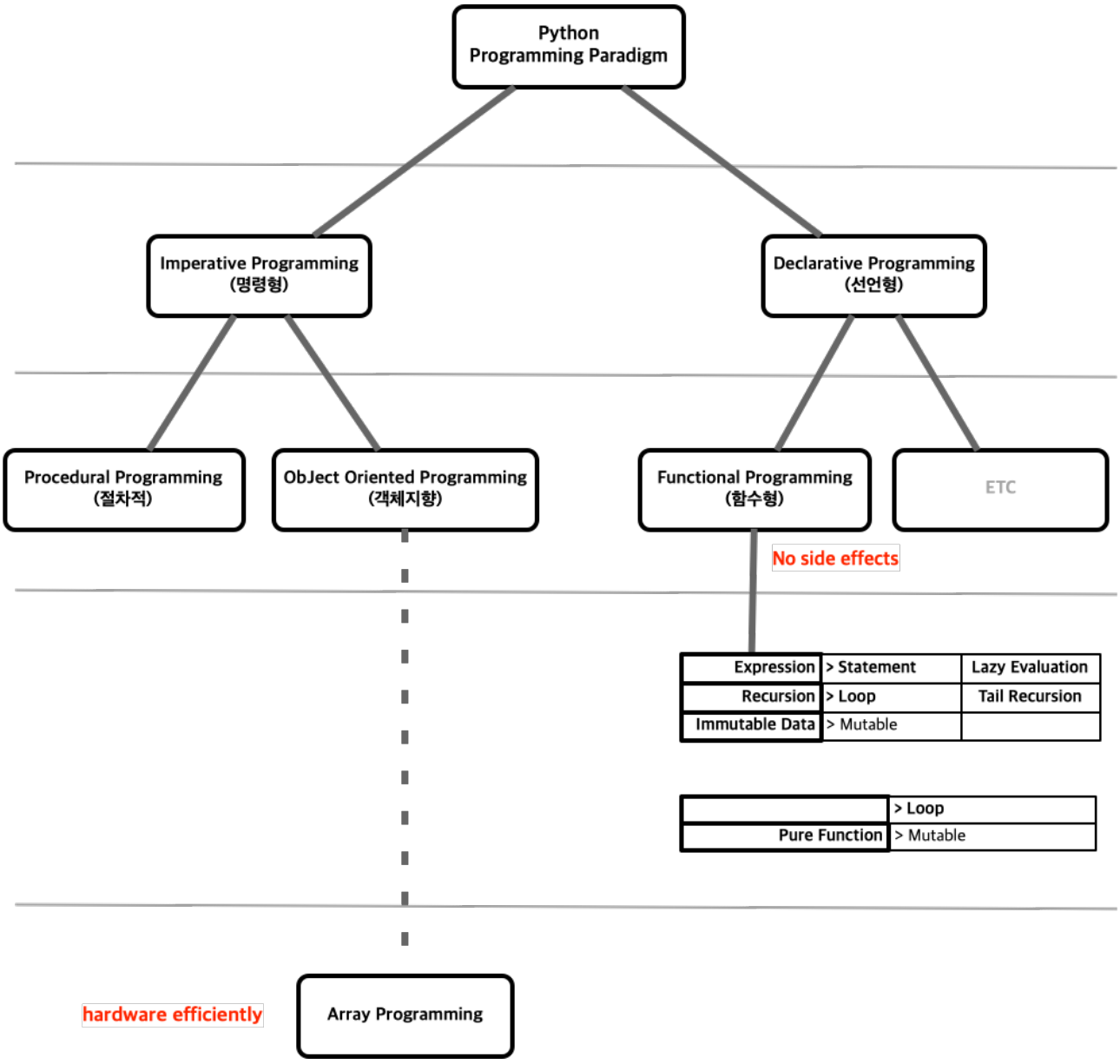
용어 정리 (*문맥에 따라 다른 의미를 가질 수 있음)

- Data : 메모리상의 연속된 비트
  ex) 01000001
- Value : 데이터에 해석이 더해진 것. 해석 없는 데이터는 아무 의미가 없음. 모든 값은 메모리상의 데이터와 연관
  ex) 01000001은 정수 65, A, ?
- Data type: 공통적인 해석을 공유하는 값의 집합
- object: 주어진 값 유형에 속하는 값이 담겨 있는 메모리상의 비트의 모음

자연과학	수학	프로그래밍	프로그래밍에서의 예
속	이론	개념	정수, 문자
종	모형	유형 또는 클래스	int, str
개체	원소	인스턴스	01000001 (65, 'A')

프로그래밍 패러다임 (방법, 스타일)



객체지향 프로그래밍 (Object-Oriented Programming, OOP)

프로그래밍 패러다임[1] > 생산성 (Money?)

- 상속, 다형성 등을 이용해서 **재사용**성 증가.
- 제약 (생성자, 소멸자, 접근제어[Public, Private, Protected] 등)을 통해 실수로 인한 취약점을 줄여 줌.
- 유지보수/테스트 용이, 협업에 유리 (SoC: Separation of Concerns)

1. 추상화 (Abstraction)

- 어떤 영역의 업무를 처리함에 있어 필요 하지 않는 정보를 제외하고 **필요로 하는 속성이나 행동을 추출하는 작업**
- **구체적인 사물들의 공통적인 특징을 파악해서 이를 하나의 개념 혹은 집합으로 다루는 수단**

2. 캡슐화 (Encapsulation)

- **실제 구현 내용을 감추는 것 (정보 은닉: information hiding)**
  - 외부 객체는 객체 내부의 구조를 알지 못하며 객체가 노출해서 제공하는 필드와 메소드 **(interface)**만 이용
- **외부의 잘못된 사용으로 인해 객체가 손상되지 않도록** 함
- 낮은 결합도를 유지

요구사항 변경에 대처하는 고전적인 설계 원리

응집도(cohesion)

- 클래스나 모듈 안의 요소들이 얼마나 밀접하게 관련되어 있는지를 나타냄

결합도(coupling)

- 어떤 기능을 실행하는 데 다른 클래스나 모듈들에 얼마나 의존적인지를 나타냄

- 결합이 많을수록 문제가 발생. 한 클래스가 변경이 발생하면 변경된 클래스의 비밀에 의존하는 다른 클래스들도 변경해야 할 가능성이 커지므로 캡슐화를 사용하여 데이터 구조에 따른 코드의 수정 범위를 캡슐 범위로 한정

접근 제한/제어 (public, private, protected)

접근 권한을 통해 제공되며 원하지 않는 (실수 등) 외부의 접근에 대해 내부의 데이터, 함수를 보호

1) https://cs.lmu.edu/~ray/notes/paradigms/

3. 다형성 (Polymorphism)

○ **서로 다른 클래스의 객체가 같은 메시지를 받았을 때 각자의 방식으로 동작하는 능력**

구분	오버로딩 [Overloading]	오버라이딩 [Overriding]
용어	Static / Compile-Time / Early binding	Dynamic / Run-Time / Late binding
특징	같은 기능의 함수에 같은 이름을 사용할수 있어서 가독성 증가 하나의 함수에 여러 매개변수 가능	상속으로 받은 함수를 그대로 사용하지 않고 새로 만들어서 사용 코드의 재사용성 증가
사용	같은 클래스 내에서 사용	다른 클래스에서 사용
함수명	동일	동일
매개변수	매개변수는 달라야 함.	매개변수는 같아야 함.
리턴	상관 없음.	같아야함

연산자 오버로딩 (Operator Overloading)

cf) 제네릭(Generic) 함수

○ 어떤 하나의 함수 (혹은 겉으로 보기에 이름이 다른 여러 다른 함수)가 여러 타입의 인자를 받고, 인자의 타입에 따라 적절한 동작을 하는 함수

○ overloading은 *static* type의 파라미터 / dispatch는 *dynamic* types에 사용

○ dispatch는 파라미터 개수에 따라 single / multi(duble 등) dispatch로 구분

EX) 기본 함수 중 len()

4. 상속 (Inheritance)

○ 단일 상속과 다중 상속

다중 상속은 강력한 상속 방법이지만 이러한 여러 가지 새로운 문제를 발생

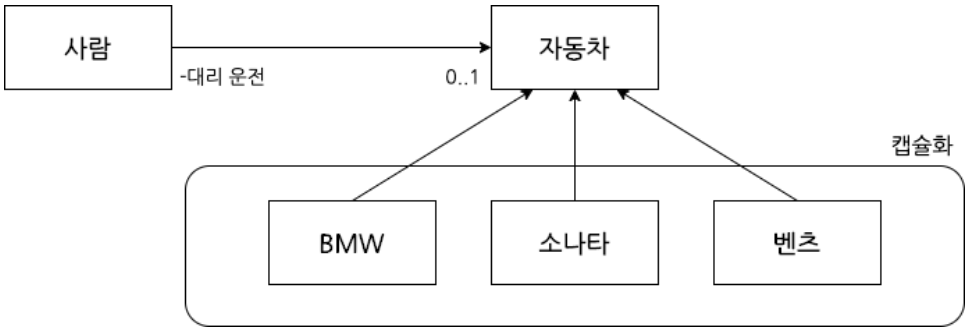
- 하나의 클래스를 간접적으로 두 번 이상 상속받을 가능성 (다이아몬드 문제).
- 다중 상속은 프로그래밍을 복잡(상속받은 여러 기초 클래스에 같은 이름의 멤버가 존재할 가능성 등)하게 만들 수 있으며, 그에 비해 실용성은 그다지 높지 않음.
- 반드시 다중 상속을 사용해야만 풀 수 있는 문제란 거의 없으므로, 될 수 있으면 사용을 자제하는 것이 좋음.

○ 일반화 관계 : “is a kind of 관계“

- 과일 -> 바나나, 사과, 배, 오렌지의 공통 개념을 **일반화(*generalization*)**
- 바나나, 사과, 배, 오렌지 -> 과일의 한 종류이므로 **특수화(*specialization*)**

○ **또 다른 캡슐화**

- 일반화 관계는 외부 세계에 **자식 클래스를 캡슐화(또는 은닉)**하는 개념으로 볼 수 있으며, 이때 캡슐화 개념은 한 클래스 안에 있는 속성 및 연산들의 캡슐화에 한정되지 않고 **일반화 관계를 통해 클래스 자체를 캡슐화**하는 것으로 확장.
- 이러한 서브 클래스 캡슐화는 외부 클라이언트가 개별적인 클래스들과 무관하게 프로그래밍할 수 있게 함



'사람' 클래스 관점에서는 캡슐화(은닉)로 인해 자동차의 종류를 볼 수 없음.
대리 운전의 경우 자동차 종류로 인해서 운전의 영향을 받지 않음

- **추상화(abstraction) : 자세함을 무시하며 필요한 부분만을 추출하는 활동**
  필요한 부분만을 표현하고 불필요한 부분을 제거하여 간결하고 이해하기 쉽게 만드는 작업. 즉, 현실에서 출발하되 불필요한 부분을 제거해가면서 사물의 본질을 드러나게 하는 과정을 말하며 객체를 모델링할 때 필요로 하는 만큼의 속성과 오퍼레이션을 추출해 내는 것. 설계 중의 클래스에 무엇을 포함시키고, 무엇을 제외시킬 지를 결정하고 나면 그 클래스의 추상화는 완료.
- **분류화(classification) : 관계있는 개체들에 대한 그룹화**
  시스템에 요구되는 데이터의 특성을 규명할 경우에는 각각의 객체들을 기술하는 것보다 묶어진 엔티티 타입을 기술하는 것이 편리하여 비슷한 엔티티를 묶는 작업
- **일반화(generalization) : 재사용을 위한 의미적 종속관계**
  연관성이 있는 2개 이상의 객체 집합을 묶어 좀 더 상위의 객체 집합을 형성하는 것. 즉 객체 사이에 유사성이 존재할 때 이 유사성을 모아 하나의 새로운 객체 타입을 정의 내리는 것. 일반화를 통해 나타나는 중요한 특성은 상위 클래스에 하위 클래스의 공통적인 속성을 표시하고, 상위 클래스의 정보가 하위 클래스에 상속된다는 것. 또한 일반화를 통해 다 계층 구조를 만들어 나갈 수 있으며, 이 경우 하위 클래스는 이 구조상 에 있는 모든 상위 클래스의 속성을 상속

○ **속성이나 기능의 재사용 (위임)**

▶ 위임 (Delegation) : 필요한 기능이 다른 클래스에서 그 기능을 제공하고 있다면 기능이 포함된 객체에게 대신 기능을 수행해 달라고 하는 것 (재사용과 유연함)

- **일반화 관계 : 암시(묵시)적**

변경의 유연함이라는 측면에서 단점
- 상위 클래스 변경의 어려움
- 클래스의 불필요한 증가
- 상속의 오용 (상속관계가 복잡할 경우, 코드 이해가 어려운 경우가 많음)

- **Composition (A owns B) / Aggregation (B is part of A) : 명시적**
- **<u>전체 기능 상속은 피하고 싶을 때</u>** 사용 (필요한 속성만 부모클래스로부터 가져와 사용하는 것)

피터 코드의 상속 규칙

○ **상속의 오용을 막기 위해** 상속의 사용을 엄격하게 제한하는 규칙

- 자식 클래스와 부모 클래스 사이는 **'역할 수행(is role played by)' 관계**가 아니어야 함
- 한 클래스의 인스턴스는 다른 서브 클래스의 객체로 변환할 필요가 절대 없어야 함
- 자식 클래스가 부모 클래스의 책임을 무시하거나 재정의하지 않고 확장만 수행
- 자식 클래스가 단지 일부 기능을 재사용할 목적으로 유틸리티 역할을 수행하는 클래스를 상속하지 않아야 함
- 자식 클래스가 '역할', '트랜잭션', '디바이스' 등을 특수화해야 함

부모 클래스 : 사람 자식 클래스 : 운전자 / 회사원

Q. 자식 클래스가 부모 클래스 사이의 '역할 수행' 관계 인가?

A. '운전자'는 어떤 순간에 '사람'이 수행하는 역할의 하나이다.
'회사원'도 사람이 어떤 순간에 수행하는 역할의 하나이다.

=> 사람과 운전자나 사람과 회사원은 상속 관계로 표현되어서는 안 되므로 규칙에 위배

EX)

-> 요리사는 사람이라는 범주를 상속해서는 안된다.
-> 요리사라는 것은 사람이라는 속성의 하나이다.

Q. 한 클래스의 인스턴스는 다른 서브 클래스의 객체로 변환할 필요가 절대 없는가?

A. '운전자'는 어떤 시점에서 '회사원'이 될 필요가 있으며 '회사원' 역시 '운전자'가 될 필요가 있다. 가령 자신이 일하는 회사로 출퇴근하는 동안에는 '운전자'로서의 역할을 수행하며, 회사에 있을 때는 '회사원'으로서의 역할을 수행.

=> 객체의 변환 작업이 필요하므로 규칙에 위배

EX)

-> 엄마라는 객체는 누군가의 딸이라는 객체가 될 수 있다.
-> 이런 속성 변경이 잦을 경우에는 객체의 변환 작업이 필요해서 해서는 안된다.

Q. 자식 클래스가 '역할', 트랜잭션', '디바이스' 등을 특수화 하는가?

A.

=> 슈퍼 클래스가 역할, 트랜잭션, 디바이스를 표현하지 않았으므로 규칙에 위배

따라서, 피터 코드의 규칙에 따라 위와 같은 관계는 상속을 사용하지 않고 집약(혹은 연관) 관계를 사용하는 편이 좋다.

S.O.L.I.D : 객체지향 설계 원칙

예측하지 못한 요구사항/변경사항에 유연하고 확장성 있도록 시스템 구조를 설계 가능한 한 영향을 받는 부분을 줄임

1. 단일 책임 원칙 (SRP-Single Responsibility Principle)

- 단 하나의 책임만을 가지는 원칙(클래스를 수정할 이유가 오직 하나이어야 함).
  객체지향 뿐만 아니라 절차적 프로그래밍 기법에도 적용되는 개념
- **책임분리** : 모든 코드를 테스트하는 문제를 해결하려면 한 클래스에 너무 많은 책임을 부여하지 말고 단 하나의 책임만 수행하도록 해 변경 사유가 될 수 있는 것을 하나로 만듦

> 책임이 많이 질수록 클래스 내부에서 서로 다른 역할을 수행하는 코드끼리 강하게 결합될 가능성이 높아짐
> - 응집도가 높으면 관련 기능이 한 곳에 모여있게 되는데, 이는 재사용과 유지 보수가 쉬워짐
> - 결합도가 높은 시스템의 한 부분이 변경이 되면 이에 연관된 부분들도 같이 변경하거나 회귀 테스트(시스템에 변경이 발생할 때 기존의 기능에 영향을 주는지를 평가하는 테스트)를 실행해야 함. 변경하려는 부분을 독립적으로 떼어놓기 어렵기 때문에 재사용성이 낮으며 이해하기도 쉽지 않음

2. 개방-폐쇄 원칙 (OCP : Open-Closed Principle)

- 기존의 코드를 변경하지 않으면서 기능을 추가할 수 있도록 설계가 되어야 함. 확장에는 열려있어야 하고, 변경에는 닫혀있어야 함

3. 리스코프 치환 원칙 (LSP-Liskov Substitution Principle)

- 자식 클래스는 최소한 자신의 부모 클래스에서 가능한 행위는 수행할 수 있어야 함 (부모 클래스와 자식 클래스 사이는 행위가 일관되어야 함)
- LSP를 만족하면 부모 클래스의 인스턴스 대신에 자식 클래스의 인스턴스로 대체해도 의미는 변하지 않음
- 피터 코드의 상속 규칙 중에 “서브(자식) 클래스가 슈퍼(부모) 클래스의 책임을 무시하거나 재정의하지 않고 확장만 수행한다“라는 규칙과 슈퍼 클래스의 메서드를 오버라이드 하지 않는 것과 같은 의미로 해석할 수 있으며, 피터 코드의 상속 규칙을 지키는 것은 곧 LSP를 만족시키는 하나의 방법에 해당

4. 의존 역전 원칙 (DIP-Dependency Inversion Principle)

○ 부모 클래스는 자식 클래스의 구현에 의존해서는 안됨. 자식 클래스 코드 변경 또는 자식 클래스 변경시, 부모 클래스 코드를 변경해야 하는 상황을 만들면 안됨

○ 자식 클래스에서 부모 클래스 수준에서 정의한 추상 타입에 의존할 필요가 있음

- 의존 관계를 맺을 때 변화하기 쉬운 것(구체적인 방식, 사물 등과 같은 것) 또는 자주 변화하는 것보다는 변화하기 어려운 것(정책, 전략과 같은 큰 흐름, 개념 같은 추상적인 것), 거의 변화가 없는 것에 의존하라는 원칙

5. 인터페이스 분리 원칙 (ISP-Interface Segregation Principle)

○ **클래스에서 사용하지 않는(상관없는) 메서드는 분리**

○ 클라이언트 자신이 이용하지 않는 기능에는 영향을 받지 않아야 함. 즉, 인터페이스를 클라이언트에 특화되도록 분리시키는 설계 원칙

○ 클라이언트 자신이 사용하지 않는 메서드에 생긴 변화로 인한 영향을 받지 않게 됨

○ SRP와 ISP 사이의 관계

○ 어떤 클래스가 단일 책임을 수행하지 않고 여러 책임을 수행하게 되면 방대한 메서드, 클래스, 인터페이스가 될 가능성이 커짐

-> 단일 책임을 갖는 여러 클래스로 분할SRP하고 각자의 인터페이스를 제공해야 함

Q. ISP는 SRP를 만족하면 성립되는가?

A. 반드시 그렇다고는 볼 수 없다. 게시판을 예로 들자면,