무엇이든지 모든 값을 택할 수 있게 한다. 포괄 함수는 여러 형의 객체에 대해 작업을 하며, 메소드는 특정 인수로 호출될 때 포괄 함수가 할 것을 결정하도록 지정된다. 메소드는 다수의 형이 있는데 accessor 메소드는 슬롯의 값 facet에 접근하거나, 수정할 수 있다. 다른 OOPL과의 차이점은 값보다는 facet의 부분(sub-slot)이 있다는 점이다. 속성 또는 구체예의 슬롯은 값을 가질 수 있다.

3.2.7 기타 다른 언어

(1) Trellis

Digital Equip. 사에서 개발한 언어로 C++처럼 기본 언어를 확장한 것이 아니라, 새로 설계한 언어이다[9]. Pascal유의 구문을 가진 강한 형 체계 언어이지만 Smalltalk-80의 영향을 받았다. 언어는 Object라고 부르는 루트 클래스로부터 상속받는 모든 클래스(Trellis 용어로 type)를 가지고 다중 상속을 지원한다. Eiffel과 같은 강한 형 체계 언어와 공통으로, 호출 목표가 컴파일 시간에 해결되어 메시지 전달 메커니즘이 직접 프로시저 호출에 의해 대치되지만 부속형 체계(상속) 메커니즘을 통해 동적 바인딩을 지원한다. 또한, 매개변수화된 형과 함수 오버로딩(Trellis 용어로 type union) 역시 지원한다. Trellis는 윈도 기반 도구, 그래픽스, 문장 I/O등에 대한 형들을 갖고 있는 다수의 클래스 라이브러리가 있으며, C나 Fortran등의 다른 언어와 연동할 수 있다.

(2) Actor

Actor[17]는 Pascal에 기반을 두고 윈도즈 응용 부문을 개발하기 위해 특별히 설계된 순수 OOPL로서, 윈도즈 객체를 다루는 메뉴, 아이콘, 그래픽스 등 다양한 함수를 수행하는 클래스를 갖는 라이브러리를 제공한다. Actor에서는 모든 것이 객체이다. 클래스 자신도 변수와 메소드를 가진 객체가 된다. 단일 상속성을 지원하며 쓰레기 수집기를 운행 시간에 불러들인다. 정적 및 동적 바인딩을 지원하나 정적형 대조가 항상 채택된다.

(3) Object Pascal

Object Pascal[18]은 N.Wirth와 Apple 엔지니어 그룹에 의해 설계되었다. 추상 데이타 형, 메소드 및 상속성의 개념을 지원하기 위해 Pascal 언어를 확장하였다. 클래스 정의의 개념을 지원하기 위해 RECORD 구조를 확장하며, 데이타와 메소드 필드를 모두 갖출 수 있다. 메소드는 클래스 이름에 한정되어 프로시저 또는 함수로 정의된

다. 메시지는 필드 한정을 위한 Pascal 구조를 사용하여 전송된다.

(4) Object COBOL

COBOL 커퓨니티는 메인프레임에 뿌리를 두고 있으므로 새로운 프로그래밍 원리를 채택하는데 매우 더디다. 그렇지만, 객체 지향화가 개발자에게 코드를 쉽게 재사용 할 수 있게 하여 유지보수 노력을 경감시킨다는데 영향을 받은 COBOL 컴파일러 판매회사들은 COBOL을 개량하기 시작했다.

1989년 Codasyl COBOL 위원회가 후원한 심포지엄에서 COBOL에 대한 객체지향 방법론과의 관련성을 조사하는 회의를 개최하여, 객체지향 COBOL 작업 그룹을 결성하였다. 이 그룹의 목표는 다음과 같다[20].

- 기존 COBOL 언어 구문과의 양립성
- COBOL 구문과 의미론과의 일관성
- 현 COBOL 구문과 의미론과의 일관성
- 운행 시간 작업에 대한 최소한의 변경
- 과거 코드의 캡슐화
- 다른 환경에서 다양하게 사용되기 위해 채택되는 객체의 전환성

4. 객체지향 프로그래밍 언어의 논쟁점

4.1 설계 논쟁점

객체 지향 특성은 정적인 능력보다는 동적인 능력을 그 핵심에서 표현하므로, OOPL 설계에서의 고려사항으로 융통성에 대한 운행 시간의 불리한 조건을 감소시키기 위한 특성들이 도입되도록 한다. Smalltalk와 같은 동적 지향언어에서는 이러한 점이 덜 중요하지만, C++, Eiffel, Simula 같은 언어에서는 운행 시간의 불리한 조건이 전체 언어 설계의 중요한 국면이 된다[19].

4.1.1 클래스와 형

데이타 형이 있는 클래스를 어떤 언어에 도입한다는 것은 클래스가 몇 가지 방법으로 형 체계에 통합되어야 함을 의미한다. 그 유형은 다음과 같다.

(1) 클래스가 특별히 형 대조에서 제외될 수 있다. 이때 객체는 형없는(typeless) 객체가 되며, 클래스 멤버십은 형 체계와 별개로 유지된다(일반적으로 운행 시간에 추가 기법을 사용함). 클래스를 기존 언어에 추가하는 이러한 방법은 가장 간단하며, Objective-C와 같은 몇 몇 언어에서 사용된다.

(2) 클래스는 형 구축자가 될 수 있다. 즉 클래스 선언을 언어 형 체계의 일부로 만들 수 있다. 이것은 C++나 객체 지향 설비를 기존의 언어에 덧붙인 다수의 언어에 의해 채택된 해법이다.

(3) 클래스는 단순히 형 체계가 될 수 있다. 즉, 모든 다른 구조화된 형은 언어에서 배제된다. 예를 들면 배열, 문자열들은 클래스로서 선언되어야 한다. 이것은 Smalltalk나 Eiffel과 같은 언어에서 시작되어 다수 언어에서 채택된 접근법이다.

4.1.2 상속성 대 다형식

OOPL이 아닌 언어는 프로그램이 코드를 재사용 하는 제한된 방법으로 다형식의 형태를 제공한다. 예를 들면 Ada는 만일 번역기가 매개변수의 형 대조를 통하여 연산의 다른 형식을 구별할 수 있다면, 함수 또는 프로시저의 이름에 대한 재사용(overloading)을 허용한다 또한 스택의 비지정 원소와 같은 형 매개변수(매개변수화)를 사용하여 포괄형의 정의를 허용한다.

반면, OOPL은 명시적 다형식 설비가 덜 필요하다. 왜냐하면 상속성과 동적 바인딩이 묵시적 다형식에 대한 기법을 제공하기 때문이다. 이것은 오버로딩에 대해서도 맞는 말로서 객체의 메소드를 호출한다 함은 자동적으로 현 객체의 클래스에 기반을 둔 메소드를 선택하게 되기 때문이다.

4.2 구현 논쟁점 [19]

4.2.1 객체와 메소드의 구현

전형적으로 객체는 C나 Pascal에서 레코드 구조가 레코드의 자료필드를 표현하는 구체예 변수를 가지는 것과 똑같이 구현된다. 유도된 클래스의 객체는 레코드의 끝에 할당된 새로운 구체예 변수를 가진 채 앞선 자료 객체에 대한 확장 형태로 할당될 수 있다. 메소드는 보통의 함수로 구현될 수 있는데, 클래스의 현재의 객체 데이타를 직접 접근할 수 있다는 점이 함수와 다르다. 이 기법을 구현하기 위하여 메소드를 묵시적 특별 매개 변수를 가진 함수로 간주하는데, 특별 매개변수는 각 호출 시 현재의 구체예에 설정되는 포인터가 된다.

4.2.2 상속성과 동적 바인딩

위에서 언급한 구현에서 구체예에 대한 공간만이 각 객체에 할당되며, 메소드의 할당은 제공되지 않았다. 이것은 메소드가 완전히 함수와 동등한 동안 작업을 하고 각각의 위치는 명령형 언어의 평범한 함수처럼 컴파일 시간에 알려지기 때문이다.

한편, 이러한 구현은 동적 바인딩이 메소드에 대해 사용될 때 문제가 발생된다. 왜냐하면 호출을 위해 사용되는 정확한 메소드는 실행 시를 제외하고는 알려지지 않기 때문이다. 이에 대해 가능한 해결책은 각 객체에 대해 할당된 레코드 구조에서 특별 필드로서 모든 가상(virtual) 메소드를 유지하는 것이다. 이것의 문제는 각 객체 구조가 그 순간에 모든 가상 함수의 목록을 사용할 수 있도록 유지해야만 하는데, 이 목록은 매우 커질 수 있어 상당한 공간을 소비하게 된다는 점이다. C++의미로 메소드가 "virtual"이다. 즉 자손 클래스에서 재정의를 가진다.

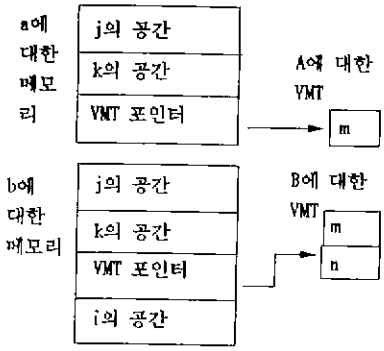
Eiffel에서는 가상 정의가 없고 모든 메소드는 자손 클래스에 의해 재정의 되는 능력을 가진다.

이 전략의 대안은 전역 기억영역과 같은 중앙 위치에 각 클래스에 대한 가상 메소드의 테이블을 유지하는 것이다. 이 테이블은 코드 포인터를 갖고 정적으로 적재되어 있으며, 각 객체 구조에서 저장된 이 테이블에 대한 단일 포인터를 가지고 있다. 이러한 테이블을 VMT(Virtual Method Table)이라 한다. (그림 7)에는 클래스 정의와 메모리 할당의 예를 보인다.

```
class A;
real j, k;
procedure l;
virtual procedure m;
.....
end A;

A class B;
real i;
procedure l;
virtual procedure n;
.....
end B;
```

(a) 클래스 정의(Simula구문)



(b) 메모리 할당

(그림 7) 클래스 정의와 VMT

이제 각 가상 메소드에 대한 호출은 VMT 포인터를 통한 별도의 간접지시에 의해 완성된다. 이들 구현의 어느 쪽도 실행시 상속성 그래프를 유지하도록 요구하지 않는다.

4.3 프로그래밍 논쟁점 [12]

4.3.1 개발 과정상에서 상속성의 효과

다중 상속성은 적은 코드 중복성의 이점이 있는 반면 다루기 어렵고 이름 충돌과 같은 문제를 해결하기 위한 좋은 개발 환경 지원이 요구된다.

대부분의 OOPL에서 클래스 계층구조에서의 변경은 모든 클래스의 재컴파일을 발생시킨다. 또 한 모든 클래스에 대한 모든 클라이언트도 재컴파일 되어야 한다. 이것은 두 클래스들 간의 관계성에서 변경 효과가 전체 클래스 계층구조를 걸쳐서 전파되기 때문에 캡슐화의 원리를 위반하게 된다. C++나 Eiffel에서의 경험은 계층구조에서 변경 다음의 시스템 재컴파일은 사용자에게 심각한 문제일 수 있음을 보여 주고 있다. 한가지 가능한 해결책은 링크 시간 바인딩을 수반하는 것이다. 이것은 전체를 재컴파일 하는 대신에 클래스 구조를 다시 연결하는 것으로 충분하다.

4.3.2 디버깅과 객체 지향 구문

(1) 자료 추상화의 효과

명확히 정의된 인터페이스와 메시지 전달의 사용을 통해 객체의 데이타 및 메소드 구현을 캡슐화 하는 것은 객체간의 숨겨진 종속성이 형성되는 것을 방지한다. 모든 종속성이 분명하므로 버그(bug)가 쉽게 분리된다.

데이타 추상화는 부주의한 버그의 수정을 경감하는데 도움이 된다. 이는 객체의 데이타와 메소드로 만들어진 변경을 국지화(localization)하기 때문이다. 이러한 객체마다에 대한 국지화는 점진적 개발과 시제품화(prototyping)시에 버그의 개입을 막아준다.

(2) 상속성의 효과

상속성의 주요 결점은 어느 정도 자료 추상화를 위반한다는 점에 있다. 상속성은 부속 클래스가 슈퍼 클래스에서 정의된 데이타 표현과 메소드를 직접적으로 사용할 수 있게 한다. 결과적으로 버그는 클래스 계층구조를 통해 전파되고 다소 국지화가 손실된다.

또, 상속성은 프로그램 수정의 용이함을 감소시킨다. 복잡한 종속성이 부속 클래스/슈퍼 클래스 관계성에서 생성될 수 있다. 다중 상속성은 메소드 운행 시간 충돌의 가능성을 일으킨다. 들어오는 메시지가 다른 클래스들로부터 상속된 2개 이상의 메소드에 부합할 수 있다. 이러한 충돌은 C++에서 처럼 클래스 계층구조의 선형화에 의해 자동적으로 해결되거나 수동적 임의성에 의해 해결될 수 있다.

5. 결 론

본 논문에서는 OOPL의 특성과 종류를 객체지향 기본 용어의 정의에 입각하여 논의하였다. 또한 OOPL에 관한 논쟁점에 대해 설계, 구현, 프로그래밍 측면에서 고찰하였다. 이들 OOPL을 선택할 때 고려할 사항들은 다음과 같다[12].

- 객체지향지원의 정도 : 모든 언어가 같은 정도를 객체지향 원리를 지원하는 것은 아니다.
- 관련된 개발환경 : 브라우저와 디버깅 설비가 있는가? 다중 사용자 개발환경인가?
- 상속성의 형태 : 단일 또는 다중 상속인가? 상속성의 관리는 어떻게 하며, 이름 충돌의 해법을 위한 설비는 있는가?
- 최종 응용부문의 성격 : 최종 이진(binary) 이미지는 얼마나 빠르게 실행되는가? 이진 이미지는 얼마나 큰가?
- 형 체계 : 정적 형 체계인가 동적 형 체계인가?
- 다른 언어와의 상호 연동 능력 : 기존 소프트웨어에서 새로운 언어와 상호연동을 통해 재사용 하는 것이 필연적일 때 중요하다.
- 언어에 대한 표준
- 언어의 성숙도 : 상업적 측면의 소프트웨어가 개발될 예정이라면 언어는 성숙한 제품의 강건한 특성을 제공해야 한다.
- 판매자 및 제3자 제품에 의한 언어 지원

· 상업적 요소 : 상업적 성공이 반드시 기술적 우월성을 나타내지는 않지만 장기간 표준이 될 가능성이 있는 제품임을 보여준다.

끝으로 앞에서 기술한 OOPL들과 그들의 특성 및 선택기준에 관한 사항들을 비교하여 〈표 2〉에 보인다.

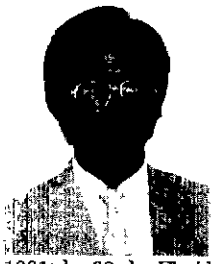
〈표 2〉 OOPL의 주요 특성 비교

특 성	Simula	Smalltalk	C++
형대조/바인딩	초기/후기	후기	초기/후기
다형식	○	○	○
정보은폐	○	○	○
병행성	○	빈약	빈약
상속성	단일	단일	다중
쓰레기수집	○	○	×
지속성	×	×	×
포괄성	×	×	○(형판)
객체라이브러리	시뮬레이션	대다수그래픽	소규모
타언어와의 연동	○	곤란	○
표준	표준그룹	IEEE (P1152)	ANSI (X3J16)
특 성	Eiffel	Objective-C	CLOS
형대조/바인딩	초기	초기/후기	후기
다형식	○	○	○
정보은폐	○	○	○
병행성	기대	빈약	곤란
상속성	다중	단일	다중
쓰레기수집	○	○	○
지속성	다소지원	×	×
포괄성	○	×	○
객체라이브러리	소규모	소규모	소규모
타언어와의 연동	○	○	곤란
표준	NICE	×	ANSI (X3J13)

참 고 문 헌

1. J. Dahl and K. Nygaard, "Simula-an ALGOL-based Simulation Language", Comm. of the ACM, Vol. 9, pp. 671-678, 1966.
2. A. Goldberg and D. Robson, Smalltalk-80: The Language, Addison-Wesley, 1983.
3. A. Goldberg, Smalltalk-80 : The Interactive Programming Environment, Addison-Wesley, 1984.
4. G. D. Buzzard and T. N. Mudge, "Object-Based Computing and the Ada Programming Language," IEEE Computer, Vol. 18, No. 3, pp. 12-19, March 1985.
5. S. Cook, "Language and Object-Oriented Programming," Software Engineering Journal, Vol. 1, No. 2, pp. 73-80, 1986.
6. B. J. Cox, Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley, 1986.
7. D. A. Moon, "Object-Oriented Programming with Flavors," OOPSLA Conference Proceedings, pp. 1-8, 1986.
8. B. Meyer, Object-Oriented Software Construction, Prentice-Hall, 1988.
9. C. Schaffert and et al, "An Introduction to Trellis/Owl", Proc. OOPSLA '86, pp. 9-16, 1986.
10. B. Stroustroup, The C++ Programming Language, Addison-Wesley, 1986.
11. P. Wegner, "Dimensions of Object-based Language Design", SIGPLAN Notices, Vol. 22, No. 12, No. 12, pp. 168-182, 1987.
12. G. Wilkie, Object-Oriented Software Engineering: The Professional Developer's Guide, Addison-Wesley, 1993.
13. A. Koenig and B. Stroustrup, "Exception Handling for C++", J. of Object-Oriented Programming, Vol. 3, No. 2, pp. 16-33, 1990.
14. B. J. Cox and A. Novobilsk, Objected-Oriented Programming-An Evolutionary Approach, 2nd ed. Addison-Wesley, 1991.
15. D. G. Bobrow, and et al, The Common LISP Object System Specification. ANSI TR 88-002R, 1988.
16. D. Moon, "The Common LISP Object-Oriented

Programming Language Standard", In Object-Oriented Concepts, Databases and Applications (Kim W. and Lochovsky F. H., eds.) Addison-Wesley, 1989.
17. Whitewater Group, Actor Language Manual, The Whitewater Group, Inc., 1989.
18. K. J. Schmucker, Object-Oriented Programming for the Macintosh, Hayden Book Co., 1986.
19. K. C. Louden, Programming Languages: Principles and Practice, PWS-KENT Pub. Comp. 1993.
20. I. Graham, Object-Oriented Methods, 2nd ed. Addsion-Wesley, 1994.



하 수 철

1981년 홍익대학교 전자계산학과 졸업(학사)
1986년 홍익대 대학원 전자계산학과 졸업(석사)
1990년 홍익대 대학원 전자계산학과 (이학박사)
1981년~84년 Army Logistics Command, System Analyst
1991년~92년 Florida State Univ. 객원교수, Univ. of Texas 객원교수
1987년~현재 대전대학교 컴퓨터공학과 부교수
관심분야 : 소프트웨어 공학, 객체지향화(OOA/OOD/OOP), 프로그래밍 언어, 시각 프로그래밍, 멀티미디어