

OO에서 재사용성을 중요시하는 이유



유명한 격언 : ‘바퀴를 다시 발명하지 마라’

- 바퀴는 이미 동작과 상태가 명확한 물체
 - 설계, 구현, 테스트까지 모두 마친 물체
 - 바퀴만 전문적으로 제조 혹은 판매하는 업체가 있음
- 이걸 그냥 사다가 다른 유용한 물체를 만들자
 - 예 : 자전거, 자동차
 - 그 바퀴를 재발명하려고 시간 낭비할 필요가 없음

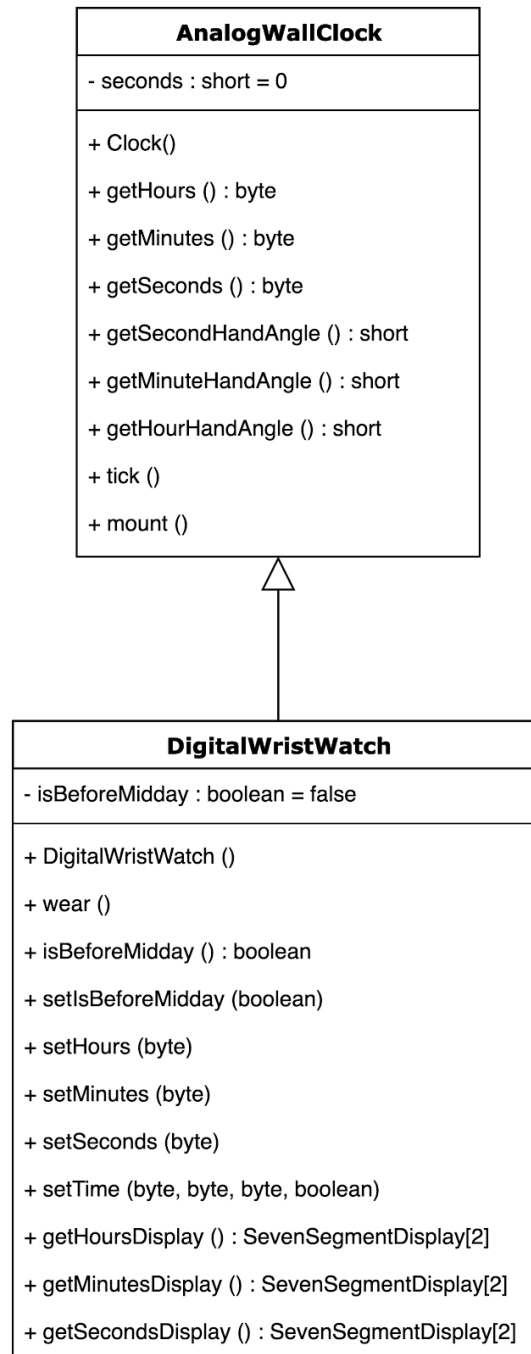
클래스를 재사용하면 좋은 점

1. 설계와 코딩에 드는 시간을 절약

- 설계 및 코딩을 다시 할 필요가 없음
- OO 외의 프로그래밍에도 적용되는 올바른 원칙
- 그러나 실전에서 100% 적용은 불가
 - 프로그램이 미래에 어떻게 변할지 완전히 예측 불가
 - 재사용성에 눈이 멀어 잘못된 바퀴를 장착할 수도 있음.

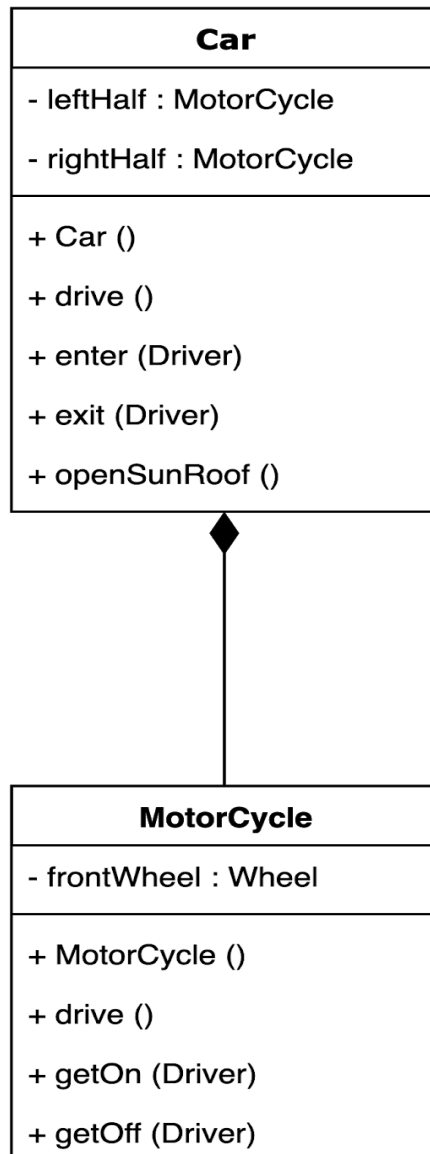
	
올바른 재사용성	잘못된 재사용성

- 다음은 잘못된 재사용성의 예시이다



-
- 벽에도 걸고, 손목에도 찰 수 있는 벽시계? 손목시계?

- 다음도 마찬가지다.



-
- 운전자가 두명이 되는? 아니 세명? 이 되는 자동차
- 앞바퀴가 따로 회전하는 자동차?

2. 테스트에 걸리는 시간을 절약

- 이미 테스트까지 끝낸 클래스를 다시 테스트할 필요가 없음
- 상속 시 부모 클래스는 이미 테스트가 끝남
- 이때 부모 클래스를 테스트할 필요가 없다 말하는 사람도 존재
- 하지만 실제로는 그렇지 않은 경우가 빈번
 - i. 새로운 방법으로 부모클래스 사용
 - 1. 자식 클래스에서 부모 클래스를 다른 방식으로 사용할 수 있음
 - 2. 이때 새로운 버그를 찾을 수도
 - ii. 부모 클래스를 변경
 - 1. 사람은 구체적인 것을 먼저 본 뒤 추상화를 함.
 - 2. 새로운 자식 클래스가 상속받을 수 있도록 수정할 수 있음

3. 관리 비용을 절약

- 코드 중복이 없음
 - i. 따라서 한 곳만 고치고 다른 곳을 실수로 안 고칠 가능성이 없음

- 관련된 코드가 모두 한 파일 안에 있음
 - i. 따라서 그 파일을 열어 모든 로직 및 데이터 파악 가능
 - ii. 단, 재사용성을 위해 클래스를 잘게 나누다 보면 파일 수가 많아짐

위의 장점들은 역시 주관성이 들어간다. 따라서 “재사용성” 과 “유지 / 관리” 의 밸런스를 잘 유지하는 게 중요

OO 모델링 실력 높이는 법

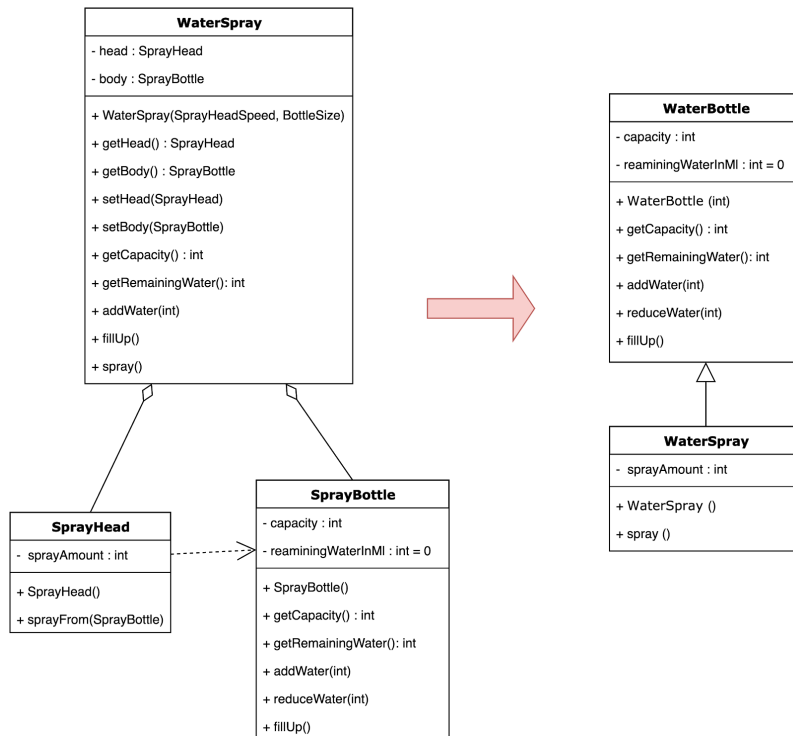
요약 : 주관적이다.

상속 vs 컴포지션 선택 시 4가지 기준

거의 모두가 동의하는 한 가지 객관적인 가이드는 있다.

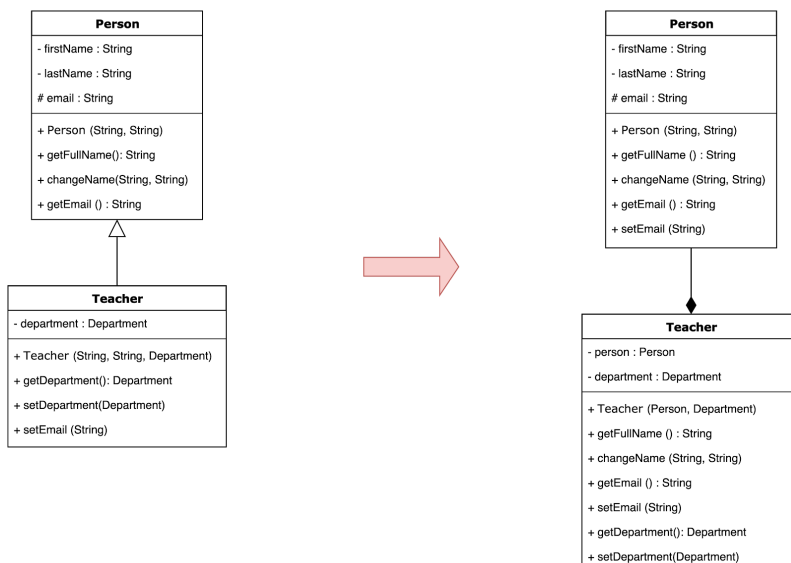
상속 vs 컴포지션

- 상속과 컴포지션 모두 **재사용성이 목적**
- 가능 / 불가능의 측면에서만 보면 많은 경우에 둘 다 사용 가능
- 컴포지션을 상속으로 바꾼 예



○ 물론 옳은 방법인지는 모르겠다.

- 상속을 컴포지션으로 바꾼 예



- 따라서 둘 중 하나를 고를 어떤 **원칙**이 필요(다음 장에서)
상속 vs 컴포지션 : 메모리

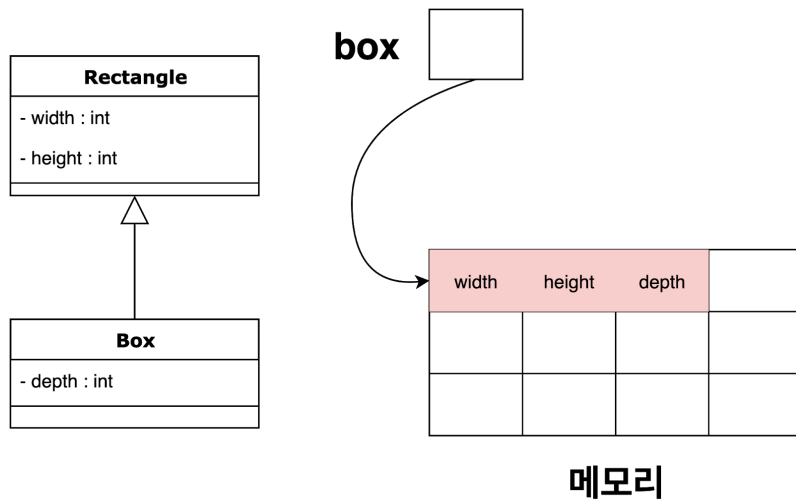
언제 무엇을 사용할지는 다음의 경우로 나눠 생각해보자

1. 기계상의 차이 때문에 하나를 골라야 할 때
 2. 용도 때문에 상속을 고를 수밖에 없을 때
 3. 관리의 효율성을 고려할 때
 4. 그 외 일반적인 상황
- 1~3 은 특수한 경우.

1.

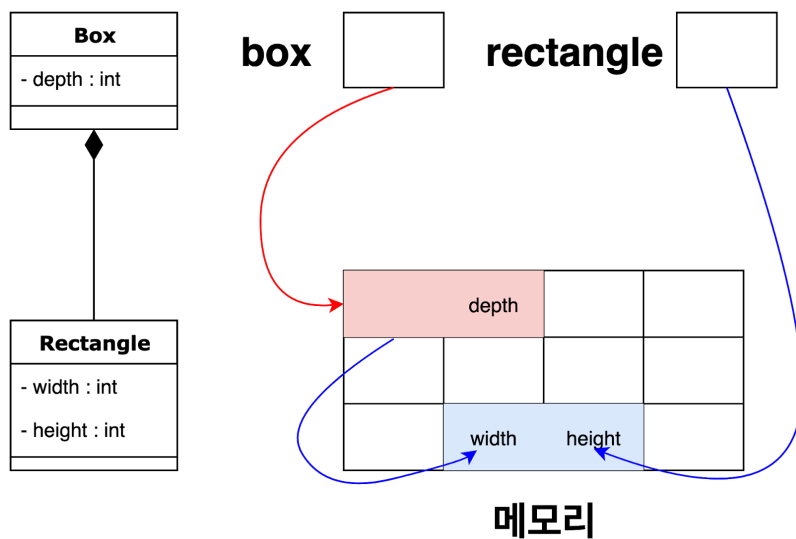
a. 상속과 메모리

i. 개체 생성 시, 메모리가 **하나의 덩어리**



b. 컴포지션과 메모리

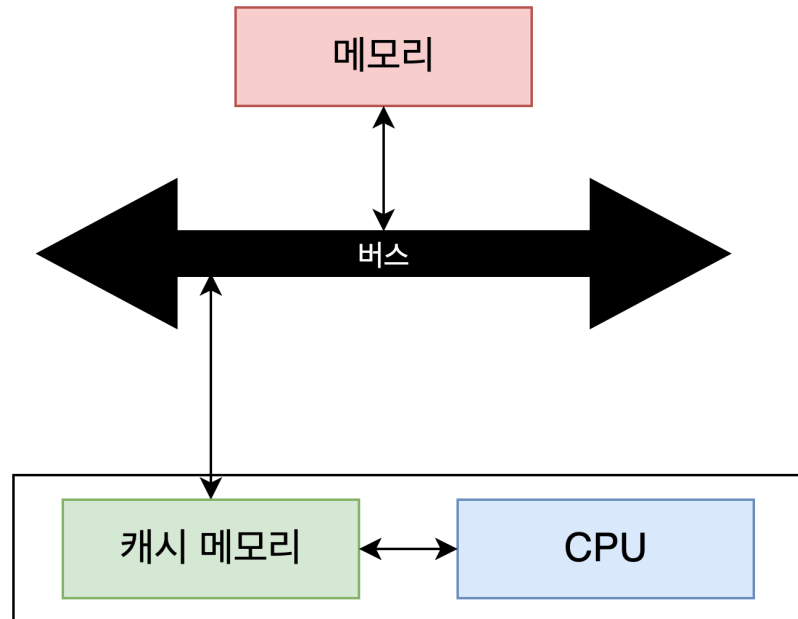
i. 개체 생성 시, 메모리가 **여러 덩어리**



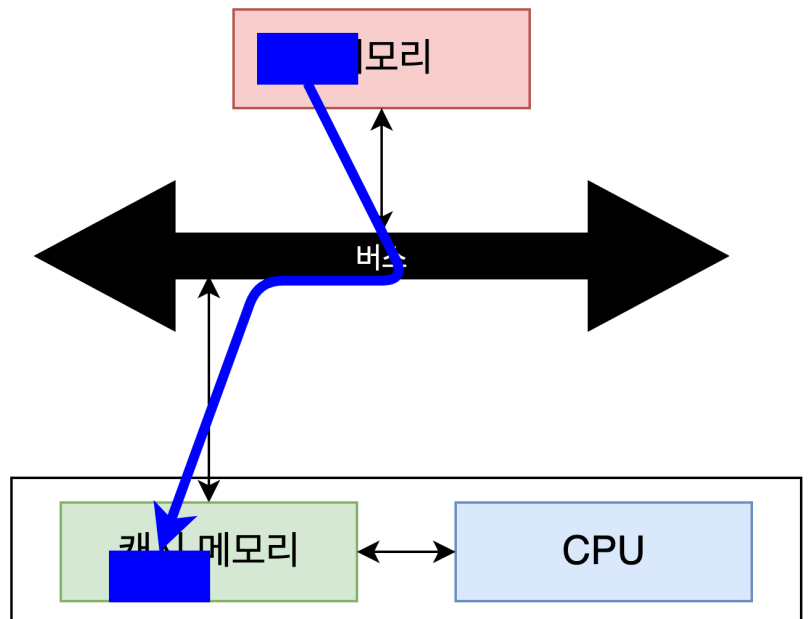
c. 이러한 메모리 상의 차이는 실행 성능에 영향을 미침

- 개체가 “컴포지션 모델” 에서와 같이 메모리가 여러 덩어리로 나누어져 있는게 안좋은 경우가 있다.

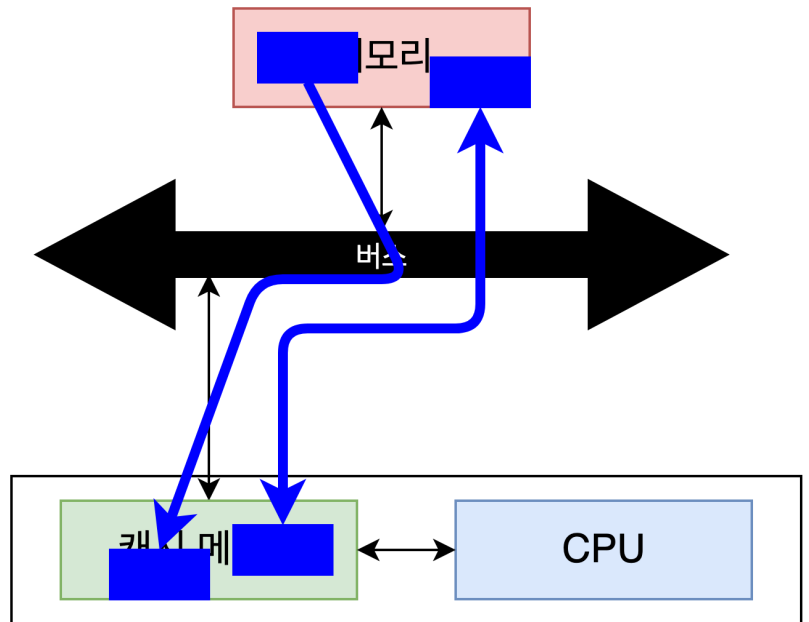
1. 프로그램 실행 중 첫 번째 병목점, CPU와 메모리 사이의 데이터 전송



- a. 캐시 메모리의 경우 메모리를 블록 단위로 읽어 온다.
b. 상속 모델의 개체의 경우, 개체가 **한 번**에 캐시 메모리에 들어갈 가능성이 높다.



- c. 컴포지션 모델의 개체의 경우, 1 + 개체 내 부품 수 만큼 캐시 메모리로 로딩할 가능성이 높다.



- d. mit 에서 최적화 공개 강의 참고하면, 행렬 연산의 경우, 캐시에 맞게 어떻게 최적화 하냐에 따라 성능이 만배 차이가 나기도 함.ㅇㅇㅇ

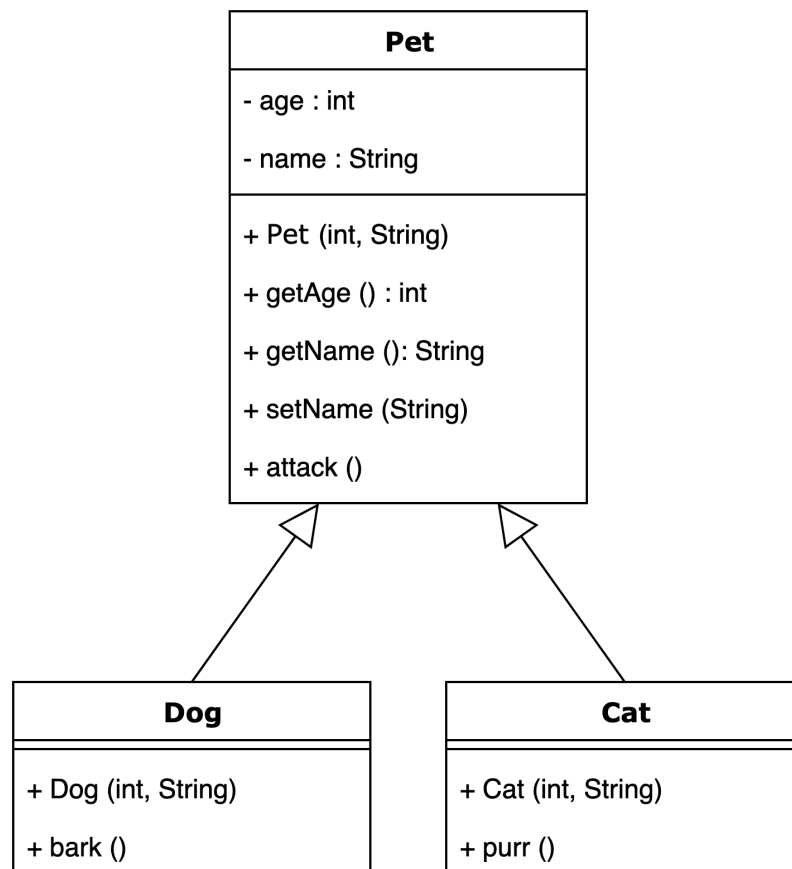
2. 프로그램 실행 중 두번째 병목점

- a. 새로운 메모리 할당(new)과 해제(delete, release)
 - b. 프로그래밍 언어에 따라 이 둘 중에 특히 느린 것이 있음
 - c. 상속 모델은 메모리 할당과 해제가 딱 한 번씩
 - d. 컴포지션 모델은 한 번 + 부품 수만큼씩
- d. 따라서 성능을 중요시 한다면, 컴포지션 보다는 상속 모델쪽을 선택하는 것이 좀 더 좋은 방법이다.

상속 vs 컴포지션 : 다형성

2. 용도 때문에 상속을 고를 수밖에 없을 때

- a. 모든 애완동물에게 동시에(Teddy, 정확히 워딩을 하자면 일괄적으로) 명령을 내리고 싶다. (일단, 다형성) 다형성은 상속 없이 돌지 않는다.



b.

```
ArrayList<Pet> pets = new ArrayList<Pet>();

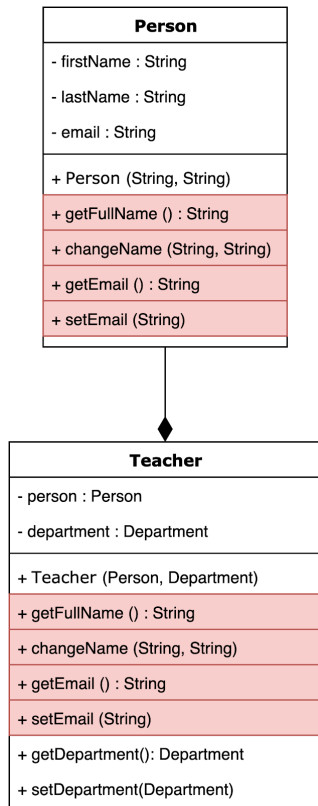
pets.add(new Dog(3, "Puppy"));
// ...
pets.add(new Cat(1, "Nabi"));

for (Pet pet : pets) {
    pet.attack();
}
```

c.

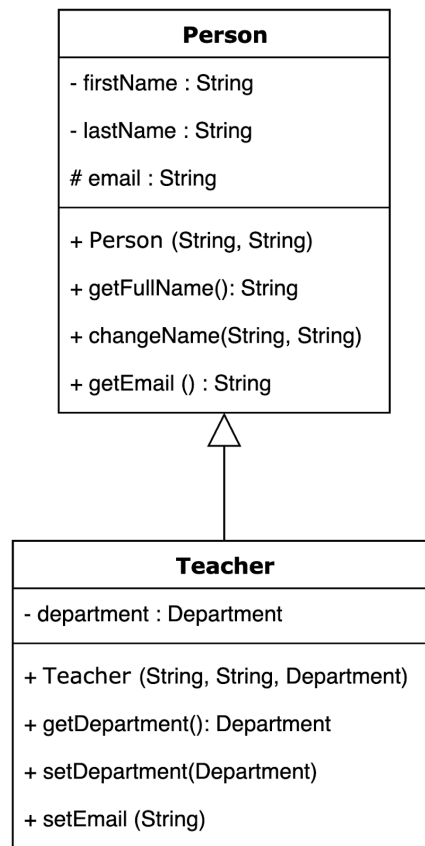
상속 vs 컴포지션 : 유지보수

3. 관리의 효율성을 고려할 때
 - a. 먼저 상속이 더 나은 예
 - i. Person-Teacher 컴포지션 모델



Person 의 메서드를 호출하는 메서드 필요. 코드의 중복은 피했지만, 시그내처의 중복이 발생

ii. Person-Teacher 상속 모델



부모의 메서드를 Teacher 에서 또 만들 필요가 없다. 즉, 자식 개체 상에서 부모의 메서드 호출 가능

b. 반대로 상속을 사용하면 관리하기 **불편한** 경우

i. 깊은 상속 관계

1. 부모 클래스를 바꾸면 그 아래 클래스도 모두 바뀜
2. 해당 자식 클래스에서 문제가 없는지 모두 확인해야 함
3. 물론 컴포지션도 비슷한 문제가 있음.
 - a. 상속보다는 덜함
 - b. 컴포지션은 상속보다 조립성을 좀 더 강조했기 때문
4. 나중에 배울 **인터페이스**와 **다형성**이 이런 문제를 조금 완화

상속 vs 컴포지션 : 일반적인 경우

4. 일반적인 상황

상식적으로 생각할 것, 그래야 모든 사람이 서로 이상한 짓을 안 함.

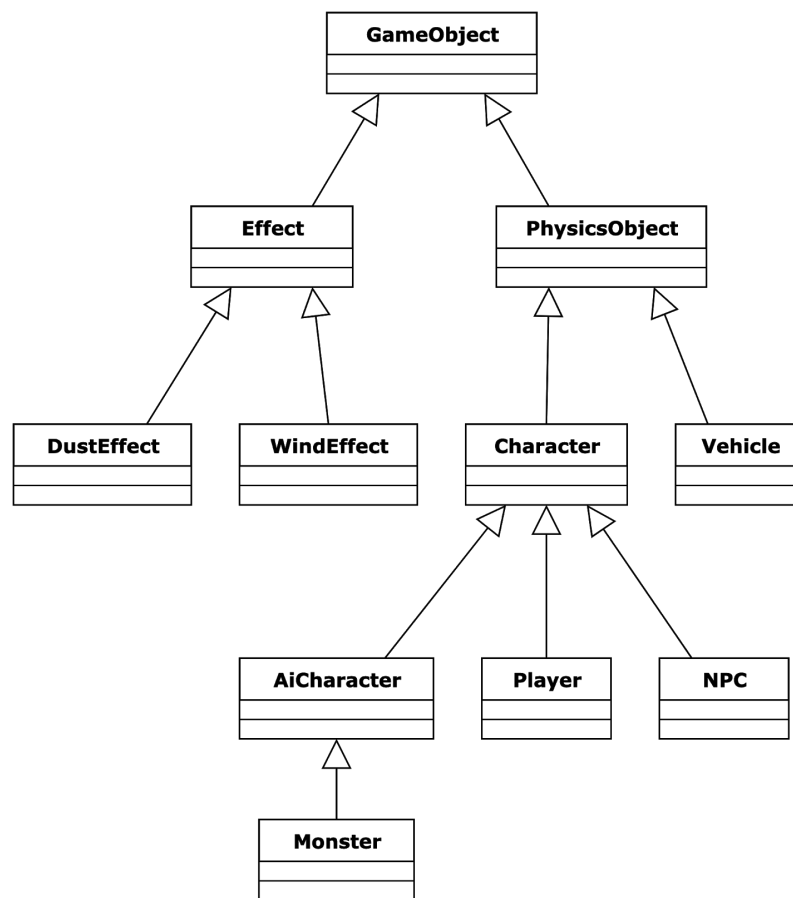
- 즉, has-a 와 is-a 관계에 충실히 하자.
- 같은 데스크톱을 구현하더라도 어떤 데스크톱을 구현하냐에 따라 다른 결과를 가진다.

컴포지션	상속
	
모니터와 본체가 분리	모니터와 본체가 하나

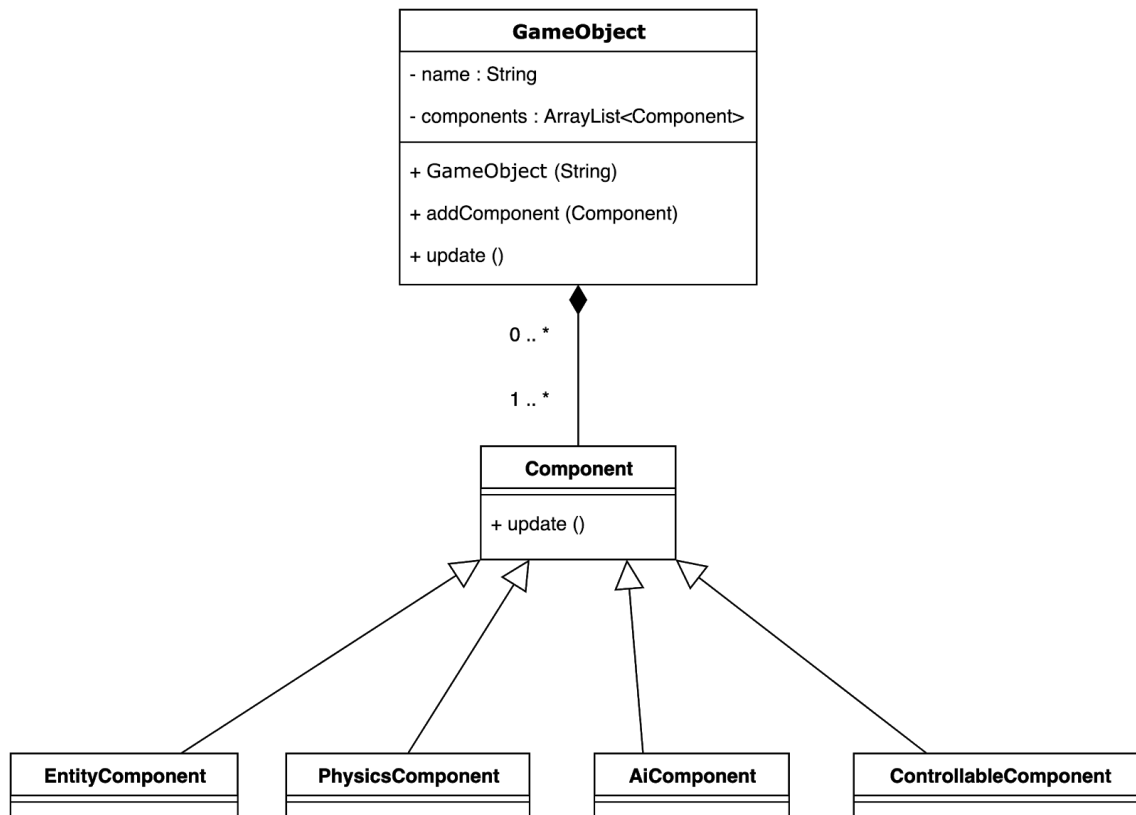
상속과 잦은 클래스 변경

엔티티 컴포넌트 시스템

- 줄여서 ECS
- 프로그래머가 컴포지션을 선호하는 또 다른 예
- **코드 변경 없이 자유롭게 개체를 만들 수 있도록** 하는 게 목적
- 특히 게임 업계에서 많이 사용 (예 : Unity3D 의 게임 오브젝트)
- 예시
 - 게임 엔진을 개발. 게임 내 개체 (이펙트, NPC, 플레이어)를 표현하는 클래스는 다음과 같다.



- 그런데 갑자기 기획자가 NPC 가 자유롭게 움직이는게 재밌을 거 같다며 이를 요청하였다 NPC 클래스는 어떤 클래스를 상속 받아야 할까?
 - AiCharacter 를 상속 받는다.
- 그런데 구현해보니 기획자가 테스트해보고 재미없다고 다시 원래 NPC 로 바꾸자고 한다면?
 - 그럼 되돌리고 다시 컴파일해야지..
- 그럼 재컴파일 없이 게임 기획자가 원하는 대로 개체를 조립할 수 없을까?
 - 하는 생각 에서 나온 게 **엔티티 컴포넌트** 시스템. 이름에서도 알 수 있듯이 컴포지션이다. 예시 다이어그램은 다음과 같다.



- 실제 플레이어, NPC 등의 클래스는 사라졌다. 대신 컴포넌트들을 조립해서 플레이어, NPC 등을 만든다.
- EntityComponent 에는 위치 정보
- PhysicsComponent 에는 물리 정보(질량, 속도 등)
 - 위 두개의 Component 를 GameObject 에 추가하면 NPC 가 된다.
 - Component::update() 메서드는 각 클래스들이 다른 동작으로 바뀜. (이게 바로 다형성, 곧 배움)
- 다음 예시 코드를 보자.

```

public class GameObject {
    private String name;
    private ArrayList<Component> components = new ArrayList<Component>();

    public GameObject(String name) {
        this.name = name;
    }

    public void addComponent(Component component) {
        this.components.add(component);
    }

    public void update() {
        for (Component component : this.components)
            component.update();
    }
}

```



```
// 플레이어 생성 예시
```

```
GameObject player = new GameObject("player");

player.addComponent(new EntityComponent());
player.addComponent(new PhysicsComponent());
player.addComponent(new ControllableComponent());

player.update();
```

```
// 한 자리에 서 있는 NPC
```

```
GameObject npc_0 = new GameObject("Teddy");

npc_0.addComponent(new EntityComponent());
npc_0.addComponent(new PhysicsComponent());

npc_0.update();
```

```
// 자기 맘대로 돌아다니는 NPC
```

```
GameObject npc_1 = new GameObject("Teddy Heo");

npc_1.addComponent(new EntityComponent());
npc_1.addComponent(new PhysicsComponent());
npc_1.addComponent(new AiComponent());

npc_1.update();
```

- 위의 플레이어의 경우
 - 게임 패드로 조종이 가능
 - 물리 충돌도 가능하다.
- npc_0 의 경우
 - 조종은 불가능
 - 물리 충돌 가능
- npc_1 의 경우
 - 조종은 불가능
 - 물리 충돌 가능
 - 알아서 움직임
- 이후 기획자가 사용하는 툴에서 컴포넌트 목록을 세팅한 뒤 파일로 저장하면 코드에서 읽어서 개체 생성 가능
 - 즉, 재컴파일이 필요 없다.