5장. 객체 지향 프로그래밍

- 좋은 아키텍처를 만드는 일은 객체지향 설계원칙을 이해하고 응용하는 데서 출발한다.
- 객체 지향은 무엇인가?
  - “데이터와 함수의 조합”
  - 실제 세계를 모델링하는 새로운 방법
  - 캡슐화, 상속, 다형성의 개념을 적절하게 조합한 것이거나 최소한 세 가지 요소를 반드시 지원해야한다.

캡슐화?

- 객체지향을 정의하는 요소로 캡슐화를 말하는 이유는 데이터와 함수를 효과적으로 캡슐화하는 방법을 OO언어가 제공하기 때문이다. 이를 통해 데이터와 함수가 응집력 있게 구성된 집단을 서로 구분 짓는 선을 그을 수 있다. private, public 등이 같은 캡슐화에 쓰이는 개념이다.
- C는 헤더파일과 c파일을 이용해 완벽한 캡슐화를 이루어냈다. C++은 약화된 캡슐화를 이루어냈고, 불완전하다. 이후 나온 자바와 C#은 헤더와 구현체를 분리하는 방식을 모두 버렸고 이로인해 캡슐화는 더욱 심하게 훼손되었다. public, private, protected 키워드를 도입함으로 불완전한 캡슐화를 보완했지만 임시방편이다.
- 이때문에 객체지형은 강력한 캡슐화에 의존한다는 정의는 받아들이기 힘들다. 객체지향 프로그래밍은 프로그래머가 충분히 올바르게 행동함으로써 캡슐화된 데이터를 우회해서 사용하지 않을 거라는 믿음을 기반으로 한다.

상속?

- 객체지향 언어는 상속을 확실하게 지원했다.
- 객체지향 언어가 고안되기 훨씬 이전에도 상속과 비슷한 기법이 사용되었다. 하지만 다중상속이나 편리하지 않았기 때문에 객체지향언어가 이루어낸 바가 있다.

다형성?

- 객체지향 언어가 있기 전에 다형성을 표현할 수 있는 언어가 있었다. 하지만 다형성을 좀 더 안전하고 더욱 편리하게 사용할 수 있게 해준다.

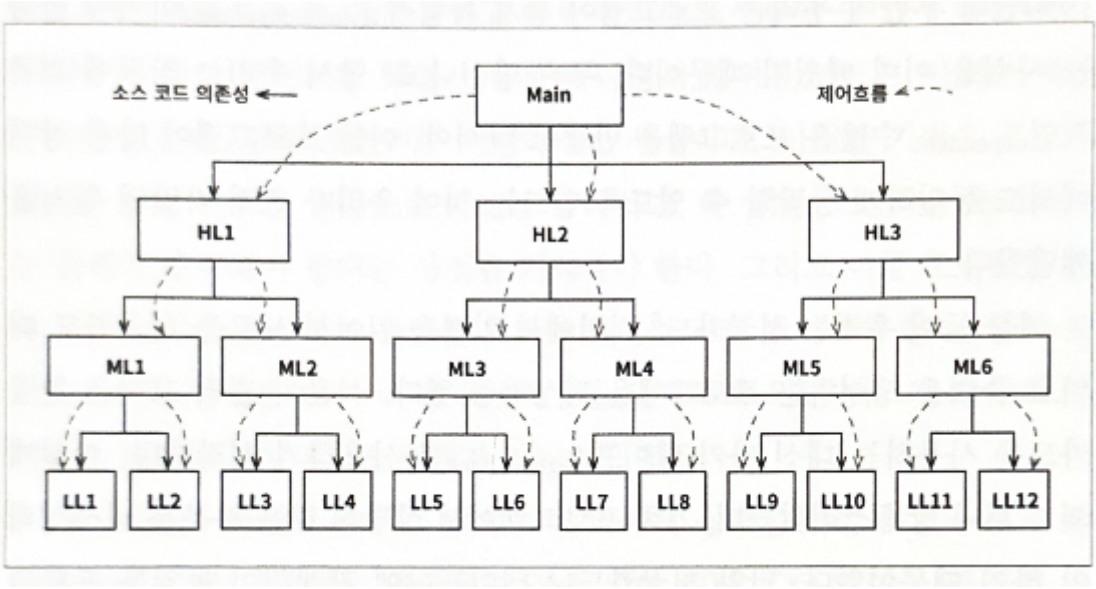
- 함수를 가리키는 포인터를 응용한 것이 다형성이다.

다형성이 가진 힘

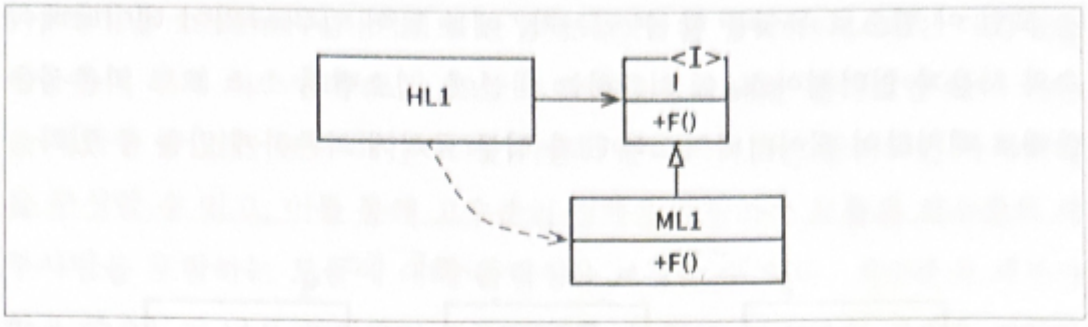
- 이전에도 소프트웨어는 장치에 독립적으로 개발되어야한다고 생각했다. 이를 플러그인 아키텍처라고 하고, 등장 이후 거의 모든 운영체제에서 구현되었다. 그럼에도 불구하고 대다수의 프로그래머는 직접 작성하는 프로그램에서는 이 개념을 확장하여 적용하지 않았다. 함수를 가리키는 포인터를 사용하면 위험을 수반하기 때문이었다.
- 객체지향의 등장으로 언제 어디서든 플러그인 아키텍처를 적용할 수 있게 되었다.

의존성 역전

- 다형성 등장 전에 소프트웨어는 메인함수는 고수준 함수를, 고수준 함수는 중간수준 함수를, 중간수준함수는 저수준 함수를 호출하는 식으로 의존성의 방향은 반드시 제어흐름을 따르게 된다. 이러한 제약조건으로인해 제어프름은 시스템의 행위에 따라 결정되며, 소스코드 의존성은 제어흐름에 따라 결정된다.



- 여기서 다형성을 사용하면 ML1과 I인터페이스 사이의 소스코드 의존성이 제어흐름과는 반대가 된다. 이를 의존성 역전이라고 부른다.



- 소스코드 사이에 인터페이스를 추가함으로써 방향을 역전시킬 수 있다. 이런 접근법을 사용하면 객체지향으로 소스코드 의존성 전부에 대해 방향을 결정할 수 있다.
- 또 이런식으로 서로 의존하지 않도록 구현해서 UI, 비즈니스로직, 데이터베이스 간의 의존성을 역전시키면 독립적으로 배포가 가능하다. 이것이 배포 독립성이고, 이를 확장해서 시스템의 모듈을 독립적으로 개발할 수 있게 되면 이것이 개발 독립성이다.

결론

객체지향이란 다형성을 이용해 전체 시스템의 모든 소스 코드 의존성에 대한 절대적인 제어 권한을 획득할 수 있는 능력이다.