30 분만에 알아보는 객체지향 프로그래밍

Intro. to Object-Oriented Programming in 30min.

안기영 (安基榮) Ki Yung Ahn

School of Computer Science and Engineering
Nanyang Technological University, Singapore

주제발표 (공개강의형식)
2018-01-08T16:30+09:00
한남대학교 공과대학 3 층 90312 호 (X-터미널실)

Outline

1. 배경
   - 프로그래밍 패러다임 비교
   - 역사 - GO TO 에서 OOP 까지
2. C++ 로 OOP 맛보기
   - 간단한 톨게이트 시뮬레이션 작성 (객체지향 X)
   - 톨게이트 시뮬레이션 OOP 로 재작성
3. 개념 정리 및 심화학습 길잡이
   - 개념 정리
   - 더 알아보기

Outline

대표적인 4 가지 프로그래밍 패러다임

객체지향 (OO) 프로그래밍을 명령형 (imperative) 프로그래밍의 발전된 형태로 분류하기도

명령형 (Imperative)

컴퓨터의 상태 (메모리) 를 변화 시킬 수 있는 명령을 나열한다

객체지향 (Object-Oriented)

사람이 세상 물건/대상 (object) 들의 동작을 이해하는 관점처럼

- 물건은 컴퓨터 말고도 많다
- 물건끼리 상호작용하며 각각의 물건 상태 변화
- 최초 동기: 시뮬레이션

함수형 (Functional)

기존의 값에 함수를 적용해 새로 필요한 값을 얻는다

- 값 (value) 중심
- 식 (expression) 중심

논리 (Logic)

결과 계산 방법/절차 대신 원하는 조건을 논리식 또는 논리규칙의 형태로 정의만 하면 자동으로 그 조건을 만족하는 해를 탐색

대표적인 4 가지 프로그래밍 패러다임

객체지향 (OO) 프로그래밍을 명령형 (imperative) 프로그래밍의 발전된 형태로 분류하기도

명령형 (Imperative)

컴퓨터의 상태 (메모리) 를 변화 시킬 수 있는 명령을 나열한다

객체지향 (Object-Oriented)

사람이 세상 물건/대상 (object) 들의 동작을 이해하는 관점처럼

- 물건은 컴퓨터 말고도 많다
- 물건끼리 상호작용하며 각각의 물건 상태 변화
- 최초 동기: 시뮬레이션

함수형 (Functional)

기존의 값에 함수를 적용해 새로 필요한 값을 얻는다

- 값 (value) 중심
- 식 (expression) 중심

논리 (Logic)

결과 계산 방법/절차 대신 원하는 조건을 논리식 또는 논리규칙의 형태로 정의만 하면 자동으로 그 조건을 만족하는 해를 탐색

Outline

최초의 객체지향언어 (OOPL) 개발 동기로 제시된 문제

1965 년 노르웨이 컴퓨팅 센터에서 최초의 OOPL 인 Simula 를 개발한 Ole-Johan Dahl 과 Kristen Nygaard 는 여러 종류의 차량이 톨게이트를 통과하는 시나리오를 이산 사건 시뮬레이션 (DES) 으로 표현하기 적합한 언어를 설계

Figure: 서울 톨게이트 (출처: 위키미디어커먼즈)

GO TO 에서 OOP 까지

- FORTRAN (50 년대 후반 IBM 에서 John W. Backus 의 팀이 개발)
  - 최초로 상용 컴파일러가 제공된 고급언어
  - GO TO 문으로 흐름제어 (반복문, 서브루틴 X)
- Structured Programming / Procedural Programming 패러다임
  - 스파게티 코드 양산하는 GO TO 좀 그만 쓰자! (Edsger W. Dijkstra, 1968)
  - 반복문, 코드블럭, 서브루틴, 프로시저 등을 언어 기능 제공 (예: ALGOL 60)
- Record Handling. C. A. R. Hoare, 1966
  - 데이타와 관련 연산을 개념적 단위로 묶는 record class 제안
- Simula (1967 년 NCC 에서 Ole-Johan Dahl 과 Kristen Nygaard)
  - 이산 사건 시뮬레이션 (Discrete Event Simulation) 을 위한 언어로 개발
  - 클래스, 상속, 가상메소드, 코루틴, 자동 메모리 관리
- Smalltalk (1980 년 제록스 파크에서 Alan Kay 외 3 명. GUI 개발 동기)
  - 더 순수한 객체지향 (정수 포함 모든 값이 Object 이며 Class 조차 Object)
  - Obj-C(애플의 주 개발 언어였음), Scratch(교육용 프로그래밍 환경) 에 영향
  - 리팩토링, 유닛 테스트 등의 SW 공학적 개념 도입

GO TO 에서 OOP 까지

- FORTRAN (50 년대 후반 IBM 에서 John W. Backus 의 팀이 개발)
  - 최초로 상용 컴파일러가 제공된 고급언어
  - GO TO 문으로 흐름제어 (반복문, 서브루틴 X)
- Structured Programming / Procedural Programming 패러다임
  - 스파게티 코드 양산하는 GO TO 좀 그만 쓰자! (Edsger W. Dijkstra, 1968)
  - 반복문, 코드블럭, 서브루틴, 프로시저 등을 언어 기능 제공 (예: ALGOL 60)
- Record Handling. C. A. R. Hoare, 1966
  - 데이타와 관련 연산을 개념적 단위로 묶는 record class 제안
- Simula (1967 년 NCC 에서 Ole-Johan Dahl 과 Kristen Nygaard)
  - 이산 사건 시뮬레이션 (Discrete Event Simulation) 을 위한 언어로 개발
  - 클래스, 상속, 가상메소드, 코루틴, 자동 메모리 관리
- Smalltalk (1980 년 제록스 파크에서 Alan Kay 외 3 명. GUI 개발 동기)
  - 더 순수한 객체지향 (정수 포함 모든 값이 Object 이며 Class 조차 Object)
  - Obj-C(애플의 주 개발 언어였음), Scratch(교육용 프로그래밍 환경) 에 영향
  - 리팩토링, 유닛 테스트 등의 SW 공학적 개념 도입

GO TO 에서 OOP 까지

- FORTRAN (50 년대 후반 IBM 에서 John W. Backus 의 팀이 개발)
  - 최초로 상용 컴파일러가 제공된 고급언어
  - GO TO 문으로 흐름제어 (반복문, 서브루틴 X)
- Structured Programming / Procedural Programming 패러다임
  - 스파게티 코드 양산하는 GO TO 좀 그만 쓰자! (Edsger W. Dijkstra, 1968)
  - 반복문, 코드블럭, 서브루틴, 프로시저 등을 언어 기능 제공 (예: ALGOL 60)
- Record Handling. C. A. R. Hoare, 1966
  - 데이타와 관련 연산을 개념적 단위로 묶는 record class 제안
- Simula (1967 년 NCC 에서 Ole-Johan Dahl 과 Kristen Nygaard)
  - 이산 사건 시뮬레이션 (Discrete Event Simulation) 을 위한 언어로 개발
  - 클래스, 상속, 가상메소드, 코루틴, 자동 메모리 관리
- Smalltalk (1980 년 제록스 파크에서 Alan Kay 외 3 명. GUI 개발 동기)
  - 더 순수한 객체지향 (정수 포함 모든 값이 Object 이며 Class 조차 Object)
  - Obj-C(애플의 주 개발 언어였음), Scratch(교육용 프로그래밍 환경) 에 영향
  - 리팩토링, 유닛 테스트 등의 SW 공학적 개념 도입

GO TO 에서 OOP 까지

- FORTRAN (50 년대 후반 IBM 에서 John W. Backus 의 팀이 개발)
  - 최초로 상용 컴파일러가 제공된 고급언어
  - GO TO 문으로 흐름제어 (반복문, 서브루틴 X)
- Structured Programming / Procedural Programming 패러다임
  - 스파게티 코드 양산하는 GO TO 좀 그만 쓰자! (Edsger W. Dijkstra, 1968)
  - 반복문, 코드블럭, 서브루틴, 프로시저 등을 언어 기능 제공 (예: ALGOL 60)
- Record Handling. C. A. R. Hoare, 1966
  - 데이타와 관련 연산을 개념적 단위로 묶는 record class 제안
- Simula (1967 년 NCC 에서 Ole-Johan Dahl 과 Kristen Nygaard)
  - 이산 사건 시뮬레이션 (Discrete Event Simulation) 을 위한 언어로 개발
  - 클래스, 상속, 가상메소드, 코루틴, 자동 메모리 관리
- Smalltalk (1980 년 제록스 파크에서 Alan Kay 외 3 명. GUI 개발 동기)
  - 더 순수한 객체지향 (정수 포함 모든 값이 Object 이며 Class 조차 Object)
  - Obj-C(애플의 주 개발 언어였음), Scratch(교육용 프로그래밍 환경) 에 영향
  - 리팩토링, 유닛 테스트 등의 SW 공학적 개념 도입

GO TO 에서 OOP 까지

- FORTRAN (50 년대 후반 IBM 에서 John W. Backus 의 팀이 개발)
  - 최초로 상용 컴파일러가 제공된 고급언어
  - GO TO 문으로 흐름제어 (반복문, 서브루틴 X)
- Structured Programming / Procedural Programming 패러다임
  - 스파게티 코드 양산하는 GO TO 좀 그만 쓰자! (Edsger W. Dijkstra, 1968)
  - 반복문, 코드블럭, 서브루틴, 프로시저 등을 언어 기능 제공 (예: ALGOL 60)
- Record Handling. C. A. R. Hoare, 1966
  - 데이타와 관련 연산을 개념적 단위로 묶는 record class 제안
- Simula (1967 년 NCC 에서 Ole-Johan Dahl 과 Kristen Nygaard)
  - 이산 사건 시뮬레이션 (Discrete Event Simulation) 을 위한 언어로 개발
  - 클래스, 상속, 가상메소드, 코루틴, 자동 메모리 관리
- Smalltalk (1980 년 제록스 파크에서 Alan Kay 외 3 명. GUI 개발 동기)
  - 더 순수한 객체지향 (정수 포함 모든 값이 Object 이며 Class 조차 Object)
  - Obj-C(애플의 주 개발 언어였음), Scratch(교육용 프로그래밍 환경) 에 영향
  - 리팩토링, 유닛 테스트 등의 SW 공학적 개념 도입

GO TO 로 인한 스파게티 코드의 예

```
1  C     A weird program for calculating Pi written in Fortran.
2  C     From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.
3
4        PROGRAM PI
5        DIMENSION TERM(100)
6        N=1
7     3  TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
8        N=N+1
9        IF (N-101) 3,6,6
10    6  N=1
11    7  SUM98 = SUM98+TERM(N)
12       WRITE(*,28) N, TERM(N)
13       N=N+1
14       IF (N-99) 7, 11, 11
15    11 SUM99=SUM98+TERM(N)
16       SUM100=SUM99+TERM(N+1)
17       IF (SUM98-3.141592) 14,23,23
18    14 IF (SUM99-3.141592) 23,23,15
19    15 IF (SUM100-3.141592) 16,23,23
20    16 AV89=(SUM98+SUM99)/2.
21       AV90=(SUM99+SUM100)/2.
22       COMANS=(AV89+AV90)/2.
23       IF (COMANS-3.1415920) 21,19,19
24    19 IF (COMANS-3.1415930) 20,21,21
25    20 WRITE(*,26)
26       GO TO 22
27    21 WRITE(*,27) COMANS
28    22 STOP
29    23 WRITE(*,25)
30       GO TO 22
31    25 FORMAT('ERROR IN MAGNITUDE OF SUM')
32    26 FORMAT('PROBLEM SOLVED')
33    27 FORMAT('PROBLEM UNSOLVED', F14.6)
34    28 FORMAT(I3, F14.6)
35       END
36
```

앞의 FORTRAN 프로그램과 같은 알고리듬을 C++ 로 작성

```cpp
#include <iostream>
#include <cmath>
#include <limits>
using namespace std;
const int N = 100;  // 이 코드는 다음 사이트를 참고하여 작성됨
                    // http://www.dreamincode.net/forums/topic/132033-pi-approximation/
int main(void) {
    double sum = 0, term, sum98, sum99, sum100;

    for (int n = 1; n <= N; ++n) { // 수열을 합산해 나가는 반복문
        term = (double)(pow(-1, n+1)) * (4.0 / (2.0*n-1));
        cout <<n <<" "<<term <<endl;
        sum += term;
        switch (N - n) { case 2: sum98  = sum; break;
                         case 1: sum99  = sum; break;
                         case 0: sum100 = sum; break;
                         default: break;
        } // 마지막 3단계의 합산이 switch 문에서 저장됨
    }
    cout.precision(numeric_limits<double>::max_digits10);
    cout <<sum98 <<endl <<sum99 <<endl <<sum100 <<endl;
    // 아래 조건은 N이 짝수임에 의존하는 검사 (홀수이면 부등호 셋 다 뒤집어야)
    if (sum98<3.141592 && sum99>3.141592 && sum100<3.141592) {
        double comans = ((sum98+sum99)/2 + (sum99+sum100)/2) / 2;
        if (comans>3.1415920 && comans<3.1415930) // 소수점 6째 자리까지 정확도 검사
            cout <<"PROBLEM SOLVED" <<endl;
        else cout <<"PROBLEM UNSOLVED "<<comans <<endl;
    } else { cout <<"ERROR IN MAGNITUDE OF SUM" <<endl; }

    return 0;
}
```

Outline

단순화된 톨게이트 문제 가정

톨게이트 총 개수는 고정되어 있으나 종류는 가변일 때, 차량의 분포 데이터 (차종 및 하이패스 장착 비율) 를 기반으로 컴퓨터 시뮬레이션을 통해 효과적인 톨게이트 종류 분배를 찾고자 한다.

- 톨게이트는 두 종류가 있다 (일반 / 하이패스)
- 차량은 세 종류 (소형 / 중형 / 대형) 로 분류
- 일반 톨게이트의 통과시간은 차종에 따라 다르다 (소형 < 중형 < 대형)
- 하이패스 톨게이트 통과시간은 차종에 관계없이 같다
  - 일반 톨게이트보다는 훨씬 빠르게 통과
    (즉, 하이패스 << 소형차 일반 < 중형차 일반 < 대형차 일반)
- 하이패스 장착 차량은 두 종류 모두 통과 가능
  미장착 차량은 일반 톨게이트만 통과 가능
- 차들이 한줄로 서서 앞차를 추월하지 않고 기다리며
  맨 앞 차는 통과중인 차량이 없고 통과가능한 톨게이트가 생기면 진입

수치값과 배열로만 시뮬레이션 작성 (1)

```
7  enum GateType { ORDINARY, HIGPASS }; // 톨게이트 종류 (일반, 하이패스)
8  enum CarType { SMALL, MEDIUM, LARGE }; // 차종 (소형, 중형, 대형)
9
10 const int NUM_GATES = 10;  // 총 게이트 개수
11 const int NUM_CARS = 1000; // 총 차량 대수
12
13 GateType gate_type[NUM_GATES] // 게이트 종류를 나타내는 배열
14   = { ORDINARY, ORDINARY, ORDINARY, ORDINARY                    // 일반 게이트 4개
15     , HIGPASS, HIGPASS, HIGPASS, HIGPASS, HIGPASS, HIGPASS // 하이패스    6개
16     }; // 위 설정값을 바꿔가며 시뮬레이션하여 최적의 분배를 찾는다
17 int gate_busy_until[NUM_GATES] = { 0 }; // 다음 차량 진입 가능한 시각 (tick)
18
19 CarType car_type[NUM_CARS];  // 차량 종류를 나타내는 배열
20 bool car_highpass[NUM_CARS]; // 차량 하이패스 장착 여부를 나타내는 배열
21
22 int ordinary_gate_passing_time(CarType ct) {
23     switch (ct) {  // 차종별 일반 게이트 통과에 걸리는 시간 (tick)
24         case SMALL:  return 100;
25         case MEDIUM: return 120;
26         case LARGE:  return 500;
27         default:     return -1; } // negative return value for error
28 }
29
30 int highpass_gate_passing_time(CarType ct) {
31     return 10;     // 하이패스 게이트 통과에 걸리는 시간 (tick)
32 }
```

수치값과 배열로만 시뮬레이션 작성 (2)

```
34  void init_cars(unsigned long rseed) { // 차량 정보 초기화
35    default_random_engine randgen(rseed);
36    uniform_real_distribution<double> urdist(0,1);
37
38    double dice;
39    for (int i = 0; i < NUM_CARS; ++i) {
40      dice = urdist(randgen); // car_type배열 확률적 초기화
41      car_type[i] = (dice < 0.10)? LARGE;  // 10% 대형
42                  : (dice < 0.45)? MEDIUM; // 35% 중형
43                  :                SMALL;  // 55% 소형
44      dice = urdist(randgen);
45      switch (car_type[i]) {  // car_highpass배열 확률적 초기화
46        case SMALL:  car_highpass[i] = dice < 0.60; break; // 소형 60%
47        case MEDIUM: car_highpass[i] = dice < 0.75; break; // 중형 75%
48        case LARGE:  car_highpass[i] = dice < 0.90; break; // 대형 90%
49      }
50    }
51  }
```

수치값과 배열로만 시뮬레이션 작성 (3)

```
53  int main(int argc, char* argv[]) {
54    init_cars(0); // car_type, car_highpass 초기화
55
56    int tick = 0; // 시뮬레이션 시작 시점으로부터 흐른 시간을 나타내는 변수
57    int i = 0;
58    while (i < NUM_CARS) { // 모든 차량이 톨게이트로 진입할 때까지 반복
59      for (int j = 0; j < NUM_GATES; ++j)
60        if (gate_busy_until[j] <= tick) { // j번 톨게이트로 진입 가능한지 검사
61          switch (gate_type[j]) {
62            case HIGPASS:
63              if (car_highpass[i]) { // 하이패스 톨게이트엔 하이패스 장착 차량만 통과
64                gate_busy_until[j] = tick + highpass_gate_passing_time(car_type[i++]);
65                cout <<"car["<<i<<"] entering highpass gate["<<j<<"] at tick "<<tick <<endl;
66              } break;
67            case ORDINARY:
68              gate_busy_until[j] = tick + ordinary_gate_passing_time(car_type[i++]);
69              cout <<"car["<<i<<"] entering ordinary gate["<<j<<"] at tick "<<tick <<endl;
70              break;
71          } // end switch
72        } // end if
73      ++tick; // 단위 시간만큼 시간의 흐름을 시뮬레이션
74    } // end while
75    cout <<tick <<endl; // 마지막 차량이 톨게이트로 진입한 직후 시간 출력
76
77    return 0;
78  }
```

Outline

배열만 가지고도 시뮬레이션 잘 했는데 OOP 는 또 왜?

- 한 대상의 정보 혹은 속성 (attribute) 이 여러 배열에 따로 흩어져 있다
  - 차량의 속성
    - car_type (차종 배열)
    - car_highpass (하이패스 장착 여부 배열)
  - 톨게이트의 속성
    - gate_type (게이트 종류 배열)
    - gate_busy_until (다음 차량 진입 가능 시각 배열)
- 한 대상과 관련된 동작 (behavior) 들도 여러 함수에 걸쳐 흩어져있다
  - 곧이어 나오는 슬라이드들에서 자연스럽게 설명될 예정
- 새로운 요구조건으로 인한 코드 유지보수에 취약점
  - 예) 새로운 차종 분류 추가 필요상황
    - 근처 차선 확장 공사 개시로 포크레인 등 중장비 특수차량 등장
  - 어느 함수들의 어느 부분을 수정해야 하는지
    프로그램 규모가 커질수록 일일히 추척하는 부담 증가

객체 모델링은 대화를 상상하라

시뮬레이션을 톨게이트와 자동차가 참여하는 대화 형식의 시나리오로
[1 인칭 관찰자 (= 시뮬레이션 진행자) 시점]

- 3 번 게이트에 질문: 3 번 게이트야 너 바쁘니?
- 3 번 게이트가 응답: 아니.
- 3 번 게이트에 요청: 그럼 지금 기다리는 5 번 차량 진입시켜 줄 수 있어?
- 3 번 게이트가 응답하는 과정:
  - (잠깐, 그러려면 5 번 차량한테 일단 확인할 게 있어)
    - 5 번 차량에게 질문: 너 하이패스 장착했어?
    - 5 번 차량의 응답: 아니
  - 3 번 게이트의 응답: (난 하이패스 게이트니까 안되겠다.) 아니, 못해.
- (그럼 다음 게이트한테 물어봐야겠군)
- 4 번 게이트에게 질문: 4 번 게이트야 너 바쁘니?
- ...

대화 시나리오 바탕으로 시뮬레이션 작성

```
int main(int argc, char* argv[]) {
  init_cars(0); // 차량 정보 초기화

  int tick = 0; // 시뮬레이션 시작 시점부터 흐른 시간
  int i = 0;
  while (i < NUM_CARS) { // 모든 차량 다 진입할 때까지 반복
    for (int j = 0; j < NUM_GATES; ++j) {
      if ( gate[j]->busy(tick) ) // j번 게이트야 지금(tick) 바빠?
        continue; // 바쁘면 다음 게이트에 물어보러 진행
      if ( gate[j]->pass(car[i], tick) ) // i번 차량 지금 통과 요청
        ++i; // 요청 성공! 다음 차량 차례
    } // end for
    ++tick; // 단위 시간만큼 시간의 흐름을 시뮬레이션
  } // end while
  cout <<tick <<endl; // 마지막 차량 톨게이트 진입 직후 시간 출력

  return 0;
}

// `차량'과 `톨게이트'라는 상위개념만 시뮬레이션 시나리오 작성에
// 나타나며 차량의 하위개념(소형/중형/대형 차량)과
// 톨게이트의 하위개념(일반/하이패스 톨게이트)는 나타나지 않는다.
```

개념적 계층 구조에 따라 클래스 정의 및 상속 (1)

```
class Car { // 상위 클래스
    bool highpass;
    int default_passing_time;
  public:
    Car(bool hp, int t) : highpass(hp), default_passing_time(t) { }
    bool has_highpass() { return highpass; }
    virtual int gate_passing_time(Gate* gate) {
      if (highpass)
          return gate->adjusted_pass_time(default_passing_time);
      else return default_passing_time;
    }
    virtual ~Car() = 0; // pure virtual destructor
};
Car::~Car() { }

// SmallCar,MediumCar,LargeCar는 상속을 통해 정의된 Car의 하위 클래
    스들
class SmallCar : public Car { public:
      SmallCar(int hp = 0) : Car(hp, 100) {} };
class MediumCar: public Car { public:
      MediumCar(int hp = 0): Car(hp, 120) {} };
class LargeCar : public Car { public:
      LargeCar(int hp = 0) : Car(hp, 500) {} };
```

개념적 계층 구조에 따라 클래스 정의 및 상속 (2)

```
class Gate { // 상위 클래스
    int busy_until;
  public:
    Gate(int tick = 0) : busy_until(tick) { }
    bool busy(int tick) { return (tick < busy_until); }
    virtual bool pass(Car* car, int tick) {
      busy_until = tick + car->gate_passing_time(this);
      return true;
    }
    virtual int adjusted_passing_time(default_passing_time) = 0;
};

class OrdinaryGate : public Gate { // 상속을 통한 Gate의 하위 클래스
  public: OrdinaryGate(int tick = 0) : Gate(tick) { }
      int adjusted_passing_time(int t) override { return t; }
};

class HighpassGate : public Gate { // 상속을 통한 Gate의 하위 클래스
  public: HighpassGate(int tick = 0) : Gate(tick) { }
     bool pass(Car* car, int tick) override {
       if ( car.has_highpass() ) {
         busy_until = tick + car->gate_passing_time(this);
         return true;
       } else return false;
     }
     int adjusted_passing_time(int t) override { return 10; }
};
```

새로운 차종을 추가하는 게 이렇게 간단해진다니!

```
// 게이트 종류에 관계없이 그냥 빨리 통과해버리는 응급차
class EmergencyCar : public Car {
    public:
        EmergencyCar() : Car(true, 10) { }
        int gate_passing_time(Gate* gate) override { return 10; }
};
```

동적 디스패치 - 드러나 보이지는 않았지만 매우 중요한 개념

- main 함수의 시뮬레이션 시나리오에서는 상위 클래스 타입 (Car, Gate) 에 대한 메소드 (멤버함수) 호출만 이루어지고 있다
- 그런데 실제로 프로그램이 실행될 때는 각각의 객체 하위 클래스의 특성에 맞게 동작하게 되어 있다
- C++ 에서는 상위 클래스에서 virtual 키워드로 선언된 가상 메소드가 상속받은 하위 클래스에서 재정의 (override) 되었을경우 프로그램 코드만 보면 상위 클래스 타입에 대해 호출했더라도 실행시 실제 객체의 값이 그 하위 클래스에 해당하면 상위 클래스 메소드가 아닌 하위 클래스에서 재정의된 메소드가 호출된다

Outline

개념 정리

- 객체지향 프로그래밍의 장점은 하위개념 차원의 추가/변경이 있더라도 상위개념의 프로토콜로 작성된 시나리오는 영향을 받지 않는다는 점
  - 예) 톨게이트 시뮬레이션에 새로운 차종 분류 추가
- 프로토콜은 객체간에 교환가능한 메시지로 규정된다 (대화를 상상하라)
  - C++ 에서는 클래스의 공개 멤버함수 (또는메소드) 들에 해당
- 객체가 메시지를 받았을 때 (즉, 멤버함수 호출) 하위개념 특성에 맞게 동작해야 하는데 이를 가능케 하는 매커니즘을 "**동적 디스패치**" 라고 한다.
  - C++ 에서는 상위 클래스의 가상 (virtual) 멤버함수를 하위 클래스에서 재정의 (overriding) 함으로써 구현
- C++ 에서는 상속 (inheritance) 을 통해 동적 디스패치 구현이 가능하다. 하지만 상속이 언어 기능에 없고 다른 방식으로 동적 디스패치를 구현하는 언어들도 있다 (예: Self, JavaScript).
  - 동적 디스패치가 OOP 를 지원하기 위한 핵심 기능
  - 상속은 Simula, C++ 등에서 동적 디스패치 효과적 지원을 위해 택한 방식

Outline

더 알아보기

- 다양한 DES 언어 및 소프트웨어
  - GPSS (1961 IBM 에서 최초 버전) - Simula 보다 단순해 구현/학습 용이.
  - Arena, AnyLogic (개인용무료), SimEvent (MATLB/Simulink 환경)
  - 최근에는 오픈 소스 DES 소프트웨어도 많아지고 있다
- Graphical User Interface (GUI) -OOP 가 많이 활용되는 대표적 분야
- 클래스 기반 OOP vs 프로토타입 기반 OOP
  - 클래스 기반 OOP
    - 정적 타입 언어에서 시너지 효과 (정적 타입 오류, 컴파일러 최적화)
  - 프로토타입 기반 OOP
    - 장점 - 유연성 (클래스 기반에서 흔치 않은 기능 활용이 용이)
    - 단점 - 자동화 도구 (컴파일러 최적화, 리팩토링 등) 지원이 더 어렵다
- Actor Model (OOP 창시자들이 진짜로 꿈꾸던 비전은 혹시 이런 방향?)
  - OOP 발전 과정에서 직렬 및 공유메모리 멀티스레드 컴퓨팅 환경에 오염
  - 그런 가정 없는 순수한 메시지 기반 이론을 토대로 병렬/분산 환경의 동시성 문제에 적합한 프로그래밍 패러다임을 추구하려는 움직임도

참고자료 I

- https://www.cs.cmu.edu/~charlie/courses/15-214/2014-fall/slides/25-history-oo.pdf
- https://craftofcoding.wordpress.com/2013/10/07/what-is-spaghetti-code/
- http://dist-prog-book.com/chapter/3/message-passing.html