**객체지향의 사실과 오해**

**-역할, 책임 협력 관점에서 본 객체지향**

**객체지향은 현실의 재현?**

객체지향 프로그래밍은 컴퓨터 상에서 현실 세계를 시뮬레이션 하기 위해 태어났다. 그러나 세계는 시뮬레이션과 달랐다. 세계를 일 대 일로 구현한다는 처음의 목표는 불가능하고 비효율적인 것으로 판명되었다. 객체지향 프로그래밍의 문제는 점차 세계의 특정 부분을 어떻게 효율적으로 도구화 할 것인가?의 문제로 바뀌어 갔다. 따라서 현재 객체지향 언어는 현실의 순수한 모방을 목표로 하지 않는다. 객체지향 프로그래밍은 현실의 문제 해결을 위해 현실의 특정 부분을 도구화 한 프로그래밍적 방법론이다.

**역할, 책임, 협력**

객체지향 프로그램의 기능은 객체들의 협력으로 동작한다. 각 객체에게는 맡은 바 역할이 있다. 객체는 서로 메세지를 주고 받으며 자신의 역할을 다 한다. 책임은 역할을 내포하고 역할은 대체 가능성을 의미한다. 한 객체가 바뀌어도 이전의 객체가 맡고 있던 역할은 다른 객체가 대체 가능해야 한다. 메시지를 통해서 일하고, 어느 때나 대체 가능한 존재라니 객체가 수동적인 존재처럼 느껴질 수도 있다. 그러나 객체는 자율적인 존재이다. 객체는 공장의 라인 조립자라기 보다는 스마트 폰으로 메세지를 주고 받으며 매출을 올리는 프리랜서 영업 사원에 가깝다. 객체가 발로 뛰든, 온라인 마케팅을 하든, 다른 업체에 다시 하청을 맡기든 객체는 맡은 바 책임을 다해 자신에게 할당된 매출을 올려야만 한다(책임). 메시지를 보낸 다른 객체는 해당 객체가 월 일 억원 이상의 매출을 올릴 거라는 사실만을 알고 있다. 그가 어떻게, 무슨 일을 해서 월에 일 억원을 파는 지는 알 수 없으며 간섭해서도 안 된다. 각 객체는 저마다의 상태와 메서드를 가지며 이러한 상태와 메서드 조합을 통해 자율적으로 맡은 일에 대한 책임을 다한다. 메세지로 전달되는 결과 외에 내부의 사정은 Private한 일들이다. 좋은 객체지향 프로그램에서 협력하는 객체들은 역할과 책임을 다한 객체들의 프라이버시를 존중한다.(캡슐화)

**객체지향은 클래스 중심이다?**

클래스 중심의 사고는 역할, 책임, 협력보다는 데이터에 종속적일 수 있다. 객체지향은 클래스 지향이 아니다. 클래스와 클래스의 관계를 먼저 깔고 거기서 파생되는 객체를 떠올리는 식의 방법론은 프로그래머의 효율적인 상상을 제한할 수 있다. 마치 화가에게 고향 풍경을 그려보라고 한 다음 그림은 반드시 유화여야 하며 사용할 수 있는 붓은 6호 뿐이라고 강제하는 것과 같다. 색체는 흰색과 초록색과 붉은 색이 조화로워야 하며 노을이 져 가는 풍경이 있어야 한다고 제한을 두는 것이다. 그렇게 하면 이발소 그림만 나올 수도 있다.

**객체의 정의 – 상태, 행동, 식별자를 지닌 실체**

싯다르타는 고정불변의 자아인 ‘나’가 존재한다는 생각은 착각이며 자아는 세계의 인과관계가 중첩된 일시적 현상이라고 역설했다. 그러나 그럼에도 불구하고 나는 생각과 감정, 기억과 현재의 위치로 나의 존재를 규정(상태, 속성, 프로퍼티)하고 이를 고정된 실체로 여긴다. 이는 구별하고 구분 지어 활용하는 인간 두뇌의 특성상 어쩔 수 없는 일이다. 객체지향 프로그램에서도 비슷한 일이 발생한다. 객체를 프로그램 흐름 안에서의 활동과 흐름만으로 기억, 구분하고 활용 가능하다면 변수나 배열을 선언해 메모리를 할당하지 않아도 되기에 이상적이다. 하지만 그런 식의 객체가 증가할 수록 프로그램의 논리적 복잡도가 기하급수적으로 올라간다. 미래에 AI 프로그래머라면 프로그램 내 객체의 모든 활동을 기억했다가 활용하면 되기에 메모리면에서 말도 안 되는 수준의 코드를 써낼 지도 모르겠다. 하지만 사람은 그렇게 할 수 없다. 이때 편의상 객체의 상태를 정의하기 위해 선언하는 속성들을 프로퍼티라고 한다. 이 프로퍼티들은 선언된 후 변하지 않기에 정적이지만 프로퍼티의 값들은 변경될 수 있기에 동적이다.

만약 한 객체가 다른 객체를 참조할 수 있다면 이는 객체와 객체가 링크로 연결되어 있다고 표현하며 다른 객체를 참조하는 객체는 참조 당하는 객체의 식별자를 알고 있어야 한다. 객체의 식별자는 객체를 구분할 수 있는 요소이다. 같은 클래스에서 파생된 정확히 동일한 프로퍼티를 갖는 두 객체가 있다 하더라도 식별자가 다르면 다른 객체(엔터티)이다. 또 시간에 따라 다른 프로퍼티를 갖더라도 식별자가 같으면 동일한 식별자로 구분할 수 있다. 책에서는 기계의 은유가 등장하는데 동일한 기종의 기계가 동일한 화면을 표시한다 할지라도 두 개의 기계가 서로 다른 기계라는 점을 이용한다. 여기서 객체의 자율성 측면에 대한 설명도 보강되는데 만약 사용자가 인터페이스를 활용해 기계를 조작한다면 기계는 사용자에게 종속된 것으로, 수동적인 것으로 여겨질 수 있다. 그러나 기계의 작동 방식과 내부 로직은 어디까지나 기계 자체에 내장된 시스템을 활용한다. 사용자가 몇 가지 선택지를 갖췄더라도 작동 방식은 기계의 내부 로직이 결정하니 기계는 자유롭다고 할 수 있다. 객체지향에서 객체의 자율성은 이러한 측면을 말한다.

이때 객체의 행동은 객체를 규정하는 중요한 요소가 된다. 클래스를 통해 객체를 규정하는 일이 바람직하지 않듯이 상태를 통해 객체를 규정하는 일도 좋지 않다. 객체의 내부 상태가 로직에 긴밀하게 얽히면 캡슐화가 저해될 수 있다. 또 객체가 상태에 근거해 생성되었으므로 타 객체와의 협력에 있어 경직된 역할만을 수행하게 될 수도 있다. 따라서 객체의 설계는 클래스가 아닌 객체를 중심으로, 타 객체와의 협력 속에서 해당 객체가 수행해야 하는 역할을 기준으로 이뤄지는 게 바람직하다.

상태, 행동, 식별자를 지닌 객체의 세계는 이상한 나라의 앨리스 속의 이상한 나라로 귀결된다. 객체 지향의 세계에서는 찻주전자와 시계, 트럼프 카드 등의 무생물이 인격체처럼 자율적으로 행동하며 협력한다(의인화). 이상한 나라는 현실을 모방하지 않는다. 외려 추상화된 현실의 일부는 목적에 맞게 변주되어 독자에게 흥미와 감동을 유발한다. 객체지향 프로그래밍의 세계도 그래야 한다.

**타입과 추상화**

객체는 세 개념으로 구성되는데 심볼, 내연, 외연이 그것이다. 심볼은 객체의 이름, 내연은 객체의 정의, 외연은 객체가 비슷한 정의의 타 객체와 관계 맺으며 발생하는 집합이다. 타입은 이러한 외연을 통해 정의된다. 이때 타입을 구분하는 것은 행위가 되어야 한다. 따라서 외연 집합은 상태가 아닌 행위를 통해 정의된다.

**명확한 책임이 애플리케이션의 미래를 결정한다.**

책임은 객체가 공유하는 유일한 정보이자 서비스가 되어야 한다. 이러한 책임은 공용 인터페이스를 구성한다. 책임의 자리는 역할이 채우고 있다. 역할은 다시 단순성, 유연성, 재사용성을 만족시키는 위치에 있어야 한다. 이를 위한 세 가지 방법론이 있다. 책임 – 주도 설계와 디자인 패턴, 테스트 – 주도 개발이 그것이다.

**메시지 추상화 정도**

일반적으로 메시지는 수신자, 메시지 이름, 인자의 조합으로 표현된다. 이때 메시지는 무엇을 할 지 말할 수 있어야 한다. 메시지가 너무 자세하고 복잡하면 객체들의 협력은 관료주의로 흘러가고 반대로 메시지가 너무 추상적이면 협력은 체계 없음과 혼란을 낳는다. 따라서 메시지의 퀄리티가 코드의 퀄리티를 좌우한다. 메시지는 단순성, 유연성, 재사용성을 만족시키기 위해 쓰여져야 한다.

**무엇을 위해 존재하는가?**

싯다르타는 무아를 내세워 실재하는 고정불변의 자아가 아닌, 상관관계 속의 피드백인 스스로를 자각하라고 설법했다. 객체 또한 마찬가지이다. 객체는 고정된 상태로 규정되어 고정된 상태로 머물다 가는 존재가 아니라 움직임을 전제로 설계되어 협력의 흐름 속에 존재해야 한다. 객체에는 속성, 메서드 등이 부여되지만 그것은 어디까지나 편의 때문이지 객체의 본질이 고정된 상태 그 자체이기 때문이 아니다.

메시지를 전달받은 객체는 메시지를 처리할 메서드를 선택하게 된다. 이때 객체지향 언어에서 객체는 동적으로 메시지를 처리한다. 이 점이 객체지향 언어와 타 언어의 차이점이다. 절차지향 언어에서는 컴파일 시에 이러한 전후관계가 미리 규정되지만 객체지향 언어에서는 이러한 선택과 실행이 런타임에 이뤄질 수 있다. 같은 일도 각자의 자율성에 따라 다르게 처리될 수 있다.(다형성) 이러한 설계는 유연함과 재사용성을 만족시킨다.

**Tell, Don’t Ask**

What-Who 사이클 또는 데메테르 법칙 등은 모두 Tell, Don’t Ask라는 문구로 표현될 수 있다. 묻지말고 시키라는 이 원칙은 메시지를 결정하기 위해 객체를 고려할 것이 아니라 메시지를 결정한 뒤에 객체를 고려하라는 의미이다. 이런 방식을 통해 송신자는 수신자의 상태를 고려하거나 상태에 관여하지 않게 된다. 송신자의 메시지를 전달받은 수신자는 어떤 식으로든 자신의 책임을 다 하기만 하면 되기에 캡슐화를 지킬 수 있다. 이렇듯 메시지를 통한 객체 간의 느슨한 결합은 결합도를 낮추고 코드의 품질을 올려준다.

**규율이 곧 자유다.**

객체의 외부로 드러난 인터페이스는 해당 객체와 소통 가능한 유일한 통로여야 한다. 외부로 드러난 인터페이스는 규율이다. 적절하게 추상화된 규율은 객체가 내부 인터페이스를 어떻게 활용하든지 관여하지 않는다. 따라서 객체의 내부 로직과 데이터에 침입할 수도 없게 된다. 이러한 객체지향의 법칙은 인터페이스와 구현을 분리해 객체의 내부구현에 대한 자유도를 보장한다.

**적절한 추상화와 높은 응집도, 낮은 결합도, 인터페이스와 구현의 분리, 캡슐화의 증진과 설계의 유연성, 단순성, 재사용성**

지금까지의 원칙들을 통해 도출할 수 있는 객체지향 프로그래밍의 장점은 위 문장으로 집약된다.

**구조와 기능**

피터 드러커는 “측정할 수 없는 것은 관리할 수도 없다.”는 격언을 남겼다. 앞서 살펴본 대로 객체를 협력관계 안에서 파악하고 정의하는 일은 중요하다. 그러나 그럼에도 전체적인 프로그램은 구조 속에 놓여야 한다. 모든 객체를 흐름 속에서 동적인 상태로 파악하는 데에는 한계가 있기 때문이다. 설계 측면에서 말하는 ‘기능’이 말 그대로 사용자가 원하는 기능을 제공하기 위한 일회적 측면이 강하다면 구조는 사용자의 변화하는 요구를 시의적절하게 반영하기 위한 큰 틀에 가깝다.

큰 관점에서의 구조와 기능 외에도 조금 다른 의미의 ‘기능’이 있다. 이는 도메인이라는 단어와 구별되는 의미로서의 기능이다. 도메인은 사용자들이 해당 프로그램을 사용할 때 갖는 심적 표상이다. 예를 들어 주식 어플리케이션을 사용하면서 사용자는 자신이 주식을 거래하고 있다고 생각하고 배달 어플리케이션을 사용하면서는 배달과 먹을 것을 떠올린다. 그 누구도 객체지향과 데이터베이스의 트랜잭션, 객체의 협력 활동을 떠올리지는 않는다. 그러니 도메인이 현실의 부분을 가져와 이용하기 좋도록 표상화 한 것이라면 ‘기능’은 보다 순수하게 프로그램이 제공하는 프로그램으로서의 로직을 말한다고 할 수 있다.

누군가가 북촌에서 광화문 광장으로 향한다면 이를 위한 기능은 전진, 전진, 좌회전, 전진, 우회전, 전진, 좌회전, 전진, 전진, 우회전으로 표현될 수 있을 것이다. 프로그램의 본질은 이것이지만 길을 설명할 때 이렇게 알려주는 사람은 없다. 북촌에서 광화문 광장에 가려면 정독도서관에서 덕성여고를 지나 대로로 나간 뒤 인사동 방면으로 가지 않고 우회전해 인왕산이 보이는 방향으로 가야 한다.

**표현적 차이(의미적 차이)**

그러나 위와 같은 설명도 경복궁 가는 길에 놓인 북촌 피냉면과 한옥으로 된 베스킨라빈스, 픽사스토어를 모두 열거할 수는 없다. 도메인은 어디까지나 현실을 생략하고 간소화한 추상화의 산물이기 때문이다. 현실과 도메인 모델과 도메인 객체 사이의 불가피한 차이를 객체지향에서는 표현적 차이라고 한다. 중요한 것은 여기서 발생하는 비효율성의 트레이드오프를 최소화하는 것이다.

이를 위해 UML을 활용할 수도 있다.