객체지향설계

배울 내용

1. 객체지향 개념
   - 자료추상화, 상속, 동적바인딩, 다형성 등
2. 객체지향 방법론
   - 문제를 해결하기 위해 체계적으로 접근하는 방법
   - 예) UML
3. 객체지향 구현
   - 객체지향 원리를 적용한 프로그램의 스타일은 특정 언어에 상관없이 대동소이
   - 예) Java, C++
4. 객체지향 기법을 지원할 CASE 도구 사용법
   - OODesigner

1장. 객체 지향 개념의 역사

1.1 소프트웨어 공학 관점의 역사

- 60년대 Fortran, Cobol 등이 소프트웨어 개발에 이용됨
  - Assembly 언어에 비해 상대적으로 용이

- Software Crisis (late 1960's)
  - 실제 프로젝트에서의 한계
    - Too Big & Complex Problem
    - 소프트웨어 관련 비용이 갈수록 커짐
      - 비용의 단위 : MM(Man-Month), 한 사람이 한달 동안 일할 분량

- 첫 번째 시도
  - 프로그래밍 언어를 통해
    - Algol68, LISP(함수적 프로그래밍 언어)
    - 구조적 프로그래밍
      - Structured Language
        - » 블록(block) 단위로 프로그램 구성
      - Goto문 사용금지
      - 제한된 제어구조 사용 (순차, 반복, 선택)
    - 70년대 Pascal, C 언어 인기
      - Fortran, Cobol도 문법이 수정 구조적 언어가 됨
  - 한계를 극복 못함

❖ 구조적 프로그래밍 (= 절차적 프로그래밍)

- 프로그램을 잘게 나누어 구현
  - Divide & Conquer, Top-down 방법
- 데이터와 절차를 구분
- C, Cobol, Basic, Fortran
- 단점
  - 프로그램의 실행흐름, 절차에 치중
  - 데이터를 소홀히 함
  - 전체적인 이해가 잘못되면 세부도 잘못됨
- C 프로그램의 예
  - 데이터를 저장하는 곳과 사용하는 곳의 위치가 다르면, 각 함수의 인수를 통해 주고 받아야 함 → 복잡
  - 전역변수로 바꾸면 간단해지나 중요 데이터를 임의로 변경할 수도 있게 됨

❑ 다음 시도...

- 소프트웨어공학(Software Engineering)을 통해
  - 소프트웨어의 생명 주기(Life Cycle)에 대한 고찰과 수집을 통해 소프트웨어의 생산성에 심각한 영향을 미치는 단계가 **시스템의 분석과 설계 단계**임을 인식
    - 폭포수모델의 예
      - » 요구분석 단계 → 설계 단계 → 구현 단계 → 테스팅 단계 → 유지보수 단계
  - Methodology
    - 구조적 분석 및 설계 기법
      - » 자료흐름도, 구조도를 이용하여 시스템 기술
      - » 배우고 사용하기 쉬워서 많이 사용됨
    - JSD(Jackson's System Development) 기법
    - Warnier Orr의 방법론 등
    - 위 방법론들은 생명주기 각 단계별로 입출력 문서와 표기법, 문제 해결 절차를 기술

- 앞 방법론들의 특징
  - 프로세스 지향적 (process-oriented) = 구조적 기법
  - 사물을 기능적으로 봄
  - 예) 건설회사 시스템 개발 시
    - » 부서가 어떠한 절차를 거쳐서 작업을 수행하는가
    - » 어떠한 입출력 자료를 생성하는가에 초점을 두고 시스템 분석
- 80년대 구조적 기법을 위한 개발 환경 다수 발표된 시기
  - DBMS 활성화
  - 4세대 언어(4GL) 등장
  - CASE(Computer Aided Software Engineering) 도구 개발 활성화
    - » 구조적 기법에서 사용되는 표기법, 절차, 관련 문서 처리 등

- 구조적 기법(Structured Technique)의 문제점
  1. 소프트웨어의 개발 공정에 치우침
     - 소프트웨어 비용의 60-80%를 차지하는 소프트웨어 **유지 보수**를 소홀히 함
     - 실제는 유지보수(Maintenance)가 더 문제임
       - 확장(enhancement), 적응(adaptation), 수정(correction)
  2. 안정된 요구 사항만을 대상으로 구조적 방법론을 적용함
     - 설계나 구현 시 새로운 요구사항 발생하거나 잘못 해석된 요구 사항이 발견되면 전체 시스템을 처음부터 다시 분석 개발해야 됨 → 상당한 비용 소모
     - 실제는 요구사항을 확정하는 것이 불가능함
  3. 생명 주기의 각 단계가 상당히 독립적
     - 단계별 수행 작업의 차이 분명 → 표기법 다름
     - 개발자는 각 단계간 정보 변환의 부담 → 변환 분석의 어려움
     - 역공학(reverse engineering) 어려움
       - 예) 소스 코드에서 구조도와 자료흐름도 추출
  4. 기존 지식의 재사용에 대한 고려가 부족함
     - 소프트웨어 위기 극복을 위한 최후의 방법은 소프트웨어 컴포넌트를 재사용할 수 있는 방법을 모색하는 것임

❑ 객체지향 기법(OO Technique)을 통한 해결법

- 소프트웨어의 개발 및 유지보수 프로세스에 모두 관심을 갖고 있음
  - Iterative & spiral process
- 요구사항의 변화가 미치는 chain effect가 적음
  - loosely coupled system architecture thanks to data abstraction, dynamic binding ...
- Seamless Process가 가능함
  - conceptual uniformity
- 재사용이 근본 아이디어임
  - 소스 단위의 재사용

❖ 객체지향 프로그래밍

- 데이터와 절차를 하나의 단위로 묶음
- 같은 단위에 속해 있는 절차(코드)를 통해 데이터에 접근 → 중요 데이터의 보호 가능
- 예) C with classes → C++
  - C 언어도 함수포인터를 포함한 구조체를 이용하면 OOP 가능 → 복잡

1.2 객체 지향 언어의 역사

❑ Simula 67

- 노르웨이의 Dahl과 Nygaard가 개발
- 객체지향 언어의 시조
- Simulation을 위한 언어로 고안됨
- "클래스"의 도입
  - 실세계의 객체를 컴퓨터 내부에서 표현하기 위한 기본 구조 제공
  - 자료추상화를 위한 도구
- 실용적인 언어로 발전하지 못함
- 단순히 학문적인 가치로만 인정받음

❑ Smalltalk

- Xerox의 Palo Alto 연구소에서 개발
- 객체지향 언어의 실질적 원조
- 가장 순수한 객체지향 언어로 인정받음
- 하드웨어, 개발환경(클래스 라이브러리, 가상 머신, 클래스 브라우저 등), 프로그래밍 언어의 조합으로 발표됨
  - 이식성 부족

❑ Ada

- 80년대 초에 미 국방성에서 개발
- 여러 업체들이 제시한 언어들을 기준으로 Ada의 골격 정의
  - 기본적으로 Pascal 문법을 기반
  - **패키지, 예외처리** 기능(신뢰성 향상이 목적)의 도입
  - 상속 개념 없음
  - 정적 바인딩 이용

    → 프로그램의 유연성 감소, 재사용 어려움

- C++
  - 계속 진화해 가는 언어
    - 클래스, 연산자 중복, 상속, 가상 함수, 추상 클래스, 예외 처리 등이 추가되어 현재는 C++3.0까지 향상됨
  - C와 객체지향 개념을 혼합한 언어(hybrid language)
    - 오용의 소지가 있음
  - 많은 사용자들이 사용
- Java
  - 90년대 중반 이후 각광받는 언어
  - SUN사의 James Gosling이 개발
    - 단순하며 특정 CPU에 얽매이지 않는 언어
    - Oak 언어(Java의 전신)의 상품화에는 실패
  - 인터넷 응용 분야에서 선도적인 언어
  - 언어의 단순성(오용의 소지가 적음), 플랫폼 독립성

- Objective-C
  - C와 객체지향 개념을 혼합한 언어
  - Smalltalk 쪽에 가깝게 정의된 언어
- Eiffel
  - 예외 처리와 선후 조건문을 이용하여 시스템의 정확성 확보에 중점을 둔 언어
- Loops
  - 함수적 언어인 Lisp에 객체지향 개념을 도입
- Flavors, Object-Pascal
- Visual C++, Delphi, C#

1.3 객체 지향 방법론과 객체 지향 도구들의 역사

- 객체지향 방법론
  - 구조적 방법론에 비해 일관성과 종합성을 보장하며 생명 주기 각 단계에서의 정보 변환의 부담이 훨씬 적은 장점 가짐
  - 80년대 중반부터 발표되기 시작하여 최근까지 계속 진화
- 종류
  - Booch(OOD), Coad & Yourdon(OOA), Meyer, Beck & Cunningham, Rumbaugh(OMT)
  - 기타 20여 가지
  - 90년대 초반까지 춘추전국시대
  - 94년 OOPSLA 학회에서 Booch가 통합 시도 → 실패

- 객체 지향 방법론 간의 구분(1)
  - Evolutionary: 객체 지향 방법론도 결국 구조적 기법이 진화해서 고안된 것이라는 관점
    - Rumbaugh's OMT, Wasserman's OOSDM
  - Revolutionary: 객체 지향 방법론은 구조적 기법에 비해 혁명적이라는 관점
    - Booch's OOD, Wirfs-Brock's RDD
- 객체 지향 방법론 간의 구분(2)
  - 자료구조 중심적(OMT)
  - 서비스 중심적(CRC, RDD)

- Booch가 Rational 회사 설립
  - Rumbaugh와 Jacobson 합류
  - 세 사람의 주도하에 새로운 객체지향 방법론 정의

  - UML(Unified Modeling Language)
    - 설계자와 사용자가 의사 소통하는데 사용하는 모델링 언어
    - 소프트웨어 시스템을 시각화, 명세화하여 구축하고 문서를 작성하는데 사용하는 그래픽 표현 언어

    - ❖ 표기법에 대한 표준화 (메타 모델 정의)
    - ❖ 1998년 OMG에 의해 표준으로 채택됨
      → 대부분의 객체지향 방법론들은 소멸됨

- CASE(Computer Aided Software Engineering) Tool의 종류

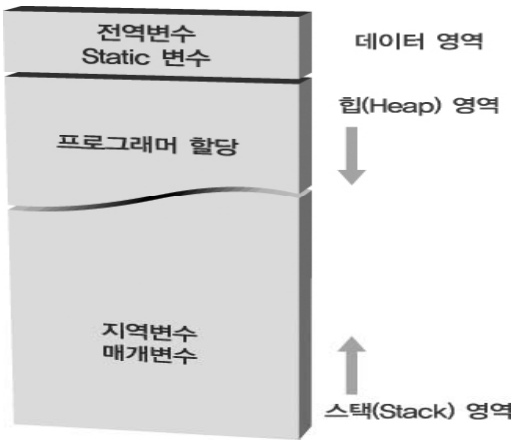
  1. 단순히 표기법만 지원하는 도구
  2. 작업 프로세스를 지원하는 도구
     - 소스 코드 생성, 역공학 등의 정보 변환 지원
  3. 메타 CASE 도구
     - 사용자가 임의로 표기법을 정의해서 사용
     - CASE를 만드는 CASE

  - Rational Rose
    - 가장 인기 있는 도구, 상용
  - OODesigner
    - UML 다이어그램 작성, 문서화 지원, 클래스 관리, C++/Java 코드 자동 생성

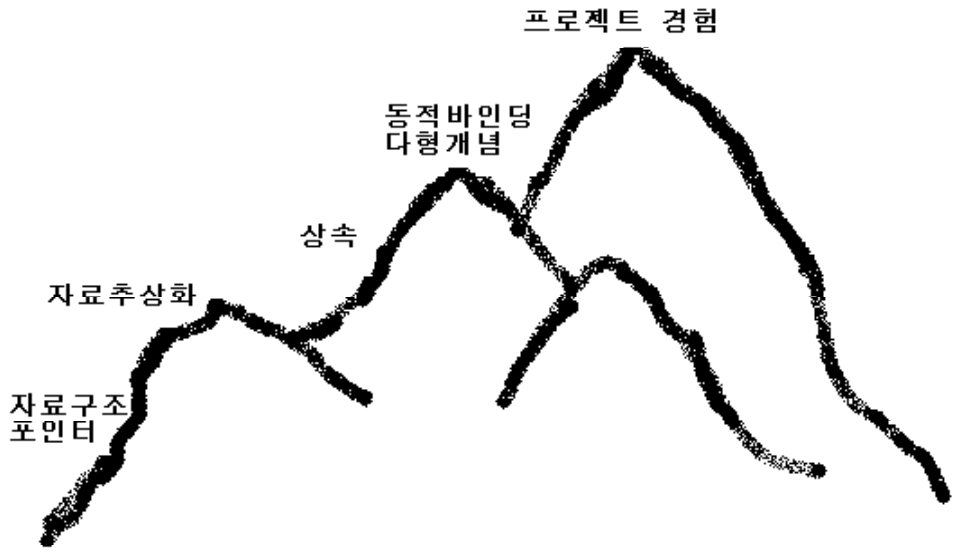
1.4 수강생이 주지해야 할 사항

- 객체 지향은 절대로 쉽게 배울 수 없다.
- 선행 지식
  - C 언어 또는 다른 구조적 언어의 문법
  - 타입(type)에 대한 이해
    - 예) 정수형이란?
      - » 자료구조 : 32비트 기억장소
      - » 연산자 : +, -, *, / 사용
  - 포인터(pointer)와 동적 할당(dynamic allocation)
    - 포인터 : 동적으로 생성된 변수에 대한 명칭의 의미

- 스택 영역과 힙 영역에 대한 구분



  - Garbage Collection
    - 동적으로 할당 받아 사용되는 기억 장소가 더 이상 사용될 필요가 없는 경우에 자동으로 수거함
  - Dangling Reference
    - 특정 객체를 가리키는 포인터가 잘못되는 경우
  - 알고리즘 표현 방식(제어문)
  - 유명한 자료 구조들
    - Array, Stack, Queue, Linked list, Tree, Graph 등



객체지향의 이해를 위한 멀고 험난한 길

- Terminology 사용의 주의
  - 예) 객체(Object) vs. 클래스(Class)
    - 구분되기도 하고, 동일한 의미로 사용되기도 함
    - 용어의 이해 및 사용 시 선입견을 버리고 문맥에 따라 용어를 이해해야 함
- 언제 객체지향 기법을 사용해야 하는가?
  - 소프트웨어의 특성과 크기에 따라 상대적
  - 객체지향 언어와 기법이 바람직한 경우
    - 소프트웨어의 크기가 10 KDSI(10,000 라인) 이상이고, 계속 확장되거나 변경될 여지가 있는 경우
- 기업에서 객체지향 기법을 도입하기 위한 부담
  - 구성원에 대한 교육에 얼마나 투자를 하는가
  - 과거에 이루어졌던 기존 투자가 무용지물이 됨

- 객체 지향의 단점
  - 난해성
    - 새로운 문법, 많은 표기법 등
  - 오용 가능성
    - 객체지향 언어나 기법을 잘못 사용하여 객체 지향의 장점을 살리지 못하는 경우 → 객체지향을 불신하게 됨
  - 느린 속도, 상대적으로 큰 사이즈 → 하드웨어의 발전