게임엔진

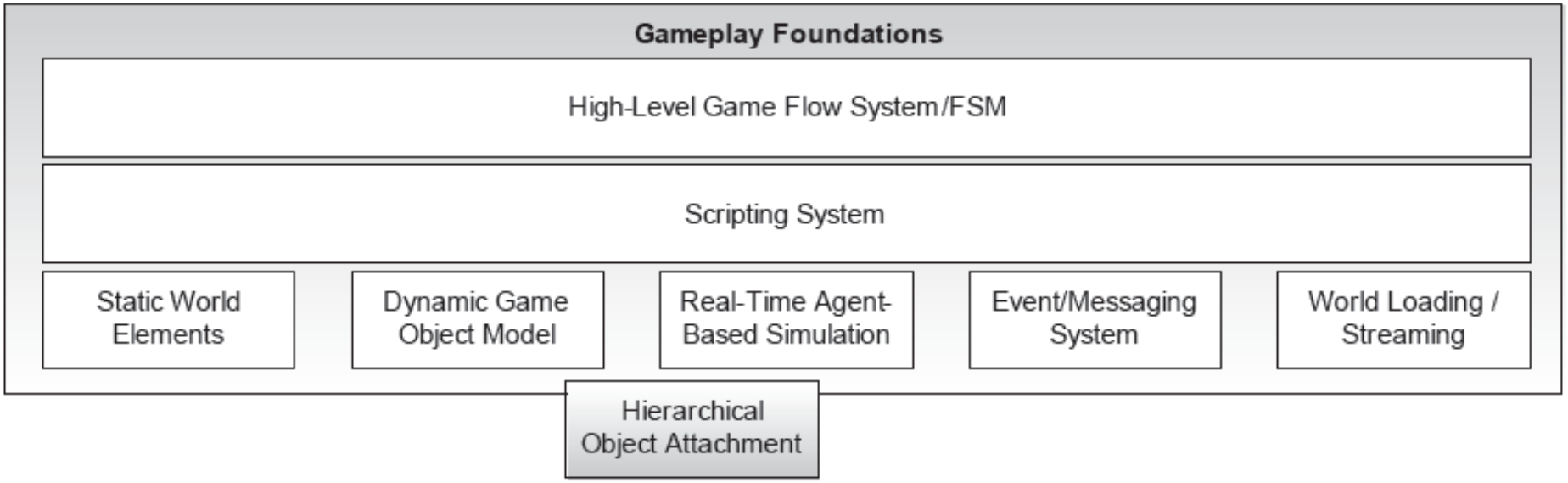
제13강 런타임 게임 플레이 기반 시스템

학습 내용

- 게임 플레이 기반 시스템
- 런타임 객체 모델 구조

게임 플레이 기반 시스템

- 게임 메카닉을 만드는데 기반으로 쓰이는 여러 런타임 소프트웨어 컴포넌트
- 게임엔진의 최상위 부분



static background geometry, like buildings, roads, terrain (often a special case), etc.;
dynamic rigid bodies, such as rocks, soda cans, chairs, etc.;
player characters (PC);
non-player characters (NPC);
weapons;
projectiles;
vehicles;
lights (which may be present in the dynamic scene at run time, or only used for static lighting offline);
cameras;

– Python
http://www.python.org

– Lua
http://www.lua.org

– GameMonkey
http://www.somedude.net/gamemonkey

– AngelScript
http://www.angelcode.com/angelscript

- **런타임 객체 모델**
  - 월드에디터를 통해서 인지하는 가상의 객체 모델을 실제로 구현하는 부분
- **레벨관리 및 스트리밍**
  - 가상 월드 콘텐츠를 불러오고 내리는 시스템
  - 빠른 메모리 스트리밍으로 심리스 월드를 구현
- **실시간 객체 모델 업데이트**
  - 모든 객체들의 주기적 업데이트
  - 이 부분에서 게임 엔진의 온갖 시스템들이 합쳐져 유기적인 전체 시스템을 이룸.
- **메시지와 이벤트 처리**
  - 객체들 간의 통신 - 대개 추상적 메시지 시스템을 통함.
  - 이벤트 시스템이라고도 함 - 게임 월드 상태의 변화에 따른 이벤트를 메시지로 전달
- **스크립트 시스템**
- **목표 및 게임 흐름 관리**
  - 챕터, 레벨, 월드 사이의 이동
  - 플레이어의 목표 성취의 기록

런타임 객체 모델

- **동적으로 게임 객체를 생성하고 파괴하기**
  - 게임 플레이 중 생성되고 소멸되는 동적 요소의 구현
    - 체력 아이템, 폭발, 새로운 적의 출현
- **로우레벨 엔진 시스템과 연동**
  - 게임 객체의 렌더링, 애니메이션, 물리시뮬레이션을 위해 하위 엔진 시스템과 연계가 필요
- **객체 행동 실시간 시뮬레이션**
  - 모든 게임 객체들의 상태를 동적으로 업데이트해야 함.
  - 경우에 따라 정해진 순서에 따라 업데이트 필요
    - 객체간의 의존성, 객체의 하위 엔진 시스템에 대한 의존성, 하위 시스템 간 상호 의존성
- **새로운 객체 타입을 정의할 수 있는 기능**
  - 월드 에디터에서 새로운 객체 타입을 쉽게 정의하고 추가하는 기능
  - 대부분의 엔진은 프로그래머의 손을 거쳐야 함.

- **고유 객체 식별자 ID**
  - 숫자, 또는 문자

- **게임 객체 질의**
  - 게임 월드 내의 객체들을 찾을 수 있는 방법(식별자들 통해)
  - 특정 조건을 만족하는 객체의 탐색(ex. 플레이어 캐릭터 주위 20미터 안의 모든 적을 탐색)

- **게임 객체 참조(Reference)**
  - 인스턴스의 포인터, 핸들, 스마트 포인터 등

- **유한 상태 기계 지원**
  - 게임 객체를 FSM으로 모델링

- **네트워크 복제(Replication)**
  - 네트워크 게임에서 PC 한 대만 게임 객체를 소유하고 관리
  - 다른 PC에서는 복제된 객체가 존재

- **게임 Save 와 Load**
  - 게임 상태의 저장, 게임 객체들의 serialization

런타임 객체 모델 구조

- **런타임 객체 모델은 월드 에디터에서 보이는 객체 타입과 속성, 행동을 충실하게 재현해야 함.**

- **객체 중심적(Object-centric) 구조**
  - 툴 상의 게임 객체 ←→ 클래스 인스턴스 하나, 또는 여러 개의 인스턴스 집합
  - 객체 = 속성 + 행동
  - 게임 월드는 다양한 게임 객체들의 집합

- **속성 중심적(Property-centric) 구조**
  - 게임 객체들은 고유의 ID로만 표현
  - 속성은 테이블에 담겨있고, ID를 통해서 속성에 액세스
  - 속성들은 하드코딩된 클래스의 인스턴스로 구현
  - 게임 객체 행동 - 객체를 이루는 속성들의 집합에 의해 간접적으로 정의
    - 'health' 속성 - 객체가 손상을입거나 체력을 잃을 수 있고, 죽을 수도 있음.
    - 'MeshInstance' 속성 - 삼각형 메쉬로 3D 렌더링할수 있는 객체

객체 중심 구조 사례

- C로 구현한 단순한 객체 기반 모델: 하이드로 썬더
  - type
    - 보트 : 플레이어 또는 AI 가 컨트롤, 떠있는 푸른/붉은 가속 아이콘, 애니메이션되는 배경 객체
    - 물 표면, 폭포, 파티클 효과
    - 레이스 트랙 섹터 - 2차원 다각형 영역
    - 정적 기하 형상, 2D HUD 요소

```
struct WorldOb_s
{
  Orient_t   m_transform;     /* position/rotation */

  Mesh3d*    m_pMesh;         /* 3D mesh */
  /* ... */
  void*      m_pUserData;     /* custom state */

  void       (*m_pUpdate)();  /* polymorphic update */
  void       (*m_pDraw)();    /* polymorphic draw */
};
typedef struct WorldOb_s WorldOb_t;
```

공통속성

고유 속성

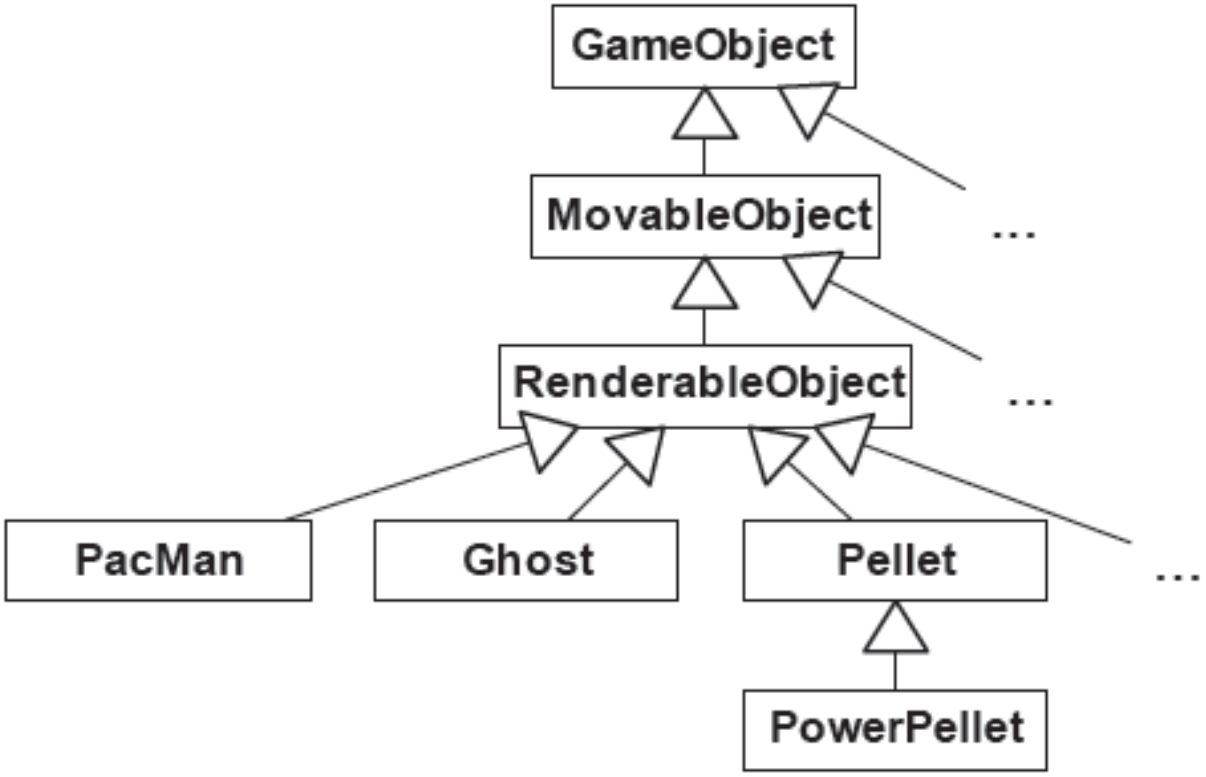
```
struct WorldOb_s
{
  Orient_t  m_transform;     /* position/rotation */

  Mesh3d*   m_pMesh;         /* 3D mesh */
  /* ... */
  void*     m_pUserData;     /* custom state */

  void      (*m_pUpdate)();  /* polymorphic update */
  void      (*m_pDraw)();    /* polymorphic draw */
};
typedef struct WorldOb_s WorldOb_t;
```

공통속성

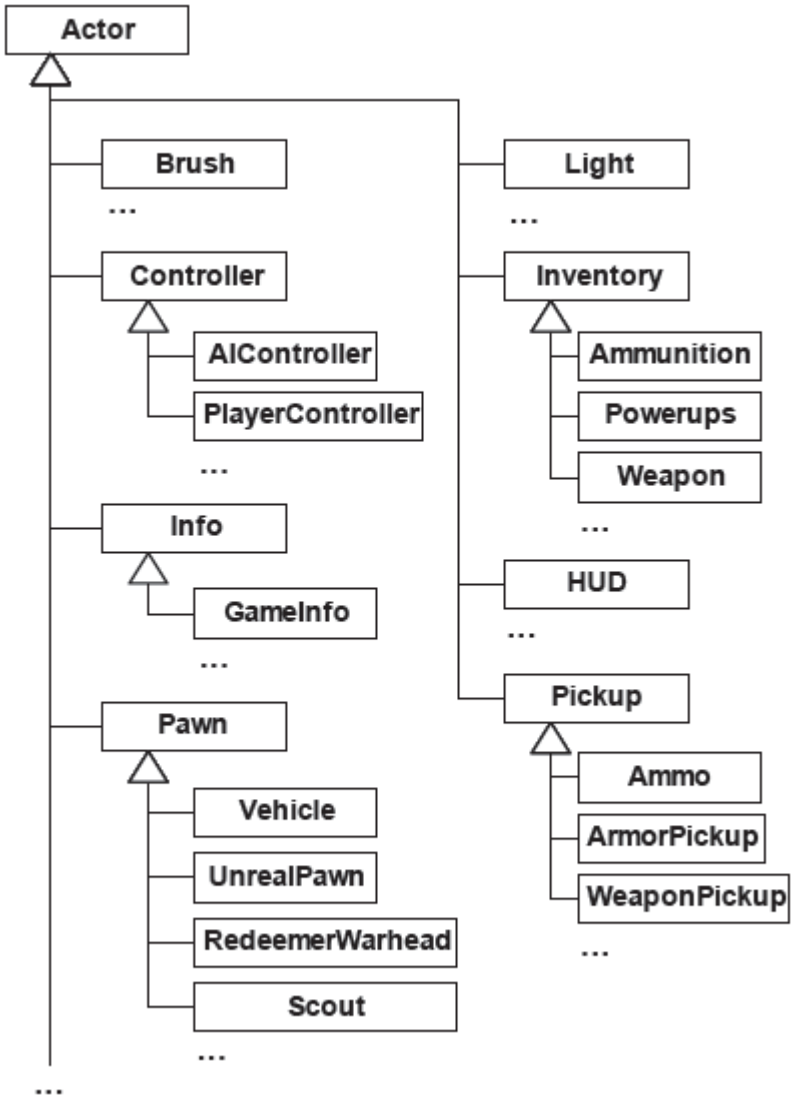
고유 속성

거대 단일 클래스 계층: 팩맨

- 게임 객체 타입을 분류하는데 생물 분류학과 비슷한 분류법을 사용함.
- 클래스 계층이 커지면서 구조는 점점 깊어지는 동시에 넓어짐.
- 하나의 공통 베이스 클래스를 거의 모든 게임 객체들이 상속.



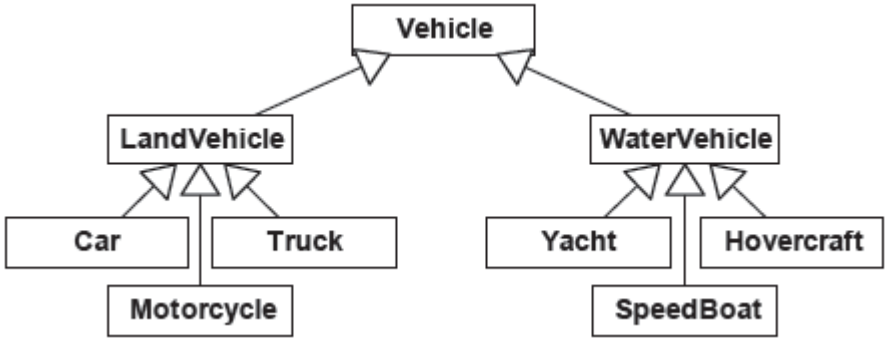
Unreal Tournament 2004 게임 객체 클래스 구조



깊고 넓은 계층의 문제점

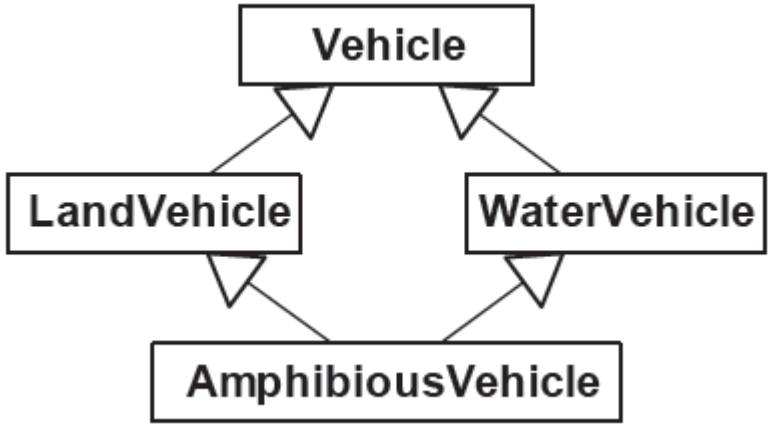
- **클래스를 이해하기 힘들고 유지 및 수정이 어려움**
  - 계층이 너무 깊다. 자식 클래스를 이해하려면 부모 클래스를 줄줄이 다 이해해야 함.
  - 자식 클래스의 가상 함수 하나를 수정하면, 자칫 부모 클래스의 가정 규칙을 어길 수 있음.

- **여러 계열 분류 구조를 구현할 수 없음.**
  - 생물학적 분류법(계,문,강,목,과, 속, 종 분류)과 유사한 방법.
  - 문제점: 객체들을 나눌 때 트리의 한 레벨에서는 오직 한 가지 기준에 따라서만 분류가 됨.
  - 일단 기준이 결정되면, 다른 기준에 의해 분류하기가 거의 불가능.
  - 아래 Vehicle 클래스 구조에서 수륙 양용차를 새로이 만드려면?



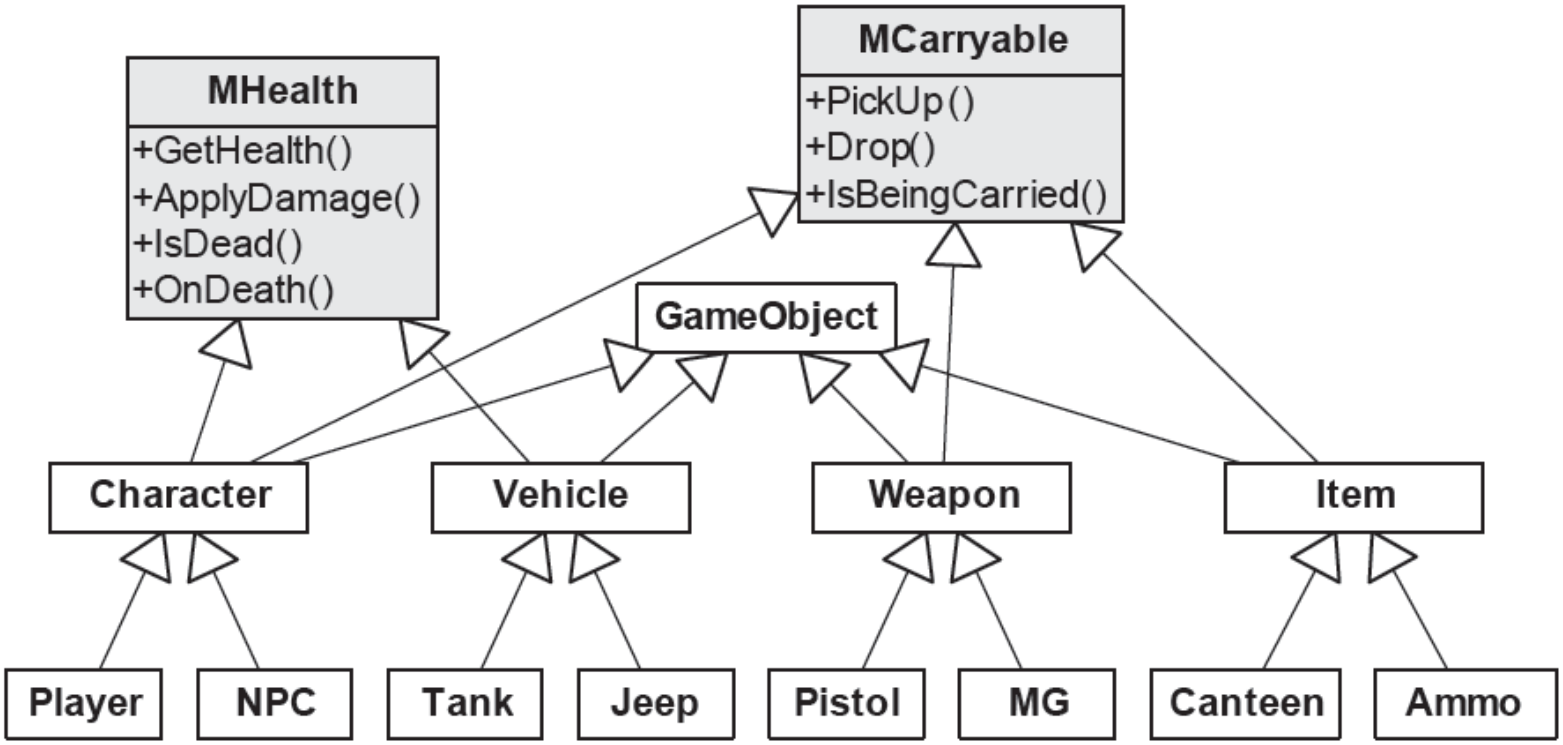
다중 상속으로 해결?

- **다중 상속의 문제**
  - 한 객체가 부모 클래스의 멤버를 여러 벌 갖게 되는 문제
  - 죽음의 다이아몬드(Deadly Diamond, Diamond of Death)
  - 좋은 다중 상속 계층 만들기 너무 어렵다!
  - 대부분의 게임 스튜디오는 다중 상속을 아예 **금지**



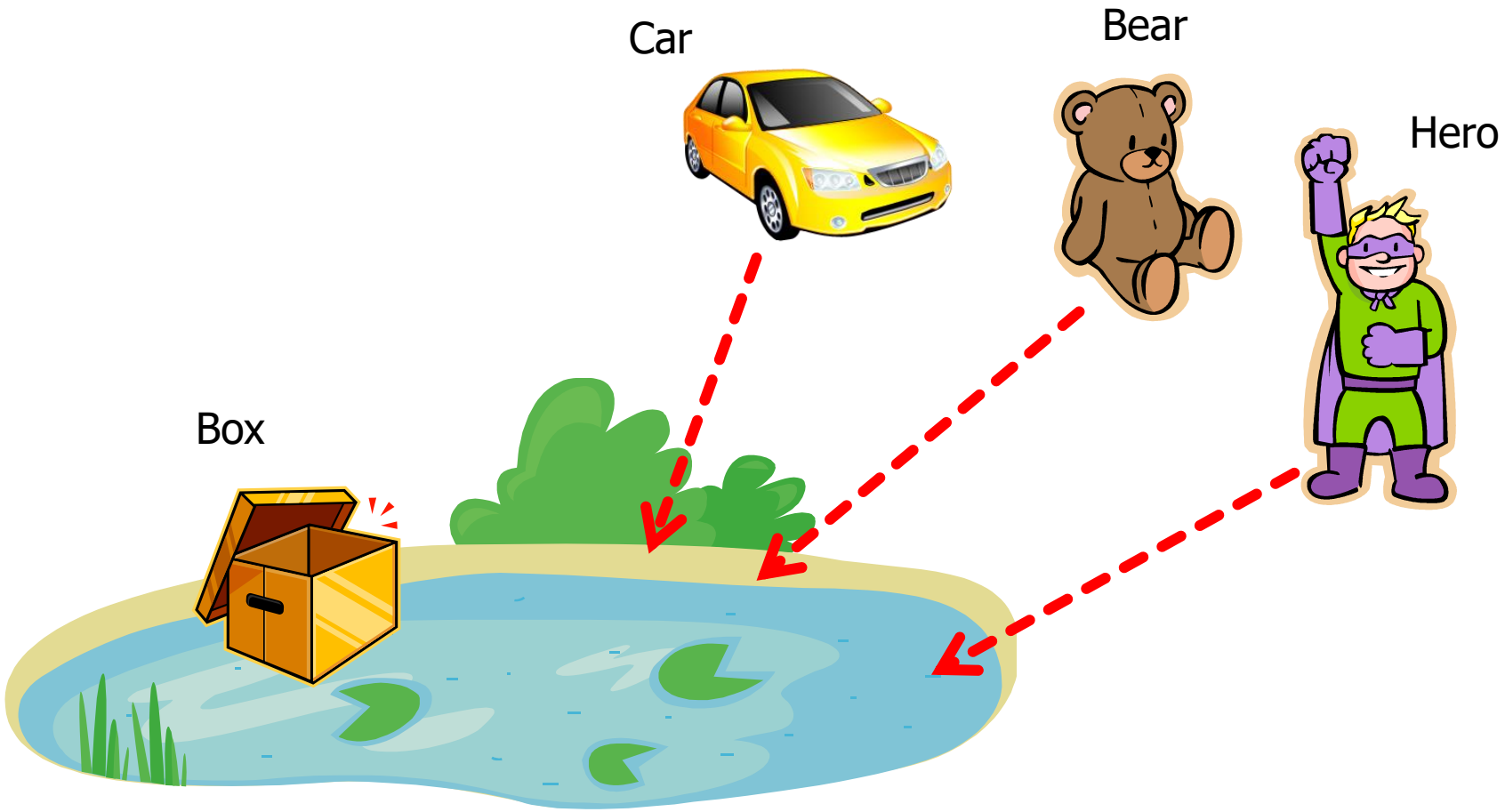
믹스 인 클래스(Mix-In Class)

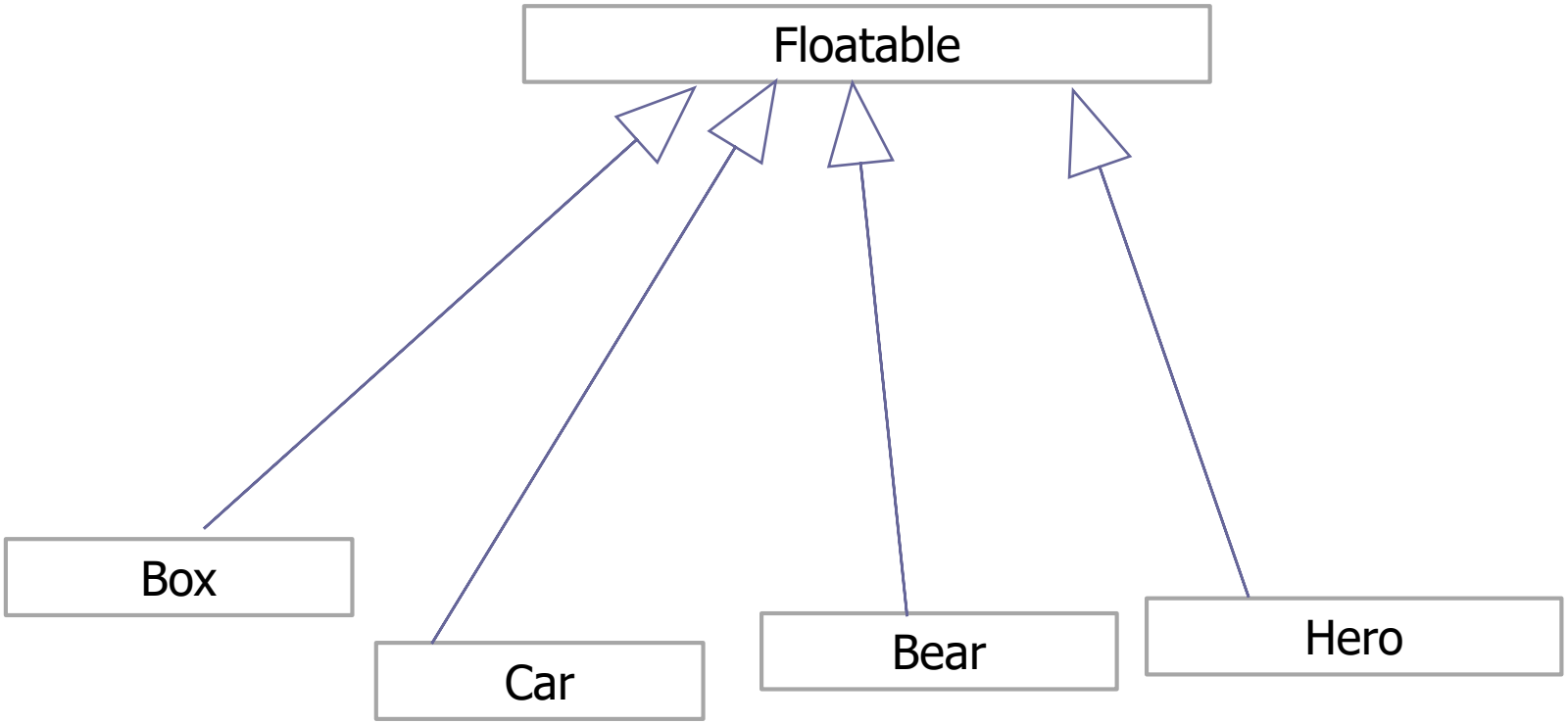
- 한 클래스는 여러 개의 부모 클래스를 가질 수 있지만, 오직 한 개의 조부모 클래스만 가질 수 있음.
- 중심적인 클래스 구조는 유지하되, 믹스인 클래스(베이스 클래스가 없는 독립된 클래스)는 여러 개 상속 할 수 있음.
- 공통 기능들을 믹스 인 클래스로 모은 후, 상속받게 할 수 있음.



버블업 효과

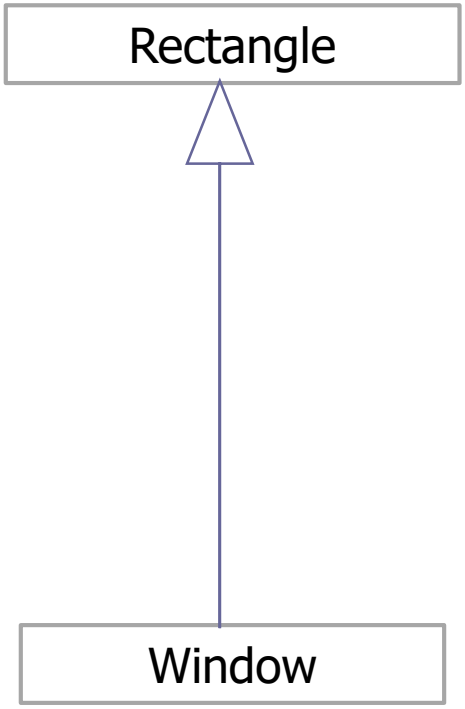
- **클래스 계층의 설계**
  - 맨 처음 설계할 때, 대부분 루트 클래스는 단순하면서 꼭 필요한 기능만 외부에 노출함.
  - 게임에 기능이 추가되면서, 서로 연관이 없는 클래스들 사이에 코드를 공유하고 싶은 마음이 생김.
  - 결과적으로 기능들이 계층 구조에서 “버블 업” 거품처럼 위로 뛰어오르는 현상이 생김.





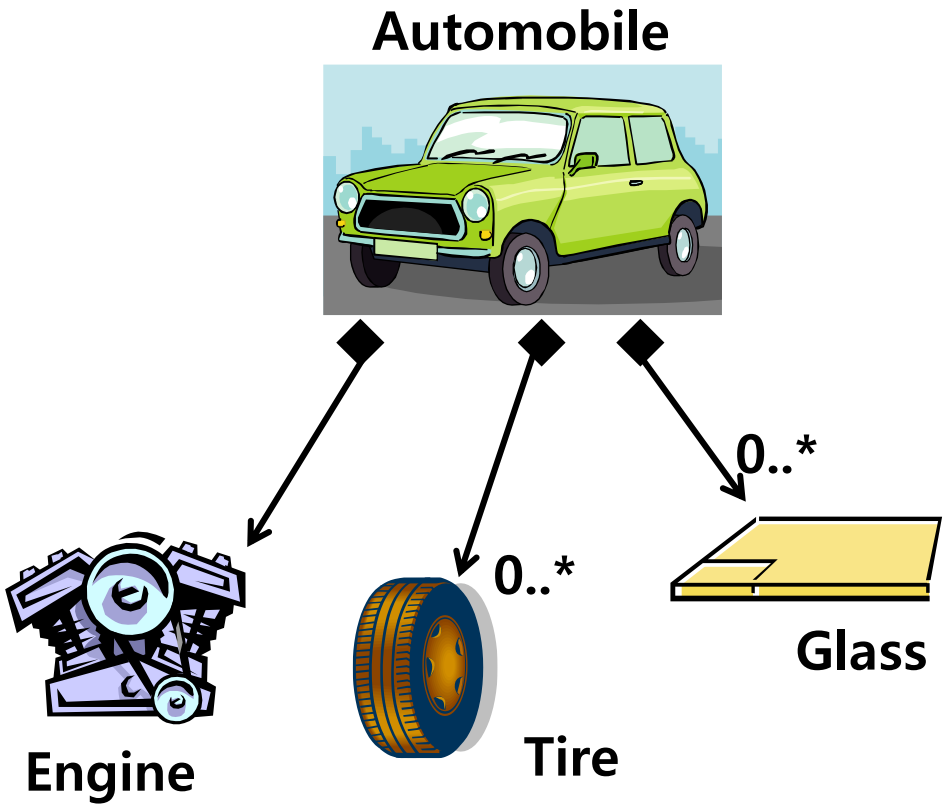
합성을 통한 계층 구조의 단순화

- **거대 단일 클래스 계층이 되는 이유?**
  - ‘is-a’ 관계의 남용, 상속 관계의 남용 및 오용
  - 속성을 상속으로 풀어 낼 경우 생기는 문제임.



합성(Composition) - 조립

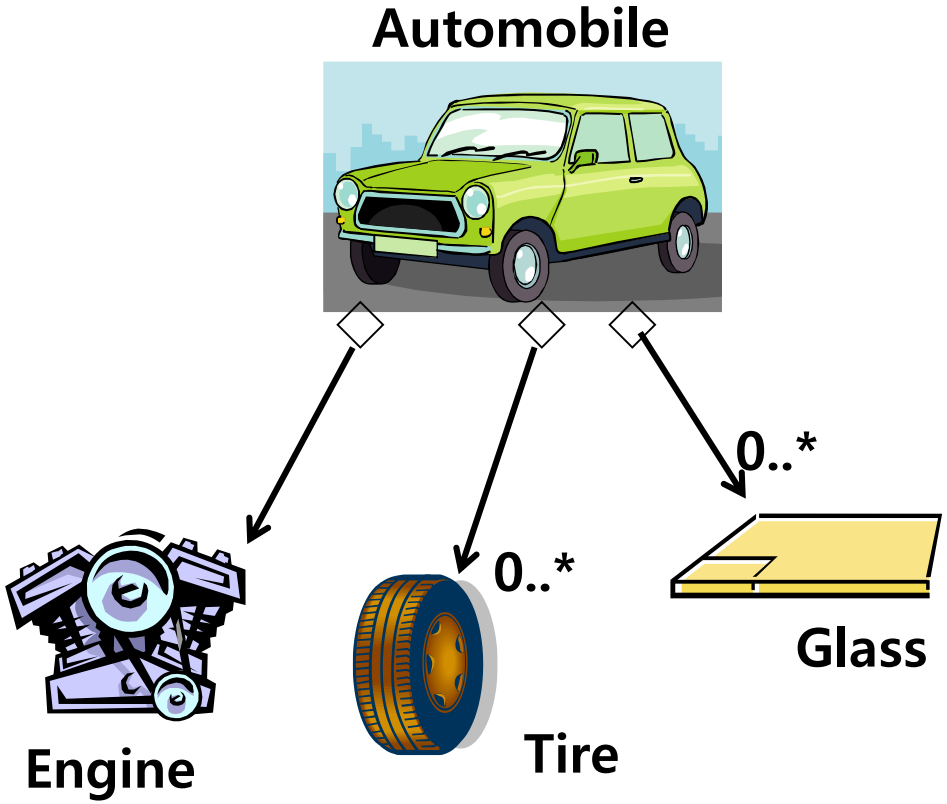
- 'has-a' 관계
- 클래스 A가 클래스 B의 인스턴스를 포함
- A 인스턴스가 생성되고 소멸될 때, B의 인스턴스로 생성 또는 소멸




```
class Automobile {
   Engine theEngine;
   Tire theTire[4];
   Glass theGlass[6];
...
};
```

조합(Aggregation) - 집합

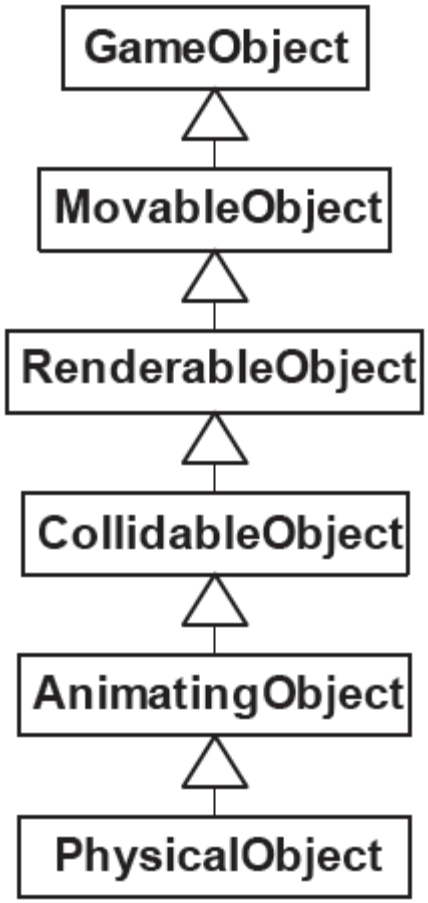
- 포함된 인스턴스의 독자적 행동이 가능하다면? - 포인터 또는 참조를 활용.




```
class Automobile {
   Engine *theEngine;
   Tire *theTire[4];
   Glass *theGlass[6];
...
};
```

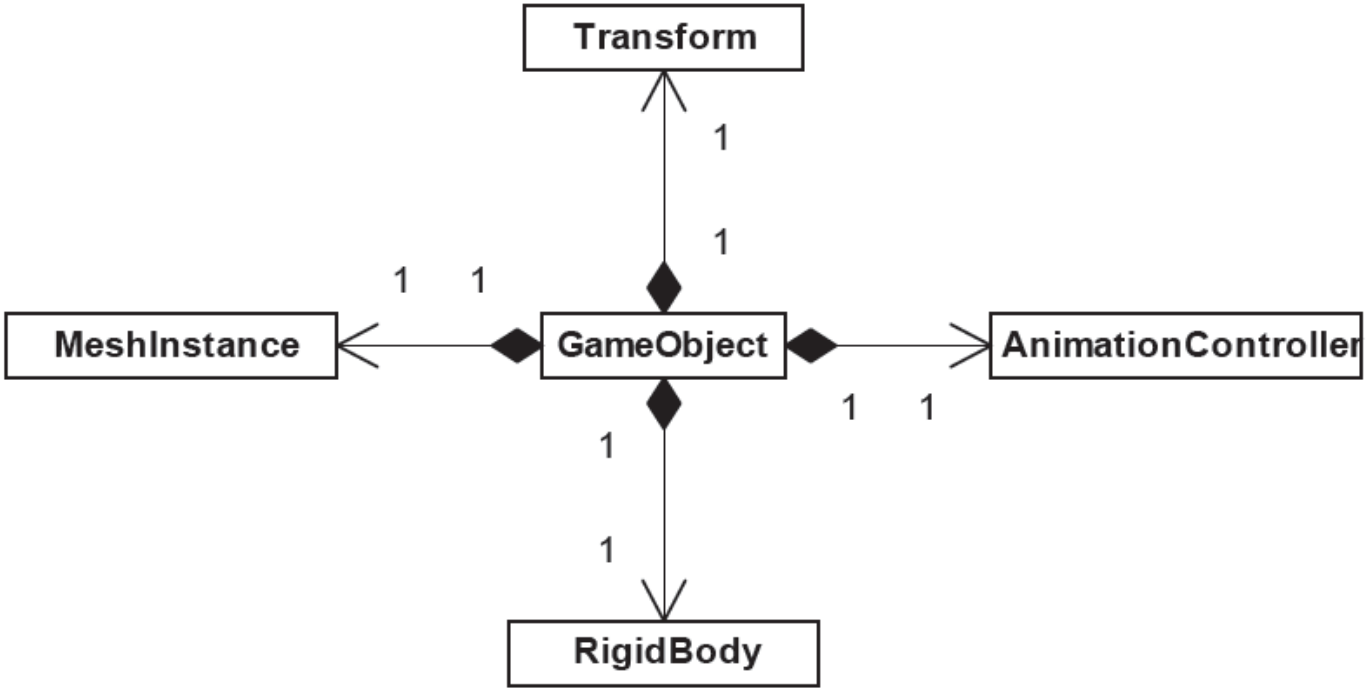
거대 단일 계층

- **물리 시뮬레이션 객체 타입을 정의(새로운 클래스의 정의)하려면?**
  - 애니메이션 하지 않더라도, PhysicalObject를 상속받을 수 밖에…



컴포넌트 객체의 합성 구조

- 서비스 객체
- GmaeObject의 여러 기능을 독립된 클래스로 분리
- 한 클래스는 한 가지 잘 정의된 서비스만 지원
- 허브 클래스(Game Object)가 컴포넌트 객체들의 수명을 관리
- 게임 객체들을 GameObject를 상속받아서 정의
  - 상속된 클래스의 생성자에서 각자 필요에 따라 컴포넌트를 생성하면 됨.



GameObject Class

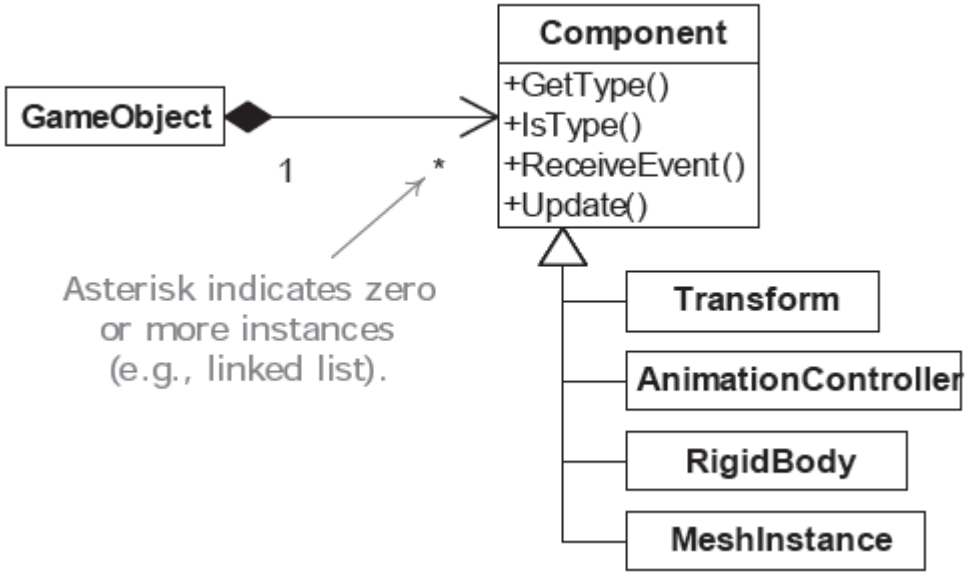
```
class GameObject
{
protected:

    // My transform (position, rotation, scale).
 Transform       m_transform;

    // Standard components:
 MeshInstance*          m_pMeshInst;
 AnimationController*       m_pAnimController;
 RigidBody*                 m_pRigidBody;

public:

    GameObject()
 {
        // Assume no components by default. Derived
        // classes will override.
        m_pMeshInst = NULL;

        m_pAnimController = NULL;

        m_pRigidBody = NULL;
 }

    ~GameObject()
 {
        // Automatically delete any components created by
        // derived classes.
  delete      m_pMeshInst;

  delete      m_pAnimController;

  delete      m_pRigidBody;
 }

    // ...
};
```

```
class Vehicle : public GameObject
{
protected:
    // Add some more components specific to Vehicles...
 Chassis*     m_pChassis;
 Engine*      m_pEngine;

    // ...

public:

    Vehicle()
 {
       // Construct standard GameObject components.
  m_pMeshInst      = new MeshInstance;

  m_pRigidBody      = new RigidBody;

       // NOTE: We'll assume the animation controller
       // must be provided with a reference to the mesh
       // instance so that it can provide it with a
       // matrix palette.
  m_pAnimController
   =        new AnimationController(*m_pMeshInst);

       // Construct vehicle-specific components.
  m_pChassis      = new Chassis(*this,
                    *m_pAnimController);
  m_pEngine      = new Engine(*this);

 }

    ~Vehicle()
 {
       // Only need to destroy vehicle-specific
       // components, as GameObject cleans up the
       // standard components for us.
  delete      m_pChassis;

  delete      m_pEngine;
 }
};
```

Generic Component

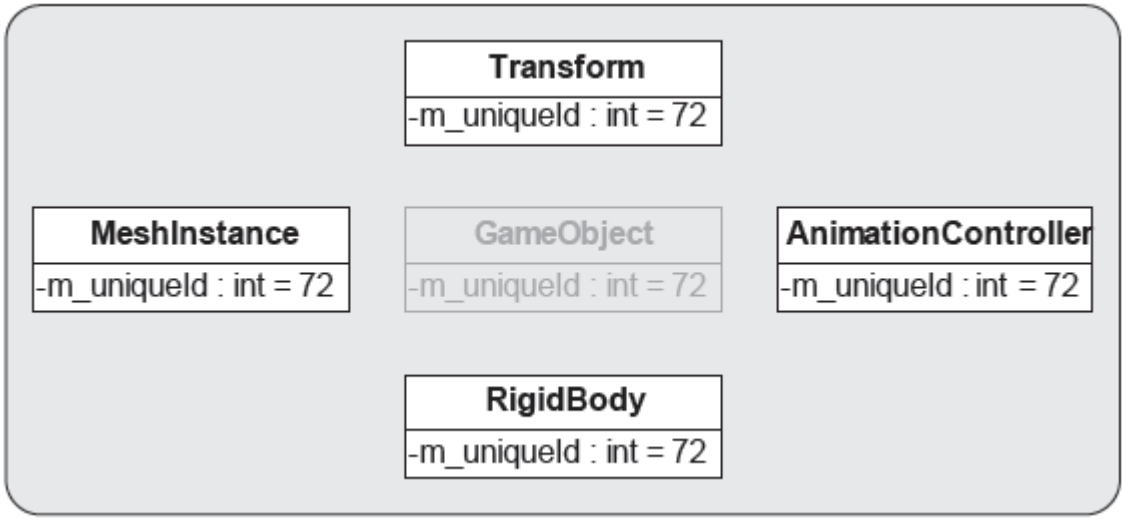
- **Generic Component(일반컴포넌트)**
  - 컴포넌트들의 베이스 클래스

- **GameObject 루트 클래스가 제너릭컴포넌트의 리스트를 관리**
- **GameObject 객체는 구체적으로 어떤 타입의 컴포넌트가 있는지 신경 쓸 필요가 없음.**
- **장점은?**
  - 새 타입의 컴포넌트를 만들 때, GameObject 클래스를 수정할 필요가 없음.
  - 게임 객체에서 가질 수 있는 컴포넌트의 수에 제한이 없음.



순수 컴포넌트 모델

- GameObject 클래스에서 모든 기능을 컴포넌트화시키면?
- 결국 GameObject 객체는 행동이 없는 컨테이너?
  - 고유 id와 컴포넌트에 대한 포인터만 갖게 됨.
  - 논리적인 행동이 없음.
  - 따라서, 차라리 없애버리면?? GameObject는 ID로 표현될 수 있으므로, id를 컴포넌트에 심어버리는 방법.
- 고민이 필요한 문제들은?
  - 최종적인 게임 객체들의 구체적인 타입은 정의할 필요가 있고, 그에 따른 컴포넌트의 인스턴스를 생성할 방법이 필요 → 팩토리 패턴을 이용하면 됨.
  - 컴포넌트간 통신은? 객체간의 통신은?



객체 중심 관점

- Object1
  - Position = (0, 3, 15)
  - Orientation = (0, 43, 0)
- Object2
  - Position = (–12, 0, 8)
  - Health = 15
- Object3
  - Orientation = (0, –87, 10)

속성 중심 구조

- Position
  - Object1 = (0, 3, 15)
  - Object2 = (–12, 0, 8)
- Orientation
  - Object1 = (0, 43, 0)
  - Object3 = (0, –87, 10)
- Health
  - Object2 = 15

속성 중심 설계

- 관계형 데이터베이스와 유사
- 각 속성은 테이터베이스 테이블의 컬럼의 역할
- ID는 프라이머리 키

- 그럼 행동의 구현은 어떻게?
  - 속성들 그 자체안에서 구현
  - 스크립트 코드를 통해 구현

속성 클래스를 통한 행동의 구현

- **속성타입 자체를 속성 클래스로 구현**

- **속성**
  - 단순하게는 bool, float
  - 복잡하게는 메쉬, AI 두뇌도 속성임

- **속성 클래스**
  - 하드 코딩된 메쏘드를 통해 행동을 제공
  - 게임 객체의 전체적인 행동 = 객체의 모든 속성들의 행동의 합

- **“health” 속성의 경우**
  - 게임 객체에 공격이 가해지면, health obje

```
static const U32 MAX_GAME_OBJECTS = 1024;

// Traditional array-of-structs approach.

struct GameObject
{
 U32      m_uniqueId;
 Vector     m_pos;
 Quaternion m_rot;
 float      m_health;
   // ...
};
GameObject g_aAllGameObjects[MAX_GAME_OBJECTS];

// Cache-friendlier struct-of-arrays approach.

struct AllGameObjects
{
 U32      m_aUniqueId        [MAX_GAME_OBJECTS];
 Vector     m_aPos      [MAX_GAME_OBJECTS];
 Quaternion m_aRot      [MAX_GAME_OBJECTS];
 float      m_aHealth      [MAX_GAME_OBJECTS];
   // ...
};
AllGameObjects g_allGameObjects;
```