□특 집□

객체지향 소프트웨어 공학

한국통신 연구개발단 김 수 동*

● 목 차 ●

Ⅰ. 서 두
1.1 객체지향 프로그래밍 기술의 인식
1.2 객체지향 소프트웨어 공학
Ⅱ. 객체지향 프로그래밍 개념
2.1 객체지향적 소프트웨어 시각
2.2 객체(Object)
2.3 메시지(Message)
2.4 클래스(Class)
2.5 속성상속(Inheritance)
2.6 복수 속성상속(Multiple Inheritance)
2.7 추상클레스(Abstract Class)
2.8 다형성(Polymorphism)
Ⅲ. 객체지향 프로그래밍의 장·단점
3.1 장점
3.2 단점
Ⅳ. 객체지향 분석 및 설계
4.1 구조적 방법
4.2 객체지향적 방법
4.3 객체지향 분석 및 설계단계
4.4 대표적 객체지향 분석 및 설계방법들
Ⅴ. 객체지향 프로그래밍 언어
5.1 프로그래밍 언어의 세대
5.2 제1세대와 2세대 초반의 언어
5.3 제2세대 후반과 3세대 초반의 언어
5.4 제3세대 후반의 언어
5.5 객체지향 언어의 특징
5.6 객체지향 언어의 선택
Ⅵ. 객체지향 소프트웨어 공학의 기타 분야
6.1 객체지향 기술의 도입
6.2 첫 과제의 선정
6.3 교육
6.4 과제위험도 측정 및 관리
6.5 객체지향 재사용
6.6 프로젝트 스텝핑
6.7 기 타
Ⅶ. 결 어

I. 서 두

1.1 객체지향 프로그래밍 기술의 인식

객체지향 프로그래밍! 이 단어가 오늘날의 프로그래머나 소프트웨어 엔지니어에게 주는 느낌은 어떤 것일까? 아마도 대부분의 이들에겐 "신기술", "고급기술", "전혀 새로운 방식의 프로그래밍 기술", "미래의 소프트웨어 개발방법", "세계 유수 소프트웨어 회사들이 주력하는 기술", 등의 인식이 보편적인 것 같다. 그래서 많은 관심이 기울고 논문들이 쏟아져 나오며 전산분야에 종사하는 사람이면 누구나 한번쯤 배우고 적용해 보고 싶어하는 기술이다. 국제적 흐름에 민감한 이들은 "미래의 절대적인 소프트웨어 개발기술"로 인식하고 일부 보수적인 이들에게는 "잠시 바람을 일으키고 곧 소멸될 한 유행"이 아닌가 하는 신중한 시각도 있다.

제각기 다른 시각과 인식속에서도 Object-Oriented Programming, OOP로 알려진 이 기술은 이미 전산 선진국의 소프트웨어 산업계에 상당히 깊은 뿌리를 내린 기술로 미래의 주된 소프트웨어 개발방법임을

* 정회원

대부분의 사람들이 인정하고 있는 것 같다. 우리에게 널리 알려진 미국의 마이크로 소프트사의 회장, 빌 게이트씨는 최근에 "객체지향은 90년대의 여러 소프트웨어 기술 중 가장 중요한 기술이 될 것이다"라고 공언하고 이 분야에 많은 투자를 하고 있다. 통신분야의 국제표준 기구인 CCITT(국제전신전화 자문위원회)에서도 모든 통신 관련 시스템에서 정보관리 및 처리를 "관리객체(Managed Object)"라는 개념과 정의를 통하여 객체지향 방식으로 수행하도록 지침을 내놓았다. CCITT X.720 시리즈의 지침서들이 모두 이 객체지향식 관리객체에 관하여 기술하고 있으며 이들 지침과 표준안은 전 세계적으로 넓게 인정을 받고 있다.

소프트웨어 기술이 하드웨어 발전에 비해 매우 비흡한 수준으로 발전을 해온 것과 소프트웨어 개발의 생산성 부진 문제를 아직도 근원적으로 해결하지 못한 문제를 가르켜 소프트웨어 위기 혹은 "25년 묵은 소프트웨어 생산성 문제"라 한다. 즉, 아직도 소프트웨어 분야는 재사용할 수 있는 소프트웨어 부품들로 결합 생산하여 생산성을 높이면서 기능 수정 및 보완을 용이하게 하는 성숙된 기술과 적절한 방법이 없는 것이다. 객체지향 기술은 이 소프트웨어 위기문제를 해결하기 위한 매우 효과적인 방법으로 전세계적으로 각광을 받고 있다.

미국에서는 92년 7월에 열린 Object World '92라는 학술회에서 객체지향 프로그래밍 기술로 인해 미국의 소프트웨어 산업에 혁명이 일고 있다고 공감을 나눴다. 즉, 재래식의 소프트웨어 개발방법과 프로그래밍 언어들로부터 객체지향의 새로운 방법으로의 전환이 이미 많이 이루어져서 소프트웨어 위기를 극복하는데 효과적이었다는 보고이다. 일본 NTT사의 최근 한 논문에 의하면 일본은 소프트웨어 재사용을 통하여 생산성 문제를 해결하려고 한다[10]. 즉, 하나의 소프트웨어를 이미 만들어진 소프트웨어 모듈들을 조립하여 개발기간과 비용을 줄이는 반면 품질은 향상시키려는 전략이다. 이를 위해서 일본에서는 객체지향 기술은 사용해야만 효율적인 소프트웨어 재사용이 가능하다고 믿고 있고 많은 투자를 하고 있다.

이처럼, 소프트웨어 위기 해결을 위한 여러 전산 선진국들의 공통적 전략은 객체지향 기술의 활용에 있다. 주된 이유로는 객체지향 기술이 하드웨어 IC칩처럼 재사용이 편리한 독립적인 소프트웨어 IC의 개발 및 활용을 가능케 하기 때문이다. 또한, 사람들에게 익숙한 방법 즉 우리의 사고방식을 그대로 반영한 방식으로 주어진 문제를 분석 설계 및 구현하게 해 주기 때문인데 이를 자연적 모델링(Natural Modeling)이라 부른다[4].

1.2 객체지향 소프트웨어 공학

객체지향 프로그래밍 기술은 원래 일종의 프로그래밍 방식을 의미하는 Object-Oriented Programming에서 출발하였다. 그후, 좋은 프로그램은 좋은 설계 나아가 좋은 분석에 의해서만 가능해진다는 인식으로 객체지향 분석 및 설계방법들이 나오기 시작했으며, 최근에는 소프트웨어 공학의 여러 요소에 객체지향의 개념을 반영한 "객체지향 소프트웨어 공학"으로 발전하고 있다.

소프트웨어 공학이란 학문을 그 목적이기도 한 "우수한 품질의 소프트웨어 시스템을 최소의 비용으로 개발"하려는 일련의 이론, 방법 및 툴들에 관한 학문으로 정의할 수 있다. 그렇다면 객체지향 소프트웨어 공학에 대한 정의도 자연스럽게 내릴 수 있다. 즉, 객체지향 소프트웨어 시스템을 보다 품질이 높게 개발함에 있어 비용을 최소화하는 일련의 방법론과 툴들에 관한 학문이다. 따라서, 본 고의 목적은 "객체지향 소프트웨어 공학"이란 제목으로 소프트웨어 개발에 실제 필요한 개념, 방법과 기술들을 알아 보는데 있다.

본 고의 구성은 II장에서는 객체지향 기술의 가장 기본이 되는 핵심 개념을 살펴보고, III장에서는 이 개념을 바탕으로 객체지향의 장·단점을 논해 본다. IV장에서는 객체지향 소프트웨어 공학의 중요한 요소인 객체지향 분석 및 설계 기술을 알아보며 V장에서는 구현에 필요한 객체지향 언어에 관하여 고찰해 본다. VI장에서는 객체지향 소프트웨어 공학의 기타 요소들에 대하여 간략히 알아본다.

II. 객체지향 프로그래밍 개념

2.1 객체지향적 소프트웨어 시각

객체지향 기술에서 가장 근본이 되는 개념은 주어진 문제를 이해하고 관찰하는 시각에 있다. 즉, 주어진 문제 영역을 그 안에 존재하며 관련된 여러 개의 객체들이 서로 정보를 주고 받는다고 보는 시각이다. 이는 우리가 프로그래밍을 떠나, 일상 생활에서 어떤 문제에 대해 생각하고 이해하며 관찰하는 방법과 동

일하다. 대학교에서 인사관리 자동화문제를 예로 삼아보자. 우선, 보통사람들의 일차적인 고찰은 인사관리의 대상이 되는 사람들과 교수, 교직원, 학부학생, 대학원생 등의 분류일 것이다. 이런 단계는 객체지향식 문제 분석과 프로그램 설계의 첫 단계이다. 즉, 인사관리 자동화 문제 영역안에 있는 대상이나 객체들이 무엇인가를 알아내고 그들의 역할(Behavior)과 상호관계(Association, Inheritance)를 규명하는 일인데 이는 우리의 일상 사고방식이며 객체지향 프로그램 개발에 가장 근본이 되는 정보들이다. 이를 표현한 좋은 말로서 "실세계 중심의 개발방법(Real-World-Oriented Paradigm)"이라고 한다.

소프트웨어에 관한 시각을 비교해 보자. 소프트웨어에 관한 재래식 시각은 "하나의 프로그램은 필요한 데이터 구조와 이 데이터 구조위에 수행되는 함수들의 집합"이다. 즉 다음과 같은 공식으로 표현할 수 있다.

데이터구조＋함수들＝프로그램

이러한 방식의 가장 큰 문제는 연관이 많은 데이터와 함수들도 별도의 독립된 것처럼 코드상에 정의 취급되는 것이다. 즉, 데이터 부분은 흔히 전역변수(Global Variable)들의 정의 부분에 포함되어 있고 이 데이터 부분과 연관된 함수들은 다른 함수들과 함께 정의되어 있어 이들간의 연관관계를 잘 표현하지 못하고 있다.

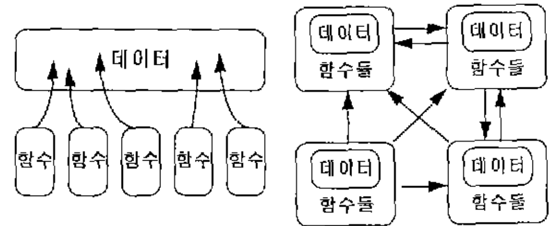
반면에 소프트웨어에 대한 객체지향식 시각은 하나의 객체지향 프로그램은 여러 개의 객체(Object)들로 구성이 되며 각각의 객체란 위에서 언급한 데이터 구조 부분과 관련된 함수들을 동시에 포함하고 있다는 것이다. 따라서 다음과 같은 공식이 생기게 된다.

데이터구조＋함수들＝객체 객체＋객체＝프로그램

(그림 1) 이 두 개의 다른 시각을 그림으로 표현한 것인데, 객체지향 기술이란 이렇게 주어진 문제에서 연관된 데이터 구조와 함수들을 포함하고 있는 객체들을 찾아내고, 그들간의 상호관계 및 특성 등을 규명하여 객체지향 프로그램이나 데이터베이스로 자연스럽게 옮겨주는 방법이다.

2.2 객체(Object)

객체란 필요한 데이터 구조와 그 위에서 수행되는



(그림 1) 소프트웨어에 관한 두 개의 다른 시각들.

함수들을 가진 하나의 소프트웨어 모듈이다. 각 객체가 데이터 구조를 가지고 있다는 것은 각 객체는 어떤 상태(State)를 가지고 있다는 것이다. 예를 들어서 그래픽 프로그램에서 하나의 점을 나타내는 객체에서는 그 점의 현재 스크린 상에서의 수평 및 수직위치가 바로 그 객체의 상태이다. 각 객체가 필요로 하는 함수를 가지고 있다는 것은 각 객체가 어떤 기능들을 수행할 수 있는 능력(Behavior)을 가지고 있다는 것이다. 따라서, 객체는 단순한 하나의 변수, 예로서

```
int X;
```

라고 정의된 "X"와는 다르다. "X"는 하나의 정수를 정의할 수 있는 장소를 가지고 있을 뿐, 이 "X" 자체가 어떤 기능을 수행할 수 있는 능력은 없다. 반면에 객체는 "X"와 같은 변수들을 가지고 있으면서 이들 변수들 위에 작동하는 기능들을 함께 가지고 있다. 예로서 스크린상의 점을 정의하는 C++ 코드를 보자.

```
class POINT
{
  int xPosition;
  int yPosition;
  ColorType color;
public:
  . . .
  void move (int x, int y);
  void setColor (ColorType z);
  . . .
}
```

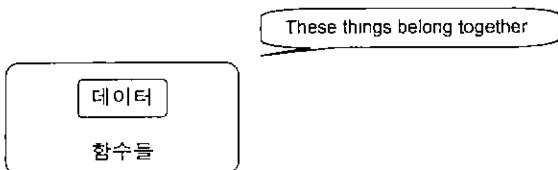
정수로 정의된 xPosition과 yPosition 및 color라는 세 개의 변수가 정의되어 각 점의 상태를 기록하고 있고, 이 객체의 능력은 점의 위치를 옮기는 move나 점의 색깔을 바꾸는 setColor 등이다. 이렇게 각 객

체는 단순한 변수만 가지는 것이 아니라 그 객체가 무엇을 할 수 있는가가 함께 정의되어 있다. 이를 "하나의 객체는 어떤 데이터 구조와 필요로 하는 함수들을 캡슐화 하여 가지고 있다"고 표현한다.

주어진 문제에서 어떤 것들이 객체가 될 수 있을까? 일반적으로 문제영역 안에서 어떤 사물이나 대상이 상태를 기록유지할 필요가 있고 필요로 하는 기능들이 잘 정의되어 있으면 이것은 좋은 객체가 된다. 하나의 객체는 주어진 문제영역 안의 어떤 사물이나 대상을 표현하고 있다. 보다 체계적으로 객체를 규명하는 방법들은 IV장에서 열거한다.

2.2.1 캡슐화(Encapsulation)

위에서 하나의 객체는 데이터부분과 함수부분을 캡슐화한다고 정의했다. 이 캡슐화는 무엇이며 왜 중요한 것인지 알아보자. 일반적으로 우리 생활에서 어떤 정보와 어떤 종류의 작업은 개념적으로 서로 연관되어 있음을 많이 본다. 이러한 연관된 정보와 작업 또는 기능들을 하나로 묶는 것은 자연스런 과정이다. 예를 들어 대학교의 인사관리에서는 학생들의 이름, 주소, 학번, 전공 등의 정보를 유지하며 학생들에 관해 가능한 작업인 평점계산, 주소변경, 과목신청 등을 기능들을 생각할 수 있다. 이러한 정보와 정보처리에 필요한 작업 즉 기능들은 모두 학생에 관한 것이므로 학생이라는 테두리로 묶어두는 것이 자연스런 것이다. 이렇게 연관된 사항들을 하나로 묶는 것은 캡슐화(Encapsulation)라 한다. (그림 2)는 연관된 데이터와 함수들을 캡슐화한 모습과 의미를 보여준다.



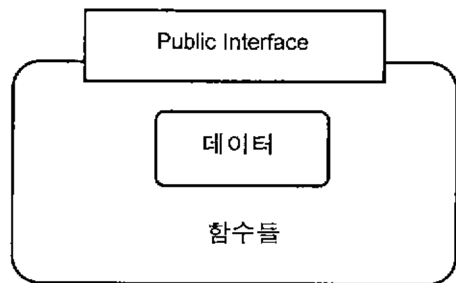
(그림 2) 캡슐화의 의미.

프로그램상에서 캡슐화의 의미는 프로그램 분석자나 설계자가 주어진 문제를 데이터와 함수 등 세부사항들은 개발의 차후단계에서 정의하고 객체라는 덩어리 단어로 문제에 대해 사고하게 하는 추상화의 수단을 제공하는데 있다.

2.2.2 정보은폐(Information Hiding)

캡슐화란 연관된 여러 항목들을 모아서 그 둘레에 캡슐을 씌우는 일이다. 그러나, 캡슐화 그 자체는 캡슐속에 있는 항목들의 정보가 외부에 은폐되었음을 보장하지는 않는다. 실제로 콘택600과 같은 감기약 캡슐들은 내부가 투명한 것도 있다. 정보은폐란 캡슐속에 쌓여진 항목에 대한 정보를 외부에 감추는 것을 의미한다. 즉, 처리하려는 데이터 구조와 함수에 사용된 알고리듬 등을 외부에서 직접 접근하지 못하도록 하고 캡슐안에 있는 함수들만이 접근하게 한다. 따라서, 캡슐화되고 정보은폐된 객체는 하나의 블랙박스가 되는 것이다. 객체지향에 관한 서적이나 논문에서 이 두 가지 개념이 중요시 소개되는 것은 바로 객체라는 것이 캡슐화와 정보은폐의 원리를 실제의 프로그래밍 언어에서 실현한 것이기 때문이다.

그러면 객체와 객체 사이의 정보교환은 어떻게 일어나는가? 각 객체는 외부에 공개하고자 하는 일련의 정보를 Public Interface로 정의해 외부의 객체들이 이 Interface를 통해서 그 객체와 정보를 교환토록 한다. 즉, 한 객체의 Public Interface는 그 객체가 "무슨일을 할 수 있다 혹은 이 정보는 공개할 수 있다"하고 외부에 선언하는 것이다. (그림 3)은 데이터 부분과 연관된 함수들을 캡슐화하고 정보를 은폐한 위에다 정의된 Public Interface의 개념을 보여준다.



(그림 3) 퍼블릭 인터페이스.

이 퍼블릭 인터페이스는 언어에 따라 표현방식이 다른데, C++에서는 "Public"이란 특별한 구문을 두어 "Public"란에 들어간 항목들만 외부에 공개된다. Eiffel이란 언어에서는 "export"라는 란에 지정된 항목들만 외부에 공개된다. 앞서 정의한 POINT라는 객체 정의를 보면 move와 setColor의 함수들이 외부에서 관찰될 수 있는 Public Interface임을 알 수 있다. 여기서 한 가지 유의할 것은 move와 setColor라는 함수들이 외부에 보여져 불리워질 수 있는 함수들이라는 것이며 각 함수가 가지고 있는 코드나

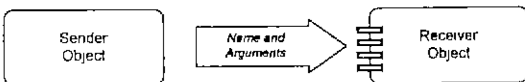
알고리듬까지 보여지는 것은 아니다. 한 함수가 외부에 보여지는 부분을 Signature라 하며 하나의 Signature는 함수의 이름, 입력 매개변수(Input Parameter)와 출력 매개변수(Output Parameter)로 구성되어 있다.

그러면, 이러한 캡슐화와 정보은폐의 이점은 무엇일까? 우선 객체 내부의 은폐된 데이터 구조가 변하더라도 주변 객체들에게 영향을 주지 않는다. 이는 객체의 은폐된 데이터 구조가 외부에 공개되지 않기 때문이다. 예로서 어떤 변수의 구조를 배열(Array)구조에서 리스트(List)구조로 바꾸더라도 프로그램의 다른 부분에 전혀 영향을 주지 않는다. 또한, 어떤 함수에 사용된 알고리듬을 바꾸더라도 Signature만 바꾸지 않으면 외부 객체들에게 영향을 주지 않는다. 예를 들어, Sorting 함수의 경우 처음 사용된 Sequential Sorting 알고리듬에서 Quick Sorting 알고리듬으로 바뀔 때 외부에 어떤 영향도 주지 않는다. 이러한 장점을 유지보수 용이성(Maintainability) 혹은 확장성(Extendability)이라 한다.

> 캡슐화와 정보은폐는 재사용이 가능하고 유지보수가 용이한 소프트웨어 모듈개발에 필수적인 원칙이며, 객체란 바로 이런 원칙들에 의해 생성되는 소프트웨어 모듈이다

2.3 메시지(Message)

일단 정의된 객체들이 서로 어떻게 접근하는가를 알아보자. (그림 4)에서 보는 바와 같이 객체지향 프로그램에서는 마치 데이터 통신에서 처럼 정보를 메시지(Message)라는 덩어리로 서로 주고 받으며, 이 메시지 교환이 객체들 사이에 정보교환을 위한 유일한 수단이다. 따라서, 하나의 객체지향 프로그램은 메시지 교환을 하며 통신하는 여러 개의 객체들로 이루어진다고 정의할 수 있다.



(그림 4) 메시지 교환.

하나의 메시지는 메시지를 받을 수신 객체(Receiver Object)가 누구인가를 명시하며, 그 수신 객체가 수행해야 할 함수의 이름을 지정하고 그 함수를 실행시 요구되는 인자들(Arguments)을 포함하고 있다. 실제로 메시지를 보내는 구문은 언어에 따라서 다소 차이가 있지만, 대체로 다음과 같은 것이 메시지의 일반적인 형태이다.

> 〈수신객체〉←〈함수이름〉〈필요한 인자들〉

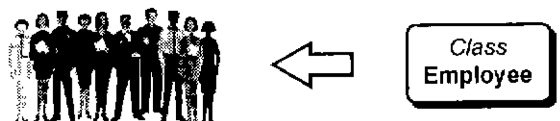
예를 들면, 한 "점"이라는 객체에 C++에서는 다음과 같이 메시지를 보낸다.

```
myPoint.move (10, 15);
```

여기서 "myPoint"는 한 점을 나타내는 객체이며 그 다음의 부호 "."는 "move"라는 메시지가 "myPoint"하는 객체에게로 전해진다는 메시지 전달의 표현이다. 괄호안의 두 숫자는 이 "move" 함수가 필요로 하는 인자들이다.

2.4 클래스(Class)

우리의 일상생활을 보면, 대부분의 사물에 종류나 유형이 있음을 알 수 있다. 예를 들면, 자동차가 무수히 많지만 승용차, 화물차, 버스 등으로 그 종류를 나눌 수 있고, 또 자동차를 모델별로 분류할 수도 있다. 이 때 각 사물이 어떤 유형에 속하는데 이러한 사고방식을 객체지향에서는 직접 표현할 수 있게 해준다. 즉, 각 객체에는 어떤 종류나 유형이 있는데 각각의 자동차는 별개의 객체이지만 여러 개의 유사한 자동차들이 하나의 자동차 유형에 속한다. 이러한 객체의 유형을 객체의 타입(Object Type) 혹은 객체 클래스(Class)라 한다. 그리고 한 클래스에 속하는 각각의 객체를 그 클래스의 인스턴스(Instance) 혹은 그냥 객체라 한다. 예를 들면 철수가 가지고 있는 파란색 엑셀 자동차는 엑셀이라는 클래스에 속하는 하나의 인스턴스이며, 현대자동차는 엑셀이라는 클래스에 해당되는 많은 엑셀 자동차들 즉, 인스턴스들을 만들어 내는 것이다. 또 다른 예로서, 회사내의 직원을 나타내는 "직원(Employee)"이라는 클래스를 생각할 수 있는데 그 중 한 직원인 영희는 직원클래스에 속하는 하나의 인스턴스가 된다. (그림 5)에서 보여주는대로 직원클래스는 회사내 모든 직원에 해당되는 공통된 특성을 정의한다.



(그림 5) 직원클래스.

이러한 클래스를 C++로 정의해 보면 다음과 같은데 "···" 부분은 생략된 코드 부분이다.

```
calss Employee
{
 char* name;
 positionType position;
 int salary;
 phoneNumberType phoneNumber;
 · · ·
 public:
   promote ( · · ·
   increaseSalary ( · · ·
   changePhoneNumber ( · · ·
    · · ·
}
```

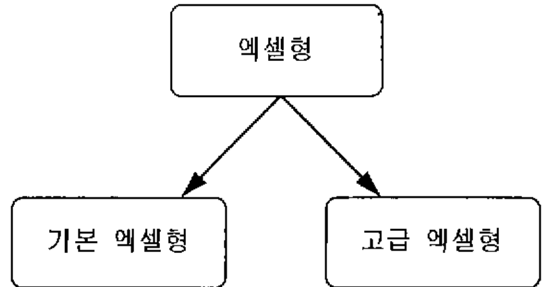
이 클래스의 정의에 따르면, Employee 클래스에 속하는 모든 객체들 즉 모든 직원들은 이름(name), 직위(position), 월급(salary) 및 전화번호(phoneNumber) 등의 변수들을 가지며 진급(promote), 월급인상(increaseSalary), 전화번호변경(changePhoneNumber) 등의 함수들을 수행시킬 수가 있다.

따라서, 클래스란 각각의 객체들이 가져야 될 속성을 정의하고 있는 본형(Template)이며 프로그램 수행시에는 이들 클래스에 근거해서 각각의 객체 즉 인스턴스를 생성시켜 동작하게 한다. 객체지향 언어 및 시스템에서는 인스턴스를 생성하는 작업을 Instantiation이라 하며 이를 위한 구문을 정의하고 있다. 예를 들어서 직원클래스에서 영희라는 인스턴스를 만들때는 C++에서 다음과 같이 표현한다.

```
Employee YoungHee ( );
     혹은
Employee* YoungHee=new Employee ( );
```

2.5 속성상속(Inheritance)

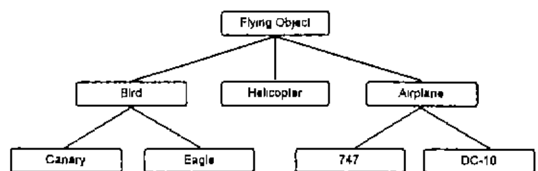
객체, 메시지 및 클래스와 더불어 중요한 객체지향 개념으로 속성상속(Inheritance)이라는 것이 있다. 이 개념 역시 우리의 일상생활에서 흔히 사용하는 개념을 프로그램으로 표현하기 위한 편리한 수단이다. 어떤 객체의 종류 즉, 클래스는 좀더 세분화 하여 분류할 수가 있는데 이렇게 세분화된 종류나 유형을 Subtype 혹은 Subclass라고 한다. 다시 자동화의 종류를 보면 (그림 6)에서 처럼 엑셀이란 자동차는 다시 기본형과 GLSi와 같이 고급형으로 나눌 수 있다.



(그림 6) 엑셀 자동차의 세분화.

이 때, 기본형이든 고급형이든 모든 엑셀자동차는 엑셀형이란 클래스가 가지고 있는 모든 특성들을 가지고 있으며 고급 엑셀형 클래스는 고급 카셋트, 더 넓은 타이어 등 추가적인 속성을 가진다. 객체지향 프로그래밍에서 "속성 상속"은 새로운 클래스를 정의할 때 모든 것은 처음부터 다 정의하는 것이 아니라 이미 존재하는 유사한 클래스를 바탕으로 하여 필요한 속성만 추가하여 정의하는 경제적인 방법을 의미한다. 이 때 새로이 생기는 클래스를 Subclass라 하고 그 바탕이 되는 클래스를 Superclass라 한다. 이렇게 하면 클래스들 사이에서 공통으로 가지는 특성, 즉 데이터 구조나 함수들을 중복해서 정의하는 일을 줄일 수 있을 뿐 아니라, 특성을 수정하거나 추가시 Superclass의 정의만 고치면 그 서브클래스들은 변경된 속성을 자동으로 상속받게 되므로 매우 편리하다.

하나의 프로그램내에서 클래스들간의 상속관계를 트리구조로 나타낸 것을 클래스 상속 구조(Class Inheritance Hierarchy)라 한다. (그림 7)은 날수 있는 모든 객체들을 클래스로 분류 정의한 상속 구조이다.



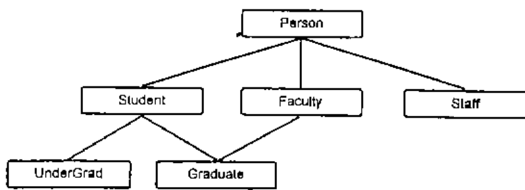
(그림 7) 나르는 객체들의 클래스 상속 구조.

날 수 있는 모든 객체들을 묘사하는 클래스는 "Flying Object"라고 정의했고 그들을 다시 새 종류, 헬

리콥터 및 항공기 등의 Subclass들로 정의했다. 다시 새 종류는 Canary와 Eagle 종류로 나누었고 항공기는 기종에 따라 Subclass들을 정의했다. Flying Object라는 수퍼클래스는 모든 날 수 있는 객체들이 가지고 있는 공통 특성들, 최고고도, 비행속도, 날개의 수 등을 정의하고 있으며 새는 생물이므로 평균수명 등의 특성을 추가 정의할 수 있다. 또한, 항공기 클래스도 최대 탑재무게, 엔진의 수 등 항공기에만 해당되는 특성을 가질 수 있다. 이 클래스 상속 구조를 잘 관찰해 보면 우리가 흔히 문제를 분류하여 이해하는 방식과 유사함을 알 수 있다. 즉, 객체지향 기술의 속성상속의 큰 장점은 주어진 문제를 사람들이 분류하는 방법 그대로 표현하게 해주는데 있다.

2.6 복수 속성상속(Mutiple Inheritance)

일반적으로 하나의 Subclass는 하나의 Superclass를 가지나 문제에 따라서는 복수의 Superclass들을 필요로 하는 경우가 있다. 이런 상속관계를 복수 속성상속(Multiple Inheritance)이라 하는데 일부의 객체지향 언어에서만 지원되고 있다. 복수 속성상속 기능은 실재의 문제를 분석하고 모델링하는데 더욱 편리함을 제공한다. (그림 8)에서는 대학교 인사관리 전산화 문제에서 자연스럽게 구상할 수 있는 상속 구조를 나타내고 있다.



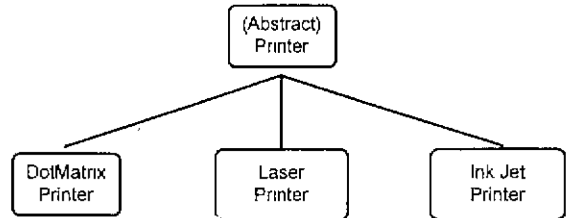
(그림 8) 대학교 인사관리 문제에서의 상속 구조.

이 상속 구조에서 대학원생의 객체를 나타내는 "Graduate"라는 클래스는 "학생(Student)"와 교수(Faculty)" 두 개의 Superclass를 가지고 있다. 이는 대학원생은 학생이면서 또한 일부는 강사로서 가르치기도 하므로 학생클래스의 속성을 상속하고 또한 교수클래스의 월급(Salary)이나 강의 과목수 등의 특성을 상속받는 것이 자연스런 방법이다.

2.7 추상클래스(Abstract Class)

클래스 중에는 인스턴트를 만들어 낼 목적이 아니라 서브클래스들의 공통된 특성을 추출하여 묘사하기 위한 클래스가 있는데 이를 추상클래스(Abstract Class, Virtual Class)라 한다. 추상클래스는 연관된 서브 클래스들의 공통 데이터 구조 즉, 변수들을 정의하고 함수 중 일부는 완전히 구현하지 않고 Signature만 정의한 것들이 있다. 이들을 추상 함수(Abstract Function)라 부르며 이들은 후에 서브클래스를 정의할 때에 그 클래스의 목적에 맞게 완전히 구현된다. 이 때 추상클래스의 한 서브 클래스가 상속받은 모든 추상 함수들을 완전히 구현했을 때, 이를 완전클래스(Concrete Class)라고 부른다. 물론, 완전클래스는 인스턴트를 만들어 낼 수가 있다.

추상클래스의 예로서 프린터 소프트웨어를 들 수 있다. 우선 모든 종류의 프린터들이 공통으로 가지는 특성을 정의한 추상클래스 "Printer"를 생각해 보자. 이 Printer 클래스는 프린터의 상태를 나타내는 변수, 프린터의 속도 등 변수를 가지고 있으며 함수로는 프린팅을 수행하는 Print 등을 생각할 수 있으나 프린터마다 프린트하는 방법이 다르므로 이 추상클래스 안에서는 Print라는 함수를 완전히 구현할 수가 없다. 다만, 이 추상클래스에는 Print라는 추상 함수의 Signature만 가지고 있고 실제의 구현은 여러 서브클래스에서 각 프린터 종류에 알맞게 하면 된다. (그림 9)는 이러한 프린터간의 상속 구조를 보여준다.



(그림 9) 추상 클래스 "Printer".

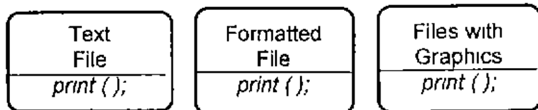
Dot Matrix Printer라는 서브클래스는 여러 개의 핀으로 구성된 프린트 헤드가 한줄 한줄씩 프린트하는 알고리듬으로 Print라는 함수를 정의하면 되고, Laser Printer라는 서브클래스는 프린트할 정보를 페이지단위로 프린터 기억장치에 저장한 후 레이저 빔을 발사해 프린트하는 알고리듬을 사용해야 된다. 이 때, Printer라는 클래스는 추상클래스로서 실존의 어떤 프린터 기능을 가지고 있지 않고 닷메트릭스나 레이저 프린터 등의 완전클래스들간의 공통된 특성만 지정하고 있으므로 그 인스턴스를 만드는 것이 무의미하다. 추상 클래스는 점진적 개발방법(Incremental De-

velopment)에 유용하게 사용될 수 있으며 프린터의 예에서 보듯 공통 속성의 추출 및 정의에 유용하므로 문제를 모델링하는데 편리함을 더해준다.

2.8 다형성(Polymorphism)

객체지향 프로그램의 중요한 특징으로 하나의 함수 이름이나 심볼이 여러 목적으로 사용될 수 있는 폴리모피즘을 들 수 있다. 객체지향에서의 폴리모피즘이란 복수의 클래스가 하나의 메시지에 대해 각 클래스가 가지고 있는 고유한 방법으로 응답할 수 있는 능력을 말한다. 즉, 별개로 정의된 클래스들이 같은 이름의 함수를 별도로 가지고 있어 하나의 메시지에 대해 각기 다른 방법으로 그 메시지를 수행할 수 있는 것을 의미한다.

예를 들어 여러 가지의 파일(File)들을 프린트하는 함수를 생각해 보자. 파일에는 간단한 텍스트파일(Text File), 문서편집기로 만든 포멧파일(Formatted File), 그래픽을 포함하고 있는 파일(Files with Graphics) 등 여러 가지가 있다. 이들 각각의 파일들을 프린트하는 방법이 다 다르다. 객체지향식으로는 (그림 10)에서 처럼 각 종류의 파일을 별도의 클래스로 정의하고 각각의 파일 종류별로 Print라는 함수를 파일의 형태에 맞게 구현하게 해준다.



(그림 10) "Print" 함수의 복수 정의.

이렇게 생성된 파일 객체들은 모두 Print라는 메시지를 이해하며 각 파일의 종류에 알맞게 프린트할 수 있다. 이렇게 폴리모피즘은 같은 이름의 함수를 여러 클래스가 각 클래스에 알맞게 달리 정의하고 같은 이름의 메시지에 응답할 수 있게 해주는 편리함을 가지고 있다.

III. 객체지향 프로그래밍의 장 · 단점

이러한 객체지향 기술의 기본 개념을 가지면 이제 객체지향 기술의 장·단점들을 쉽게 생각해 볼수 있다. 우선 중요한 장점들을 열거해 보자.

3.1 장점

3.1.1 생산성 향상

객체지향 기술에서 추구하는 궁극적 목표는 잘 설계된 클래스들 즉 소프트웨어 IC의 라이브러리를 재사용하여 소프트웨어를 조립 생산하는 것이다. 객체는 하드웨어 IC처럼 독립적이어서 이를 재 사용함으로써 생산성이 증가한다. 전산분야의 대가인 제임스 마틴(James Martin)의 보고에 의하면 객체지향 소프트웨어 개발에 기존 클래스의 재사용률이 80%에 이른다고 한다[7].

3.1.2 자연적인 모델링

객체, 클래스, 속성상속 및 폴리모피즘 등은 우리의 일상생활에서 보통사람들이 대하고 생각하는 방식을 프로그램 언어로 그대로 표현할 수 있게 해준다. 따라서, 객체지향 프로그램의 분석과 설계방법은 이미 우리에게 친숙해져 있는 자연적인 모델링(Natural Modeling) 방법으로 쉽고 효율적으로 프로그램을 개발케 해준다.

3.1.3 재사용

객체지향 프로그래밍은 코드 재사용을 극대화한다. 한 프로그램내에서 서브클래스들이 수퍼클래스의 속성을 표현한 코드를 재사용하며, 새로운 프로그램 개발시 기존 프로그램이 가지고 있는 클래스 상속 구조에서 많은 클래스들을 소프트웨어 IC로 재사용할 수 있다.

3.1.4 유지보수 용이성

객체지향 프로그램은 기존의 기능을 수정하거나 새로운 기능을 추가하기가 용이하다. 기존 기능을 수정시 함수를 새롭게 바꾸더라도 캡슐화와 그 함수의 세부정보가 은폐되어 있어 주변에 미치는 영향을 최소화 한다. 새로운 객체의 종류(클래스)를 추가시에는 속성상속을 통하여 기존의 기능을 활용하고 존재하지 않은 새로운 속성만 추가하면 되므로 매우 경제적이다.

이외에도 점진적 프로그램 개발의 용이성, 요구사항 변화에 대해 안정된 프로그램 구조 등의 장점들을 들 수 있다.

3.2 단점

객체지향 프로그램의 거의 유일한 단점으로 실행

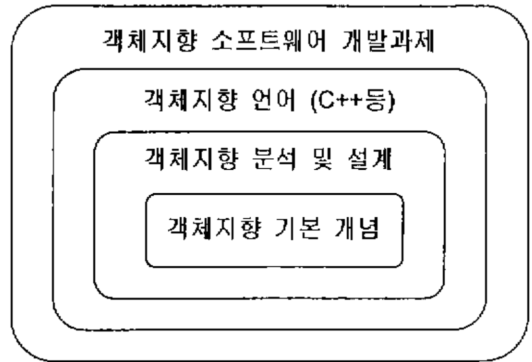
시의 속도(Runtime Efficiency)가 떨어진다고 알려져 왔다. Smaltalk이나 C++ 등의 객체지향 언어로 만들어진 응용 프로그램은 Fortran이나 C 등 재래식 언어로 만들어진 프로그램에 비해 실행 속도가 늦다는 말을 흔히 듣게된다. 이는 객체지향 프로그램은 객체라는 단위를 컴퓨터의 기억장치에 어떤 형태로든 표현해야 하고, 객체간의 정보교환이 모두 메시지 교환을 통해 일어나므로 재래식 프로그램 보다 실행 시스템에 많은 부하가 생기게 된다. 하지만, 이러한 운용시 효율성문제는 효율성이 뛰어난 새로운 언어들의 탄생으로 이제 거의 문제가 되지 않는다. 특히, 기존의 언어 C나 Pascal 등을 확장한 C++나 Object Pascal 등은 기존언어의 효율성과 객체지향의 장점들을 모두 제공한다.

IV. 객체지향 분석 및 설계

객체지향 프로그래밍 기술이 80년대에는 재래식 방법으로부터의 전환기를 거쳐왔고, 90년대에는 소프트웨어 개발 중심기술이 될 것을 대부분의 소프트웨어 기술자들은 믿어 의심치 않는다. 그러나, 이러한 추세에도 객체지향 방법으로 소프트웨어 개발을 해 본 적이 있는 이들은 극소수이며, 새로운 과제에 적용해 보려는 시도도 망설이는 경우가 많다. 또한, C++나 Smaltalk 등의 객체지향 언어로 작성된 프로그램도 그 코드를 보면 웬지 C나 Fortran 등 재래식 프로그램 냄새가 나는 것은 흔히 보게 된다.

이러한 일련의 현상들에 대한 원인은 소프트웨어 개발의 가장 기본적인 원리에서 찾아 볼 수 있다. 소프트웨어의 개발단계를 크게 분석, 설계, 구현, 시험 등으로 나눌 수 있는데 좋은 객체지향 소프트웨어를 구현하기 위해서는 그 전단계인 설계가 객체지향 방식으로 잘 되어야 한다. 또 훌륭한 객체지향식 설계는 그 분석도 객체지향식으로 수행해야만 가능하다. 즉, 위의 현상들은 객체지향식 분석(Object-Oriented Analysis, OOA)과 객체지향식 설계(Object-Oriented Design, OOD)에 익숙하지 않거나, 재래식 분석 설계 위에 객체지향식 구현을 시도하는데서 비롯된다[4,9]. 객체지향 언어를 안다는 것은 객체지향 구현단계에서 사용할 하나의 소프트웨어 툴을 배운 것에 지나지 않다. 따라서 언어 학습과는 별도로 객체지향 분석과 설계방법을 익혀야 비로소 객체지향 기술을 실제 소프트웨어 개발에 어려움 없이 적용할 수 있다.

객체지향 기술을 완전히 소화하기 위해서는 (그림



(그림 11) 객체지향 기술요소의 상관관계.

11)에서 나타난 객체지향 기술의 기본요소들을 순서대로 익혀야 한다. 즉, 객체지향 기술의 여러 응용분야에 공통이 되는 기본 개념을 우선 익힌 다음에 주어진 문제를 객체지향식으로 분석하고 설계하는 방법들을 익히고 그 후에야 특정 객체지향 언어를 익히는 것이다. 이들 세 개의 요소들 가운데 객체지향 기본개념이 가장 중요한 바탕이 되며 분석설계 방법이 그 다음 중요하다. 그 다음을 프로그램 언어라 할 수 있는데 이는 어떤 소프트웨어를 객체지향식으로 설계한 후 재래식 언어로의 구현은 가능하나, 그 반대로 재래식 방법으로 설계를 하고 객체지향 언어로 구현하는 것은 그렇게 할 어떤 장점이나 의미가 없고 언어에 따라서는 불가능하기도 하기 때문이다.

본 장에서는 이러한 객체지향 분석과 설계방법이 기존의 구조적 방법들과의 차이점 및 그 특색을 알아보고 대표적인 객체지향 분석 및 설계방법들을 살펴본다.

4.1 구조적 방법

대다수 프로그래머들에게 익숙해져 있는 방법이 구조적 분석 및 설계이다. 즉, 크고 복잡한 프로그램을 여러 개의 작은 서브 프로그램으로 나누고 각 서브 프로그램은 다시 더 작은 여러 개의 서브 프로그램으로 나누어 개발한다는 철학이다.

4.1.1 기능적 분할 방법

여러 구조적 방법들은 크게 두 가지로 분류되는데 시스템의 기능에 의해 서브 프로그램으로 나누는 방법이 가장 많이 쓰이며 이를 기능적 분할(Functional Decomposition)이라 한다. 대표적인 기능적 분할 방

법들로는 Yourdon, Constantine, DeMarco 및 Gane 등이 개발한 것들이 있다. 이들 방법의 공통적인 한계는 프로그래머들이 함수를 중심으로 설계 코드를 만들기 때문에 데이터 부분은 관련 함수들이 필요한 데로 일관성 없이 혹은 중복되어 정의되어 지는 점이다. 이는 대규모 프로그램에서 더욱 문제가 되는데 데이터에 관한 정형적인 분석과 설계 모델을 사용하지 않으므로 코드의 크기가 늘어나며 복잡도를 소화하기가 힘들게 된다.

4.1.2 이벤트-반응 모델링 방법

다른 유형의 구조적 방법으로는 McMenamin 등이 개발한 이벤트와 그 반응(Event-Response)을 관찰 문제를 서브 프로그램으로 나누는 것이다. 이 방법에서는 시스템을 하나의 블랙박스로 보고 외부에서 발생되는 이벤트와 그에 대한 반응을 시스템의 프로세스로 정의한다. 이 방법 역시 데이터 부분에 대한 정형적 모델이 결여되어 있다.

4.2 객체지향적 방법

객체지향 방법은 80년대에 Peter Chen이 고안한 정보 모델링(Information Modeling) 방법에서 발전되어 나온 것인데, 객체와 관계도형(Entity-Relationship Diagram)을 사용하여 데이터 중심의 모델링을 한다. 그러나, 객체지향 소프트웨어 개발에 적용하는 데는 캡슐화와 속성상속 및 메시지 교환 등을 지원하지 않는 한계가 있어 이 객체와 관계모델을 객체지향식으로 확장하여 아래에서 설명할 객체모델이라는 것이 개발되었다.

소프트웨어 시스템에는 세 가지의 측면이 있다. 무슨일이 수행되어 지는가에 관한 프로세스(Process), 프로세스가 그 위에서 동작하는 데이터(Data), 그리고 프로세스가 수행되어지는 순서와 시점을 지정하는 컨트롤(Control) 부분이다. 이 세 가지를 잘 분석 설계하여 코드화 하는 일이 프로그램 개발에서 큰 비중을 차지하는데, 위에서 언급한 기능적 분할방법, 이벤트-반응 모델링 방법, 객체지향 방법은 각기 강조하는 바가 다르다. 이러한 차이점을 〈표 1〉에서 보여주고 있는데 객체지향 방법에서는 데이터 모델링 부분은 가장 중요시 한다.

〈표 1〉 세 가지 방법에서 강조되는 시스템의 측면.

기능적 분할 방법	이벤트-반응 모델링	객체지향 방법
프로세스	컨트롤	데이터
컨트롤	프로세스	컨트롤
데이터	데이터	프로세스

객체지향 방법이 데이터 모델링을 강조하는 것은 프로세스, 컨트롤, 데이터 중에서 가장 안정적인(Stable) 요소가 데이터이기 때문이다. 소프트웨어의 기능이나 알고리듬이 바꾸어 질때 가장 영향을 많이 받는 부분은 프로세스이며 그 다음이 컨트롤이고 데이터는 상대적으로 변화가 작다. 구조적 방법이 객체지향 프로그램 개발에 효과적으로 사용될 수 없는 것은 이렇게 모델링 대상이 다르기 때문이다. 즉 객체지향의 객체, 클래스, 메시지, 속성상속 등의 개념을 지원하지 못하기 때문이다.

4.3 객체지향 분석 및 설계단계

객체지향 기술의 한 장점은 분석 설계 및 구현단계들 사이에 Semantic Gap이 거의 없다는 것이다. 즉 객체지향 분석단계에서는 주어진 문제안에서 객체들을 발견해 내고 이들 객체들을 분류하고 그들간의 상관관계를 분석하는 일이며, 객체지향 설계단계에서는 이런 결과를 근거로 객체들을 클래스로 정의하고 상관관계를 속성상속 단계로 정의하며 각 함수에 알고리듬을 정의하는 일이다. 객체지향 구현단계에서는 설계단계에서 정의된 클래스들을 특정언어로 거의 1 : 1 관계로 정의하면 된다. 실제 하나의 객체지향 소스 프로그램(Source Program)은 여러 개의 클래스 정의로 이루어지고 분석 초기단계에서부터 클래스란 단위로 작업을 하는 것을 생각할 때 객체지향 방법은 재래식 방법에서 문제시 되었던 단계간의 Semantic Gap을 줄이는데 매우 효과적이다[3,4].

객체지향 분석과 설계는 별개의 작업이 아니라 실제 반복적인 과정을 통해 필요한 정보를 추출 보완하는 작업이다. 각 방법론마다 다소의 차이는 있지만 일반적으로 객체지향 분석 및 설계과정은 다음의 네 단계로 이루어진다.

1. 객체들과 클래스들을 찾아낸다.
2. 각 클래스의 특성(필요한 데이터와 기능)을 정의한다.
3. 연관된 클래스들을 속성상속 구조로 정의한다.
4. 각 클래스의 특성을 구체화 한다. 즉 데이터의 구조와 알고리듬을 정의한다.

이처럼 객체지향 프로그램 설계는 클래스들의 정의로 시작되며 각 클래스가 필요한 함수와 메시지

교환 및 상속관계가 정의되면 설계가 완성되는 것이다. 한 클래스안에 정의되어 지는 항목들은 변수, 함수, 대중 인터페이스, 상속관계 등이다. 이런 요소들은 주어진 문제를 설명하고 있는 "프로그램 개발 요구서" 혹은 "문제 설명서(Problem Statement)"에서 거의 발견이 된다. 객체지향 설계기술의 대가인 Booch는 "문제 설명서를 읽고 재래식 프로그램을 개발하려면 동사에 밑줄을 긋고 객체지향 프로그램을 개발하려면 명사에 밑줄을 그어라"라고 말한다[1]. 이렇게 문제 설명서에서 분석과 설계를 체계적으로 시작하는 것이 객체지향 방법의 특징인데 이는 바로 이 문제 설명서가 개발하려는 소프트웨어의 기능과 특성을 가장 잘 묘사하기 때문에 이를 바탕으로 분석을 시작함은 매우 의미있는 일이다.

문제 설명서 분석에 관해 좀 더 알아보자. 예로 장난감 가게의 업무를 묘사한 문제설명서를 읽어보자.

> . . .
> 만일 고객이 어린이를 위하여 장난감을 사려고 어떤 가게에 들어 간다면 그 장난감이 그 어린이에게 적절한 것인지에 관하여 짧은 시간안에 조언이 제공될 수 있어야 한다. 이 조언은 어린이의 연령과 그 장난감의 특징에 의존된다. 즉 위험한 장난감이라면 어떤 어린이들에게는 부적합할 수 있고
> . . .

이런 문제 설명서에서 필요한 정보를 추출해 내는 일반적인 지침이 〈표 2〉에서 나타나 있다.

〈표 2〉 문제 설명서 분석지침.

설명서 부분	해당되는 객체지향 항목	예 제
고유 명사	인스턴스	철수
일반 명사	클래스	장난감
수행동사 (Doing Verb)	클래스의 함수	사려고
존재동사 (Being Verb)	상속 관계	위험한 장난감이라면
소유동사 (Having Verb)	구성 관계	가지고 있다.
지정동사 (Stative Verb)	불변의 조건	소유하고 있다.
형용사	변수나 클래스	부적합할 수 있고

물론 이 지침이 절대적인 것은 아니지만 클래스에서 필요로 하는 정보를 상당부분 이런 방식으로 찾아낼 수 있다. 실제의 객체지향 방법은 이들 정보를 보다 구체적이고 체계적으로 찾기 위한 단계들을 제시한다.

4.4 대표적 객체지향 분석 및 설계방법들

다수의 객체지향 분석 및 설계방법이 있지만, 아직 특별히 타 방법들보다 월등히 뛰어난 방법은 대두되고 있지 않다. 이는 객체지향 분석 및 설계의 방법과 툴들이 아직도 성숙한 단계에 이른것이 아니기 때문이지만, 여러 방법들 사이에 유사점이 많기 때문이기도 하다.

객체지향 분석과 설계에 관한 여러 가지 방법들을 잘 정리 분석한 논문을 객체지향 기술분야에서 가장 권위있는 학술지인 "Journal of Object-Oriented Programming"의 92년도 3, 4월 합본에서 찾아볼 수 있다 [2]. 이 논문에서는 12개의 대표적인 방법들을 다음의 14가지 기능과 특성별로 분석을 했다.

1. 속성상속
2. 복수 속성상속
3. 객체의 에트리뷰트(Attriutes) 표시
4. 포함관계
5. 관계 및 함수표시
6. 상태변환 도면지원
7. 이벤트 표시
8. 이벤트 추적도 지원
9. 트리거(Trigger) 지원
10. 자료 흐름도 지원
11. 전체적 동시 수행성
12. 객체내부의 동시 수행성
13. 방법론을 지원하는 툴
14. 통합개발 툴

〈표 3〉은 이 비교결과를 요약한 것이다. 첫번째 칸은 각 방법을 개발한 사람의 이름이다.

〈표 3〉 객체지향 분석(설계)방법 비교표.

Author	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Bailin	n	n	n	y	y	n	y	y	?	y	?	?	n	n
Coad	y	y	y	y	y	y	y	n	n	y	n	n	n	n
Colbert	y	y	y	y	y	y	n	n	?	n	y	y	y	y
Edwards	n	n	n	n	y	n	y	y	y	n	y	?	y	y
Gibson	n	n	n	n	y	n	y	n	n	n	?	?	n	n
Jacobson	y	y	y	n	y	y	y	y	n	y	?	?	y	y
Kurtz	y	y	n	y	y	y	y	y	y	n	y	y	n	n
Odell	y	y	y	y	y	y	y	y	y	y	y	y	n	n
Page-Jones	y	y	y	n	y	y	y	y	n	n	n	n	n	n
Rumbaugh	y	y	y	y	y	y	y	y	n	y	y	y	n	n
Shlaer	y	y	y	n	y	y	y	y	y	y	y	n	y	n
Wirfs-Brock	y	y	y	y	y	n	n	n	n	n	?	n	n	n

V. 객체지향 프로그래밍 언어

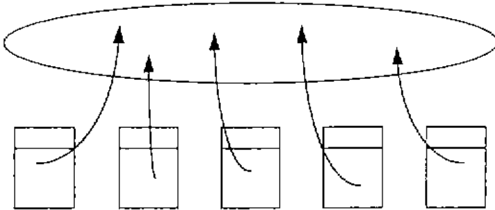
본 장에서는 객체지향 방법의 여러 개념들을 구문화하여 객체지향 설계를 프로그램으로 쉽게 표현할 수 있도록 해주는 객체지향 언어들에 대해서 알아본다.

5.1 프로그래밍 언어의 세대

프로그래밍 언어는 대개 3세대로 나눌 수 있다[1]. 54년에서 58년을 그 1세대로 보며 이 세대의 언어들로 Fortran 1, Algol 58, Flowmatic 및 IPL V 등이 있는데 주로 공학적 계산을 위해 풍부한 수학적 표현을 제공하는데 그 목적을 두고 개발되었다. 이후 59년에서 61년 사이의 제 2세대에는 Subroutine과 Block Structure에 중점을 둔 Fortran II, Algol 60, Cobol 등이 개발되었다. 제 3세대인 62년에서 70년 사이에는 PL/1, Algol 68, Pascal 등 보다 다양한 모듈화와 풍부한 기능의 언어들이 등장하여 고급 프로그래밍을 지원했다. 그 이후 70년에서 80년 사이에는 수많은 언어들이 등장했으나 크게 부각된 언어는 없다. 이를 언어발전 공백기라 한다.

5.2 제 1세대와 2세대 초반의 언어

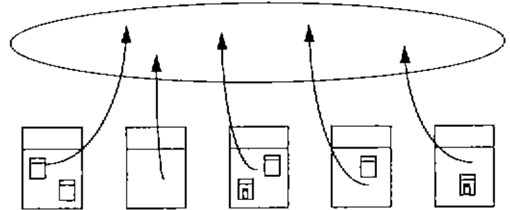
1세대와 2세대 초반의 언어들은 전역변수(Global Variable)들과 서브 프로그램들을 정의하고 모든 서브 프로그램들이 이 전역변수들을 공동으로 접근하여 그 값을 읽거나 수정할 수 있게 했다. (그림 12)은 이러한 언어들의 특징을 묘사하고 있다.



(그림 12) 제 1세대와 2세대 초반 언어들의 특징.

5.3 제 2세대 후반과 3세대 초반의 언어

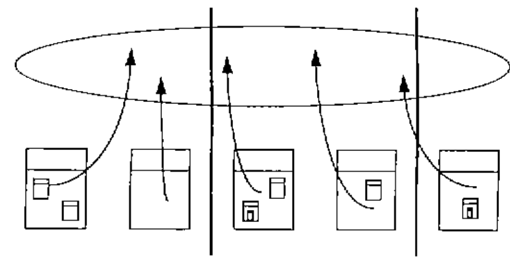
2세대 후반과 3세대 초반의 언어들은 하나의 서브 프로그램내에 또다른 서브 프로그램 즉, Nested Subprogram들을 정의할 수 있는 것이 특징이다. (그림 13)에서 보는대로 여러 내부 계층까지 서브 프로그램들을 내포 정의하여 알고리듬의 추상화(Algorithmic Abstraction)를 시도했다. 또한 이 세대에 값(Call by Value)이나 주소(Call by Address)에 의한 서브 프로그램 호출 등 여러 가지 변수를 교환하는 방식들이 고안되었다.



(그림 13) 제 2세대 후반과 3세대 초반 언어들의 특징.

5.4 제 3세대 후반의 언어

3세대 후반의 언어들의 특징은 대규모의 프로그램(Program-In-The-Large)을 개발시 흔히 여러 팀들로 나누어 코드를 작성하는 점을 고려, 한 프로그램을 여러 개의 모듈로 나누어 각 팀이 코드를 작성한 후 별도로 컴파일(Separate Compilation)을 할 수 있도록 했다. (그림 14)에서 보는 바와 같이 하나의 모듈은 여러 개의 전역 변수들과 여러 개의 서브 프로그램들로 구성되어 별도로 컴파일한 후, 컴파일된 다른 모듈들과 링크하여 완전한 실행 프로그램을 생성하게 된다.



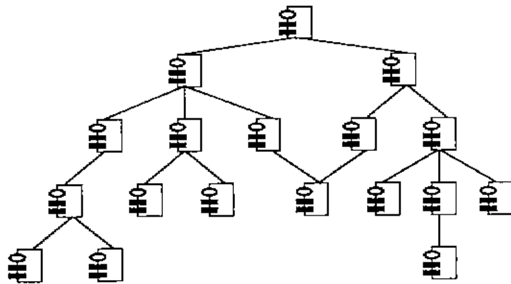
(그림 14) 제 3세대 후반의 언어들의 특징.

이러한 지난 3개의 세대들에 걸친 언어들의 공통점은 우선 전역 변수의 사용에 있다. 실제로 프로그램 개발시 많은 전역 변수들을 정의해 놓고 여러 함수들이 공용하도록 하는 것이 프로그래머들에게 매우 익숙해져 있는 방식이다. 그러나, 이러한 방식은 프로그램의 크기가 커짐에 따라 변수와 함수들의 수가

증가할 때 문제가 된다. 즉, 많은 변수들과 함수들의 연관관계를 캡슐화와 정보은폐를 사용하여 프로그램에서 쉽게 표현해 주지를 못한다. 그 결과 단순한 평면구조(Flat Structure)에서의 많은 변수와 함수가 정의되어 프로그램 이해와 수정 및 기능 추가의 어려움이 있다.

5.5 객체지향 언어의 특징

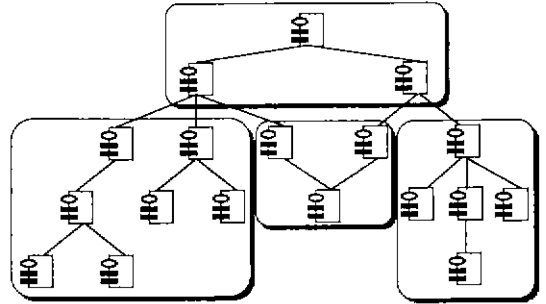
객체지향 언어들은 이런 재래식 언어들과는 매우 다른데, 우선 객체지향 언어들은 클래스 정의를 통한 캡슐화와 정보은폐를 그 기본원칙으로 하며 유사한 객체유형 사이의 속성상속 등을 제공하여 주어진 문제를 최적으로 표현하게 된다. (그림 15)에서 보듯이 캡슐화와 정보은폐를 그 기본으로 하므로 전역 변수는 순수한 객체지향 언어나 프로그램에서는 사용되지 않는다.



(그림 15) 객체지향 언어들의 특징.

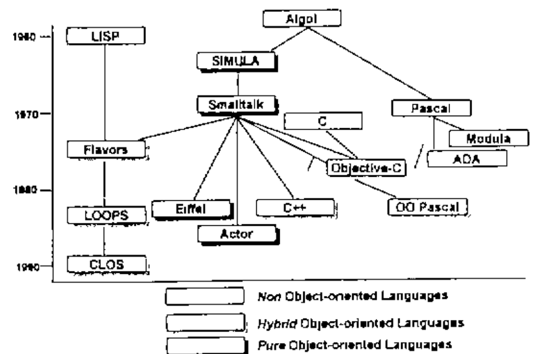
대규모 객체지향 프로그램 개발시 여러 개의 팀들이 공동 개발할 수 있도록 모듈화를 지원해야 하는데, 객체지향 프로그램에서는 여러 개의 연관된 클래스들이 군을 이루어 큰 모듈이 되며 이는 하나의 팀이 독립적으로 개발할 수 있는 단위가 된다. (그림 16)에서는 이러한 연관된 몇 개의 클래스들이 모여 하나의 모듈을 이루고 모듈들이 모여 하나의 프로그램이 되는 것을 보여준다.

객체지향 언어들은 크게 두 그룹으로 나눌수 있다. 한 가지는 순수(Pure) 객체지향 언어 그룹이며 다른 한 가지는 혼성(Hybrid) 객체지향 언어 그룹이다. 순수 객체지향 언어들은 프로그램내의 모든 것들이 다 객체하는 단위로 정의 수행된다. 따라서 전역함수를 사용하는 등 비객체지향 방식의 프로그램 작성이 불가능하다. 대표적으로 Simula, Smalltalk, Actor, Eiffel 등이 있다.



(그림 16) 모듈화된 객체지향 프로그램.

혼성 객체지향 언어들은 재래식 언어에다 객체지향 방식의 프로그램 작성도 가능토록 객체지향식 구문을 추가한다. 대표적으로 C++, Objective-C, Object Pascal, Modula 3 등이 있다. 이들 언어들의 공통적인 장점은 재래식 언어의 높은 실행효율(Runtime Efficiency)를 유지 제공하는 것이다. 객체지향 언어들은 여러 언어들에서 영향을 받아 개발 발전되어 왔는데 객체지향 관련 언어의 발전과정은 (그림 17)에서 보여준다.



(그림 17) 객체지향 언어의 발전과정.

5.6 객체지향 언어의 선택

흔히 소프트웨어 개발 계획단계에서 어떤 언어를 쓸것인가 고심하게 된다. 객체지향 소프트웨어인 경우 그 언어 선정은 더욱 그럴 것이다. 여러 면에서 절대적으로 다른 모든 언어보다 우수한 언어는 드물다. 객체지향 언어의 선택에 있어 일반적으로 고려할 사항들을 알아보자.

우선 개발하려는 어플리케이션(Application)의 성격과 요구사항 및 주변환경들을 점검한다. 어떤 어플

리케이션은 그래픽 사용자 인터페이스가 중시되어 그래픽 처리를 많이 요구하는 것도 있을수 있으며, 어떤 어플리케이션은 실행시의 효율보다도 빠른 기간내에 프로토타입(Prototype)을 만드는 것이 목적일 수 있으며 어떤 어플리케이션은 이미 개발해 놓은 재래식 언어로의 라이브러리를 활용해야 하는 제약이 있을수도 있다. 이러한 여러 사항들을 확인 열거한 후 검토대상인 객체지향 언어들의 특색과 기능 등을 나열하여 어플리케이션이 필요로 하는 사항들을 지원하는가를 비교 확인해 본다. 이렇게 함으로써 목적에 맞는 언어를 고를 수 있다.

객체지향 프로그래밍을 체계적으로 익히기 위해서는 Smalltalk 등의 순수 객체지향언어를 사용하는 것이 좋다. 이는 혼성 언어들을 사용할 때 우리에게 보다 더 익숙해져 있는 재래식 방법으로 프로그램을 작성하기가 쉬우며 이 때 컴파일러는 아무런 문제없이 이런 프로그램을 받아주므로 객체지향 프로그래밍을 익히는데 덜 효율적이다. 실례로 C++ 언어를 개발한 AT&T 내부의 많은 C++ 프로그램들을 검색 통계를 내본 결과 겨우 10분의 1 정도의 C++ 프로그램들만이 하나 이상의 클래스를 정의 사용하고 있었다고 한다.

객체지향 프로그래밍이 익숙한 상태에서는 개발용 언어로는 실행시 효율성이 재래식 언어와 별차이가 없는 C++나 Object Pascal 등 혼성언어들이 좋다. 특히 기존의 C 라이브러리를 활용해야 하는 상황에서 거의 선택의 여지없이 C++를 쓰게 될 것이며, 순수 객체지향 언어이면서도 실행시 효율성과 프로그램의 신뢰도 보장의 장점을 지닌 Eiffel언어도 다목적 객체지향 언어로 크게 각광을 받고 있다. 〈표 4〉에서는

〈표 4〉 객체지향 언어의 비교.

Features	Simula	Small-talk	Object Pascal	C++	Eiffel
Information Hiding	Yes	Yes	Yes	Yes	Yes
Inheritance	Yes	Yes	Yes	Yes	Yes
Multiple Inheritance	No	Some	No	Yes	Yes
Polymorphism	Yes	Yes	Yes	Yes	Yes
Genericity	No	No	No	No	Yes
Binding	Late/Ear	Late	Late	Late/Ear	Early
Persistency	No	Yes	No	No	Some
Concurrency	Yes	Poor	No	Poor	No
Garbage Collection	Yes	Yes	Yes	No	Yes
Object Libraries	Some	Rich	A few	A few	A few

대표적인 객체지향 언어들을 구성요소와 기능면에서 비교한다.

VI. 객체지향 소프트웨어 공학의 기타 분야

6.1 객체지향 기술의 도입

새로운 기술의 도입에 따르는 혼란과 진통은 어느 조직에서나 공통적인 현상이다. 이는 누구나 새로운 기술이나 방법을 채택함으로 인한 제품개발의 지연 등을 염려하기 때문이다. 객체지향 기술도 이미 증명되고 넓게 받아들여 지고 있는 기술의 수준이지만, 시스템 개발에 적용하려는 결정이 결코 쉬운 일은아닌것 같다. 소프트웨어 관련 기술을 받아들이는 수준에는 다섯 가지가 있다고 한다[5]. Initial Level은 아무런 기록이나 지침에 의거하지 않는 개발방법이 사용되는 수준이며, Repeatable Level이란 개발순서나 방법에 대한 일반적인 공감대가 형성되어 있는 수준을 가리킨다. Defined Level이란 정형적인 문서나 지침서로 개발방법이 잘 정의되어 있는 수준을 나타내며, Managed Level이란 Defined Level 위에 개발과정에서 체계적인 진단과 측정을 시행하는 단계이며, 마지막으로 Optimized Level이란 이러한 측정이 개발방법을 더욱 다듬어 가는데 반영이 되는 수준을 가르킨다. 일반적으로 조직에서 새로운 방법이나 기술이 Defined Level에 이르지 않으면 수용하지 않는 것이 보통이다.

오늘날의 객체지향 기술은 Defined Level에 있는 것이 일반적인 평가이다. 한단계 위인 Managed Level에서의 객체지향 기술 적용사례나 보고들도 드물지 않게 접할 수 있다. 따라서 객관적으로도 이 기술을 도입함에 결코 시기적으로 이르지 않음을 알 수가 있다. 이 분야의 권위있는 한 학자인 Jacobson은 지금의 객체지향 기술에 대해 "You should not be afraid to adopt object technology but you must do it carefully."라고 제언하고 있다[6]. 다시 말하면, 이 기술에 대한 평가의 시기는 이미 지났으며, 어떻게 도입할 것인가의 전략수립이 필요한 시기인 것이다.

새로운 기술을 성공적으로 수용하는 몇 가지 요소를 살펴보자. 우선 경영진들의 이해와 적극적인 지원이 밑바탕 되어야 한다. 이들의 지원을 바탕으로 해야만, 전 조직원들의 이해와 수용도 기대할 수 있다. 이미 OOPSLA '90 학술회에서는 경영진들을 이해시키는데 관한 특별 패널이 있었을 만큼 현실적인 문제이다[8]. 새로운 기술이 적용되는 첫 과제에 대한 관심은 매우 높으며 따라서 첫 과제의 선정에서부터 수행과정에

이르기까지 세밀한 계획과 주의를 요한다. 첫 과제에 대한 부정적인 평가는 새로운 기술 자체에 대한 평가로 인식이 되어 종종 기술 정착에 실패를 부르기도 하기 때문이다. 이외에도 과제 수행팀은 새로운 기술과 새로운 절차 등 변화에 대해 긍정적이며 의욕적인 팀원들로 구성되어야 한다. 변화에 부정적이며 소극적인 엔지니어들로는 성공적인 기술 도입을 기대하기 힘들기 때문이다.

6.2 첫 과제의 선정

첫 과제는 곧잘 평가를 위한 시험 개발이다. 그렇다고 너무 작은 규모의 소프트웨어 개발이라면 기술 평가의 목적을 달성할 수가 없을 것이다. 따라서, 첫 과제는 어느정도 중요한 상용 소프트웨어 개발 과제이어야 한다. 그러나, 개발 일정에 너무 여유가 없거나 다른 제약조건이 많다면 피하는 것이 좋다.

첫 과제는 그 문제 영역이 잘 정의되고 쉽게 이해할 수 있어야 한다. 그렇지 않을 때는 문제 자체의 이해와 정의에 많은 시간을 보내어 새로운 기술에 대한 충분한 적용 노력을 기울일 수가 없다. 과제 책임자는 새로운 기술에 많은 관심과 문제 영역에 익숙해야 하고, 팀원들 역시 신기술로 인한 변화를 경험해 본 중견 기술자들이 좋을 것이다.

충분한 계획을 미리 세워 실행함은 물론, 각 단계마다 충분한 측정을 하여, 문제가 발생시 즉각 대처할 수 있도록 한다. 새로운 기술에 대한 경험과 지식 부족으로 첫 과제에 대한 계획이 비교적 부정확할 수 있으므로 전문가의 기술자문 등이 바람직하다.

6.3 교육

새로운 기술에 대한 교육은 누구에게나 필수적인 과정이다. 특히 팀원 전체가 객체지향 개념과 방법들에 대해 공통된 이해를 가져야함은 두말할 나위없이 중요하다. 필요한 교육내용과 깊이는 조직과 개인에 따라서 차이가 나겠지만, [5]에서 제안된 내용과 기간은 〈표 5〉와 같다.

여기에는 개발 툴에 대한 교육은 포함되지 않았는데 필요하다면 객체지향 언어나 DBMS 혹은 CASE 등을 생각할 수 있다. 개발팀의 역할에 따라서 필요한 교육의 내용이 달라진다. 〈표 6〉에서는 역할에 따라 필요한 교육의 내용을 요약한 것이다.

〈표 5〉 객체지향 교육과목 및 소요기간.

과 목	기 간
객체지향 기본 개념	1 - 2 일
객체지향 과제 관리	1 일
객체지향 개발 방법론	3 - 4 일
객체지향 분석	3 - 7 일
객체지향 설계 및 구현	3 - 7 일
객체지향 테스팅	2 -3 일

〈표 6〉 역할에 따른 교육의 내용.

● 필수적 과목
○ 선택적 과목

과 목	Project Manager	Analyst	Designer/ Pgmmer	Tester	QA	Upper Manager
객체지향 기본 개념	●	●	●	○	●	○
객체지향 과제 관리	●					
객체지향 개발 방법론	●	○	○	○	●	○
객체지향 분석	○	●	●	○	○	
객체지향 설계/구현	○	○	●	○	○	
객체지향 테스팅	○			●	○	

6.4 과제위험도 측정 및 관리

과제 수행에 있어서 위험도(Risk)를 미리 측정하고 나아가 이 위험도를 줄이기 위한 수단을 강구하는 것은 적용하려는 개발방법의 종류에 무관하게 중요한 과정이다. 더구나, 객체지향이란 신기술을 적용함에 있어 위험도를 측정해 보는 것이 실패의 요소를 사전에 제거하기 위한 필수적 절차이다. 일반적으로 다음 세 단계에 걸쳐 위험도는 관리된다.

1. 위험요소 규명(Risk Identification)
2. 위험도 산출(Risk Valuation)
3. 위험도 관리(Risk Management)

첫 단계인 위험요소 규명 단계에서는 여러 가지가 있을 수 있으나, 다음의 몇 가지는 일반적인 요소들이다.

- 객체지향 기술에 대해 충분한 기술적 이해
- 개발방법과 절차에 관한 지침서의 존재 여부와 지침서의 이해도
- 객체지향 언어를 포함한 새로운 개발 툴에 대한 익숙도
- 문제 영역의 이해도 및 정의된 정도
- 여유있는 개발일정이나 판매예정일 등의 제약조건

〈표 7〉 위험도 산출의 예.

위험 요소	확률	중요도	확률 x 중요도
객체지향 기술의 이해가 미흡	3	5	15
개발 방법/지침서 이해가 미흡	2	4	8
개발용 툴의 익숙도 미흡	5	3	15
문제영역 이해도 미흡	1	4	4
...	..	..	...

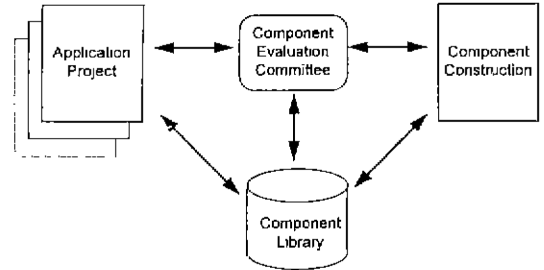
이렇게 규명된 요소들에 대하여 중요도(Consequence Degree)와 발생될 확률(Probability)을 곱하여 위험도를 산출한다. 가장 단순한 방법으로 중요도를 1(무시해도 됨)에서 5(매우 치명적인 영향)까지로 하고, 확률도 1(발생확률 거의 없음)에서 5(매우 높음)까지로 Scale을 주었을 때, 한 예로서 〈표 7〉과 같은 분석을 할 수가 있다.

이렇게 위험도가 산출이 되면 위험도를 줄이기 위한 수단을 강구한다. 즉, 발생할 확률을 낮추고 중요도를 낮춤으로 해서 전체 위험도가 줄게된다. 예로서, 개발용 툴의 익숙도가 매우 미흡하여 확률이 5인데 교육과 실습 등을 통하여 이를 1이나 2정도로 낮출수가 있다. 이외에도 매우 복잡한 위험도 측정 및 관리방법들이 있다.

6.5 객체지향 재사용

소프트웨어 재사용은 이미 오래 전부터 추구되어 왔으며, 그 중요성과 필요성은 잘 이해되고 있다. 소프트웨어 재사용은 개발비용을 줄임은 물론 시험된 모듈들을 조립식으로 재사용함으로써 품질이 향상된다. 그러나, 소프트웨어 개발에 실제로 코드가 재사용된 경우는 매우 드문 실정이다. 이처럼 소프트웨어 재사용이 잘 이루어지지 않는데는 몇 가지 주된 이유가 있다. 우선은 대부분의 과제가 매우 제한된 예산과 개발일정을 가지고 있어 재사용할 수 있는 소프트웨어를 생성하거나 혹은 찾아서 쓸 여유가 없다는 점이다. 또한, 프로그래머들의 일반적인 "Not Invented Here"의 자세에도 기인한다. 즉, 남이 만들어 놓은 코드에 대한 불신과 그 코드를 이해해야 하는 불편함의 선입감이다. 또한, 전사적 차원에서의 재사용 관리 및 지원의 표준과 시스템이 마련되어 있지 않음도 큰 이유로 든다.

객체지향 프로그래밍에 있어서 재사용은 매우 고무적이다. 앞서 2장에서 살펴본 바와 같이 객체가 가지고 있는 캡슐화와 정보은폐의 특성으로 클래스나



(그림 18) 재사용 시스템 구조.

클래스 상속 구조는 이미 재사용의 적절한 단위가 되기 때문이다. 단, 이를 관리하는 표준과 재사용 시스템의 구축이 있어야 효율적이며 지속적인 재사용을 기대할 수 있다. 재사용의 코드 단위를 "Component"라 칭한다면, (그림 18)과 유사한 재사용 시스템 구조를 생각해 볼 수 있다[5].

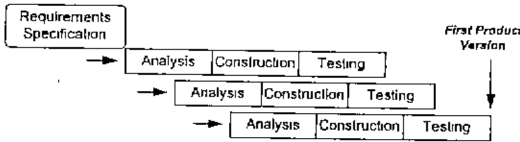
소프트웨어 공학에서의 재사용은 프로그램 코드 뿐 아니라, 재사용될 수 있는 모든 종류의 정보 즉 지식, 경험, 분석모델, 설계모델, 구조 등도 모두 포함한다. 그러나, 재사용 공학의 현수준은 코드 재사용에 머물러 있는 실정이다.

6.6 프로젝트 스텝핑

과제를 수행하는 조직과 팀원의 배치는 과제의 성격과 팀원의 배경 등 여러 요소를 고려해야 한다. 일반적으로 분석과정에 참여한 인원이 설계과정에도 참여하게 되는데 이는 두 단계를 보다 매끄럽게 연결해 주는 이점이 있다.

객체지향 기술에 비교적 생소한 팀원이 많은 경우 고려해 볼 수 있는 조직으로 전문성 위주의 스텝핑(Speciality-based Staffing)이 있다. 이는 (그림 19)에서 처럼, 점진적 개발방법(Incremental Development)과 분업의 원리를 이용한 것으로 점진적 개발의 각 싸이클에서 시스템 분석가는 Analysis의 작업만 전담하고, 설계자와 프로그래머는 Construction에 해당되는 설계 및 구현 작업만 치중한다. 테스팅도 테스팅 전문가가 이렇게 함으로써 작업의 전문성을 높이고 분업의 효율을 기대할 수 있다.

실제 개발에 종사하는 팀원들 이외에도 필요한 인력으로는 개발방법론 전문가(Methodologist), 품질보증(QA) 담당, 문서 및 매뉴얼 담당, 교육강사, 재사용 코드 관리자, 개발환경 관리자 등이 있다.



(그림 19) 전문성 위주의 스텝핑.

6.7 기타

위에서 언급한 객체지향 소프트웨어 공학의 요소들 이외에도 Human Factor의 고려, 비용 산출, 객체지향식 테스팅 등 여러 중요한 이슈들이 있다.

VII. 결 어

객체지향 기술은 90년대의 중심 소프트웨어 개발 기술로 자리를 잡아가고 있다[6]. 이는 하나의 유행이 아니라 객체지향 기술이 소프트웨어 개발의 생산성을 높이기 위한 즉 소프트웨어 위기를 극복하기 위한 매우 효과적인 기술이기 때문이다. 기술 선진국으로 발돋움하고자 하는 국가적 의지에 맞추어 우리 전산인들이 소프트웨어 위기를 효과적으로 극복하여 생산성을 높일 수 있는 객체지향 기술을 널리 활용, 국제 소프트웨어 시장에서 유리한 고지를 확보하는데 기여할 수 있기를 기대해 본다.

참 고 문 헌

1. G. Booch, *Object-Oriented Design with Applications,* Benjamin/Cummings Publishing Co., 1991.
2. D. Champeauz and P. Faure, "A Comparative Study of Object-Oriented Analysis Methods", *Journal of Object-Oriented Programming,* Vol. 5, No. 1, March/April 1992.
3. D. Embley, B. Kurtz and S. Woodfield, *Object-Oriented Systems Analysis : A Model-Driven Approach,* Yourdon Press, 1992.
4. I. Graham, *Object-Oriented Methods,* Addison-Wesley, 1991.
5. I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, *Object-Oriented Software Engineering : A Use Case Driven Approach,* Addison-Wesley, 1992.
6. I. Jacobson, "Is Object Technology Software's Industrial ? ", *IEEE Software,* Jan. 1993.
7. J. Martin and J. Odell, *Object-Oriented Analysis Design,* Prentice-Hall, 1992.
8. N. Meyrowitz. ed., *Proceedings of OOPSLA/ECOOP '90 Conference,* ACM Press, Oct. 21~25, 1990.
9. A. Paepcke, ed., *Proceedings of OOPSLA '92 Conference,* ACM Press, Oct. 18~22, 1992.
10. I. Toda, "How To Prevent The Coming Software Crisis", *IEEE Software,* May, 1992.

김 수 동

1984 미국 Missouri State University, 전산학 학사
1988 미국 University of Iowa, 전산학 석사
1991 미국 University of Iowa, 전산학 박사
1985 ~1990 미국 ComStat Corporation, 선임시스템 엔지니어
1991 ~현재 한국통신 연구개발단, 연구실장