# Crazy Arcade 기술문서

김동현

# Contents

# 0. 게임 개요

- **Platform :** Windows
- **개발 언어 :** C++
- **라이브러리 :** WinAPI, DirectX11, FMOD
- **제작 기간 :** 1개월
- **제작자 :** 김동현
- **Video :** https://youtu.be/sivbOkdwXT0?si=nwD7aMcKPKCm-Kkr
- **Github :** https://github.com/DongHyun96/DX_CrazyArcade_PortFolio

# Enemy AI

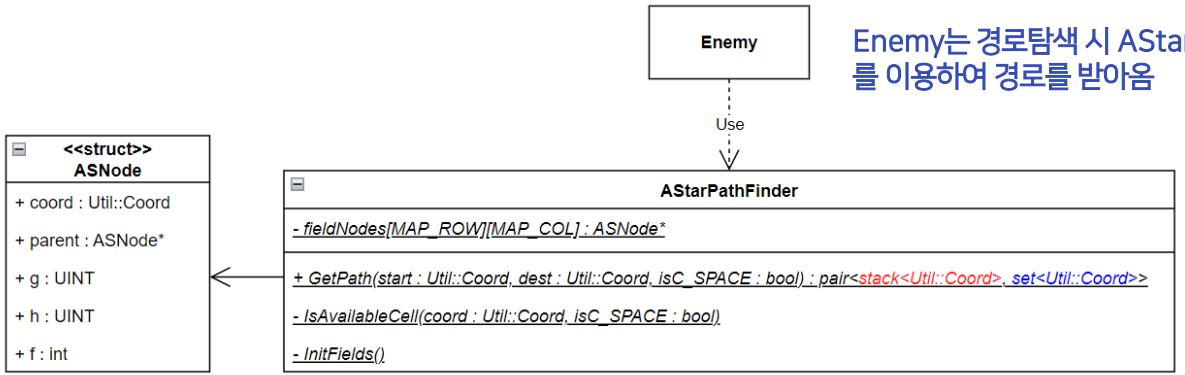
A* algorithm, Enemy decision-making FSM

Crazy Arcade

# 1. Enemy AI

## " A* Algorithm "

- Enemy의 길찾기 알고리즘으로 A* algorithm 사용
- 휴리스틱 값으로 유클리드 거리 사용

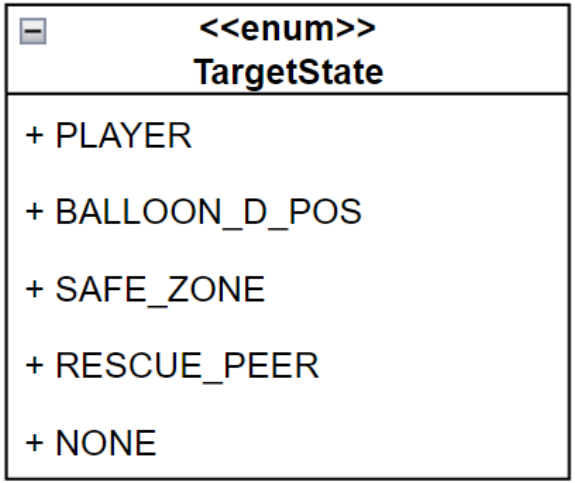



Enemy는 경로탐색 시 AStarPathFinder 를 이용하여 경로를 받아옴

```
/// <summary>
/// A* 알고리즘을 통한 start부터 dest까지의 경로 찾기
/// </summary>
/// <param name="start">     : 출발점                                                    </param>
/// <param name="dest">      : 도착점                                                    </param>
/// <param name="isC_SPACE"> : 움직이려는 객체의 상태가 C_SPACE인지(우주선일 경우 최단경로 계산이 달라짐) </param>
/// <returns> path, 검사한 visited coords | 만약 경로가 존재하지 않다면 empty path return </returns>
static pair<stack<Util::Coord>, set<Util::Coord>> GetPath(const Util::Coord& start, const Util::Coord& dest, const bool& isC_SPACE);
```

# “ Decision-making FSM“

- **현재 이동할 목표인 TargetState와 경로 상황에 따른 FSM 처리 구현**




- 플레이어에게 이동
- 물풍선을 놓을 자리로 이동
- 물줄기에 닿지 않을 자리로 이동
- 구조요청을 받은 상태로, 물풍선에 갇힌 동료 위치로 이동
- 목표 target이 잡히지 않은 상태

```
private: /* 이동 경로 관련 */

    stack<Util::Coord>  path{};     // AStarPathFinder를 통한 목표지점까지의 경로
    set<Util::Coord>    visited{};  // AStarPathFinder를 통해 AStar 알고리즘 상 방문했던 좌표들
```

# 1. Enemy AI

## “ Decision-making FSM“

- 현재 이동할 목표인 TargetState와 경로 상황에 따른 FSM 처리 구현

```
/* targetState에 따른 FSM 상황조치 */
void Enemy::UpdateState()
{
    // path가 empty이면 도착했다는 얘기

    switch (targetState)
    {
    case Enemy::PLAYER:
        if (path.size() <= player_approach_lv){ ... }
        break;
    case Enemy::BALLOON_D_POS: // 물풍선을 놓은 뒤 가장 가까운 safe_zone으로 가야함
        if (path.empty()){ ... }
        break;
    case Enemy::SAFE_ZONE:
        if (path.empty()){ ... }
        break;
    case Enemy::RESCUE_PEER:

        if (rescueTarget->GetCharacterState() != C_CAPTURED || path.empty()){ ... }
        break;
    case Enemy::NONE:{ ... }
        break;
    default:
        break;
    }
}
```

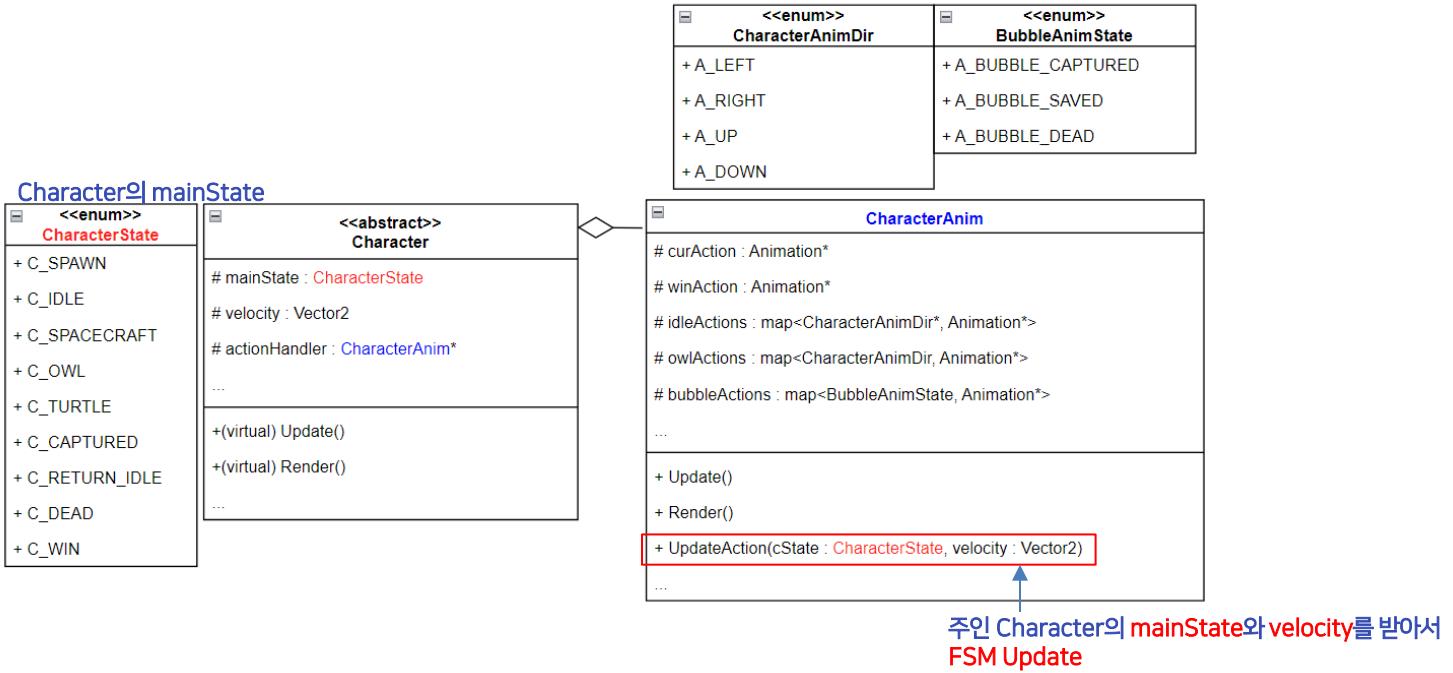
# Animation FSM

Character Animation FSM

Crazy Arcade

# " Character Animation FSM"

- 캐릭터의 mainState와 velocity 정보를 통해  anim transition이 일어남
- CharacterAnim 객체가 실질적으로 화면에 출력될 캐릭터의 anim sprite를 관리

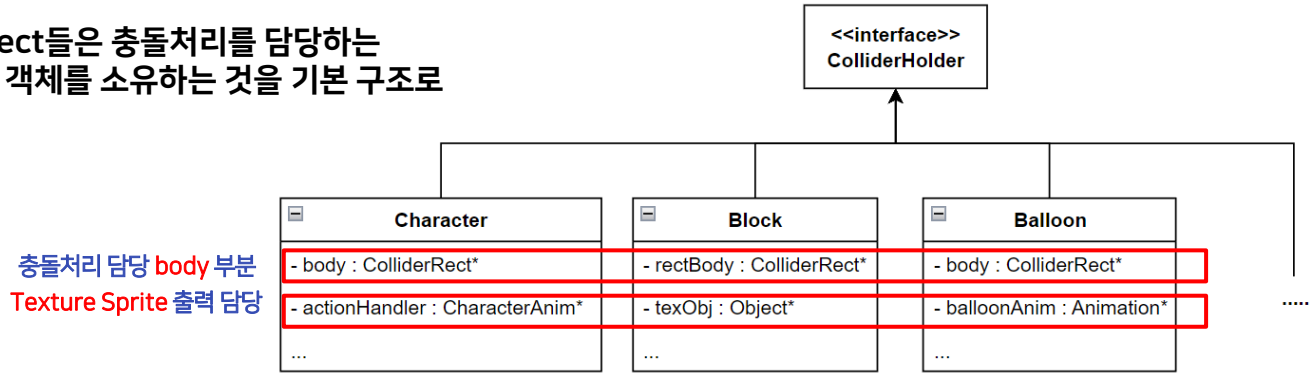


# 충돌상태처리

OnColliderEnter, OnColliderStay, OnColliderExit

# 3. 충돌상태처리

- 충돌처리가 필요한 Game object들은 충돌처리를 담당하는 body 부분과 sprite 출력담당 객체를 소유하는 것을 기본 구조로 가짐

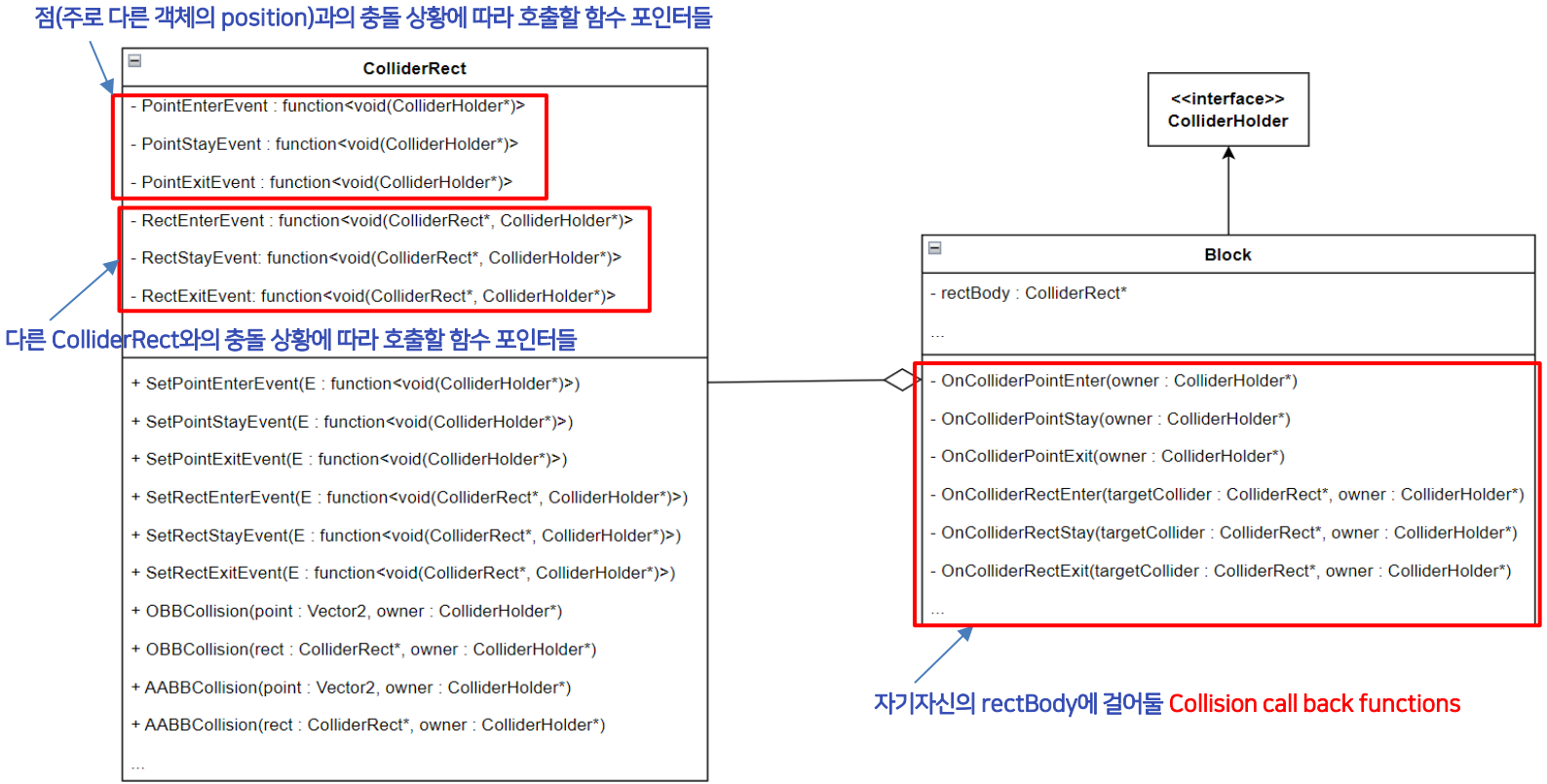



- 충돌처리가 필요한 GameObject들은 모두 ColliderHolder를 상속

- 추후 Collision 상태에 따른 Call back 함수 내에서 down-casting을 통해 어떤 형식의 Collider body owner가 들어왔는지 구분

```
void StreamBlock::OnColliderPointEnter(ColliderHolder* owner)
{
    Character* c = dynamic_cast<Character*>(owner);

    if (c)
    {
        CharacterState cs = c->GetCharacterState();

        switch (cs)
        {
        case C_IDLE:
            c->SetCharacterState(C_CAPTURED);
            break;
        case C_SPACECRAFT: case C_OWL: case C_TURTLE:
            c->SetCharacterState(C_RETURN_IDLE);
```

# 3. 충돌상태처리


점(주로 다른 객체의 position)과의 충돌 상황에 따라 호출할 함수 포인터들
ColliderRect
- PointEnterEvent : function<void(ColliderHolder*)>
- PointStayEvent : function<void(ColliderHolder*)>
- PointExitEvent : function<void(ColliderHolder*)>
- RectEnterEvent : function<void(ColliderRect*, ColliderHolder*)>
- RectStayEvent: function<void(ColliderRect*, ColliderHolder*)>
- RectExitEvent: function<void(ColliderRect*, ColliderHolder*)>
다른 ColliderRect와의 충돌 상황에 따라 호출할 함수 포인터들
+ SetPointEnterEvent(E : function<void(ColliderHolder*)>)
+ SetPointStayEvent(E : function<void(ColliderHolder*)>)
+ SetPointExitEvent(E : function<void(ColliderHolder*)>)
+ SetRectEnterEvent(E : function<void(ColliderRect*, ColliderHolder*)>)
+ SetRectStayEvent(E : function<void(ColliderRect*, ColliderHolder*)>)
+ SetRectExitEvent(E : function<void(ColliderRect*, ColliderHolder*)>)
+ OBBCollision(point : Vector2, owner : ColliderHolder*)
+ OBBCollision(rect : ColliderRect*, owner : ColliderHolder*)
+ AABBCollision(point : Vector2, owner : ColliderHolder*)
+ AABBCollision(rect : ColliderRect*, owner : ColliderHolder*)
...
<<interface>>
ColliderHolder
Block
- rectBody : ColliderRect*
...
- OnColliderPointEnter(owner : ColliderHolder*)
- OnColliderPointStay(owner : ColliderHolder*)
- OnColliderPointExit(owner : ColliderHolder*)
- OnColliderRectEnter(targetCollider : ColliderRect*, owner : ColliderHolder*)
- OnColliderRectStay(targetCollider : ColliderRect*, owner : ColliderHolder*)
- OnColliderRectExit(targetCollider : ColliderRect*, owner : ColliderHolder*)
...
자기자신의 rectBody에 걸어둘 Collision call back functions

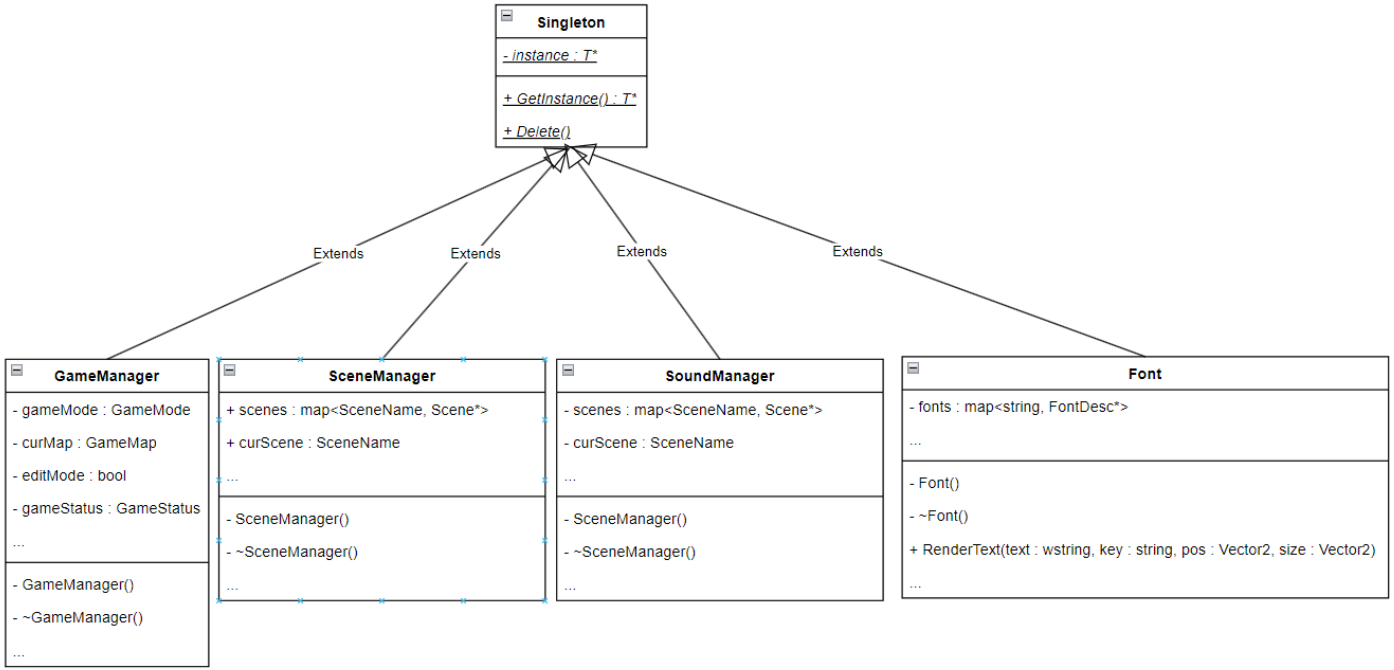
# Design patterns

Singleton, Strategy pattern, Template method pattern

Crazy Arcade

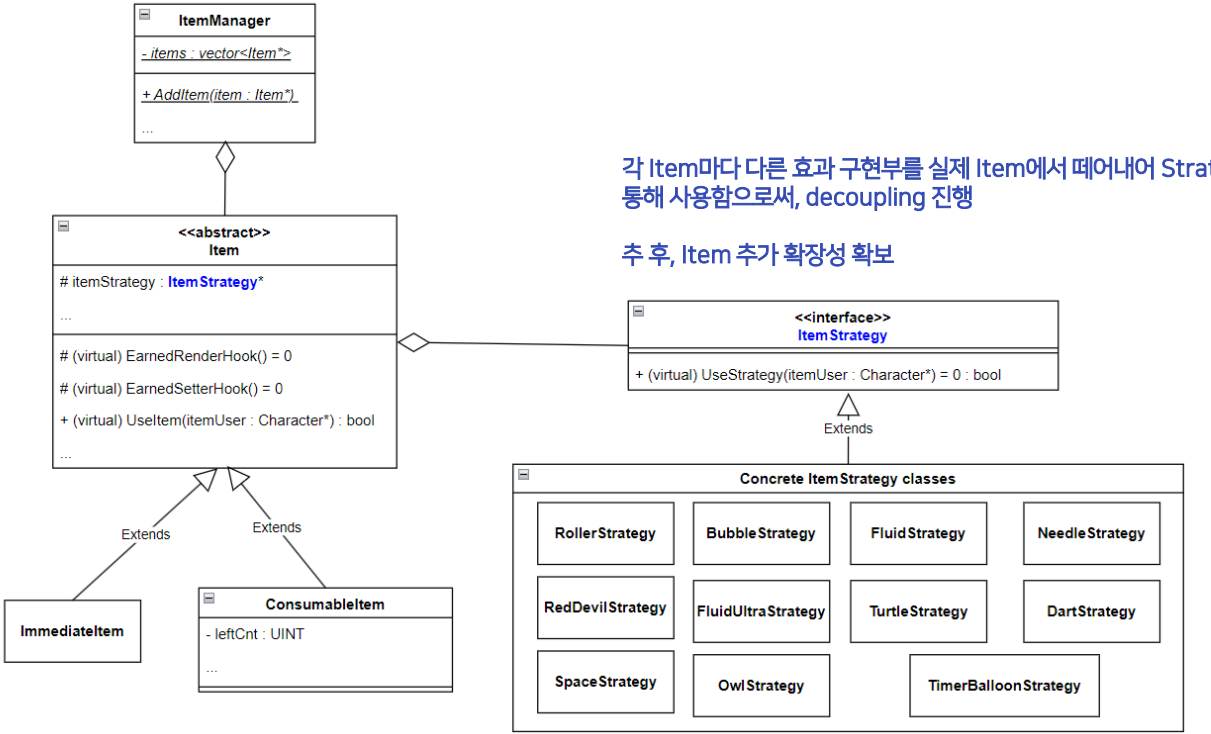
# “ Singleton ”

- Manager와 같은 클래스들은 Singleton 패턴으로 구현함으로써 접근성을 높이고, 단일 객체만 허용 가능하도록 구현



# 4. Design patterns

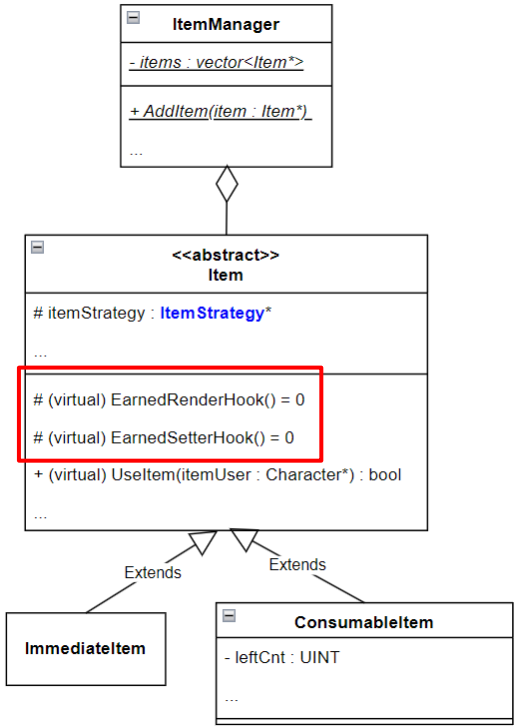
## " Strategy pattern "




각 Item마다 다른 효과 구현부를 실제 Item에서 떼어내어 Strategy interface를 통해 사용함으로써, decoupling 진행

추 후, Item 추가 확장성 확보

# " Template method pattern "



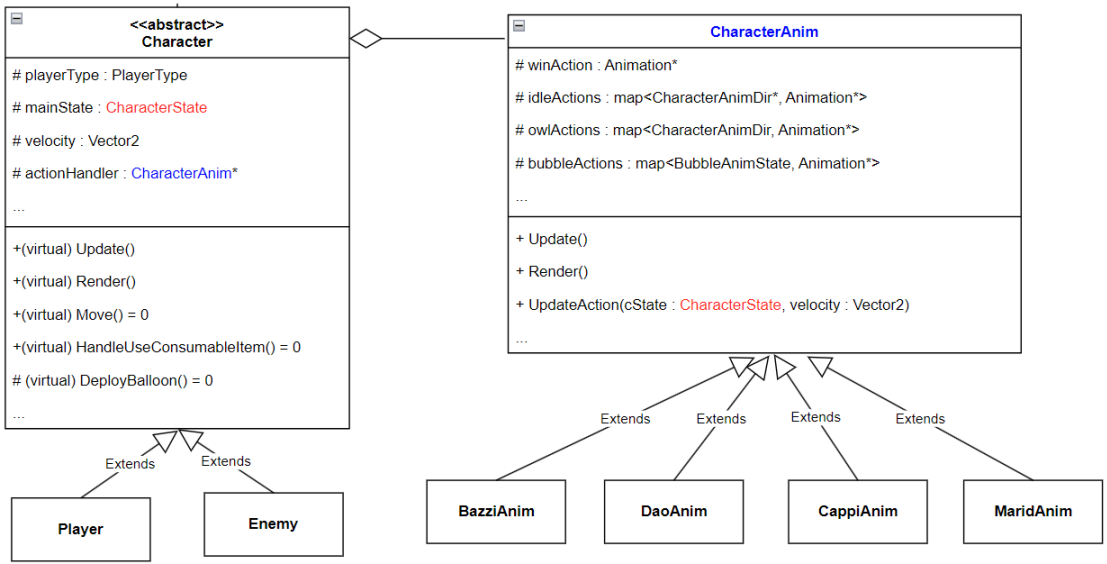

```cpp
void Item::SetItemState(const ItemState& itemState)
{
    // prevItemState
    if (this->itemState == SPAWNED) { ... }

    // prevItemState
    if (this->itemState == EARNED) { ... }

    switch (itemState)
    {
    case SPAWNED:
        isActive = true;
        body->scale = {};
        break;
    case EARNED:
        EarnedSetterHook(); // Hook Method (자식에서 결정)
        break;
    default:
        break;
    }

    this->itemState = itemState;
}
```

```cpp
void Item::Render()
{
    if (!isActive)
        return;

    switch (itemState)
    {
    case HIDDEN: case SPAWNED:  break;
    case EARNED:                EarnedRenderHook(); // Hook Method  (자식에서 결정)
        break;
    default:
        break;
    }

    body->Render();
    texObj->Render();
}
```

Item의 SetItemState()와 Render()의 특정 단계에서, 자식에 따라 처리가 다른 부분은
순수가상 함수인 Hook method으로 두어 해당 단계 구현을 자식 단에서 처리함

# " 기타 Decoupling "

- 캐릭터의 animation의 기본 틀은 모두 같고, 캐릭터 종류에 따른 anim sprite만 다름
- 기본적인 캐릭터 animation 틀은 CharacterAnim 부모 클래스에서 구현하고, 각 캐릭터 sprite는 자식 클래스에서 서로 다르도록 구현
- Character 클래스는 자신이 어떤 캐릭터인지 알 필요 없이 Character Animation을 사용하도록 decoupling 구현

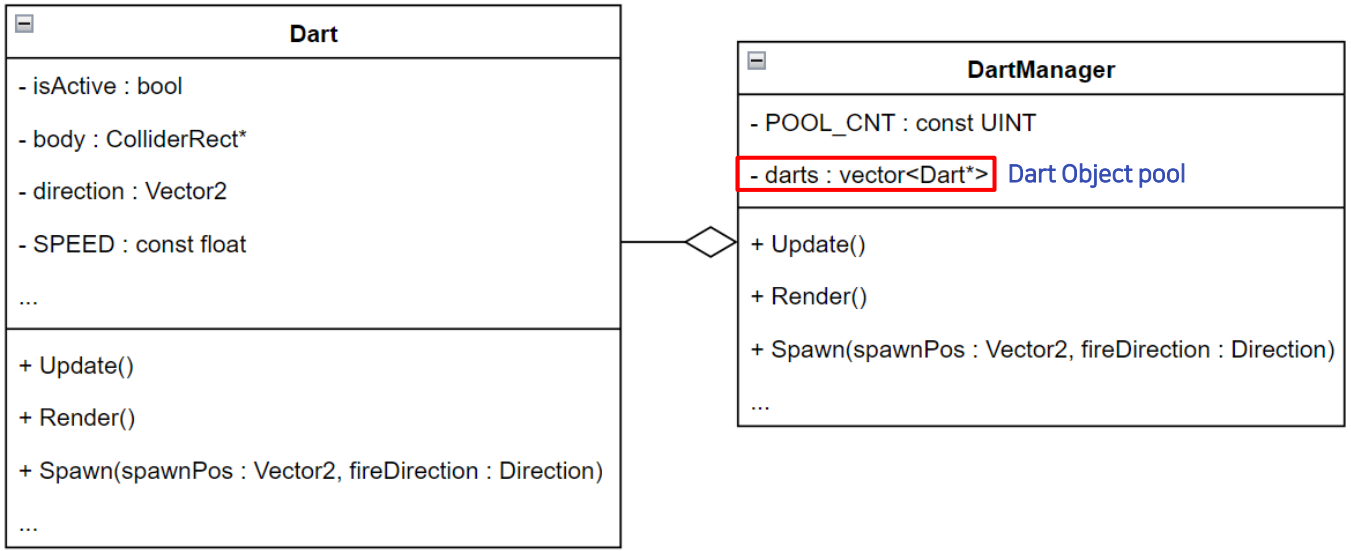


# Object pooling

Dart, Balloon, Stream, StreamBlock

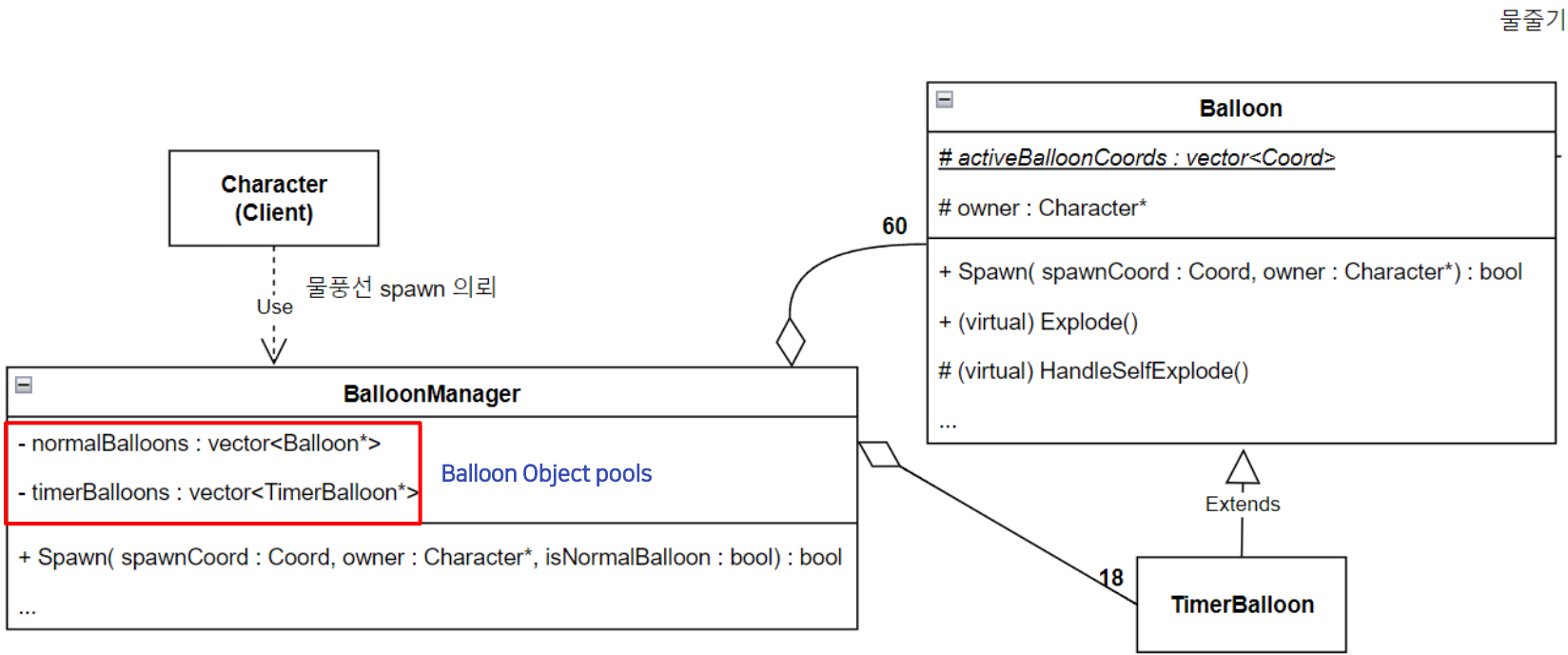
# 5. Object pooling

## “ Dart Pooling “



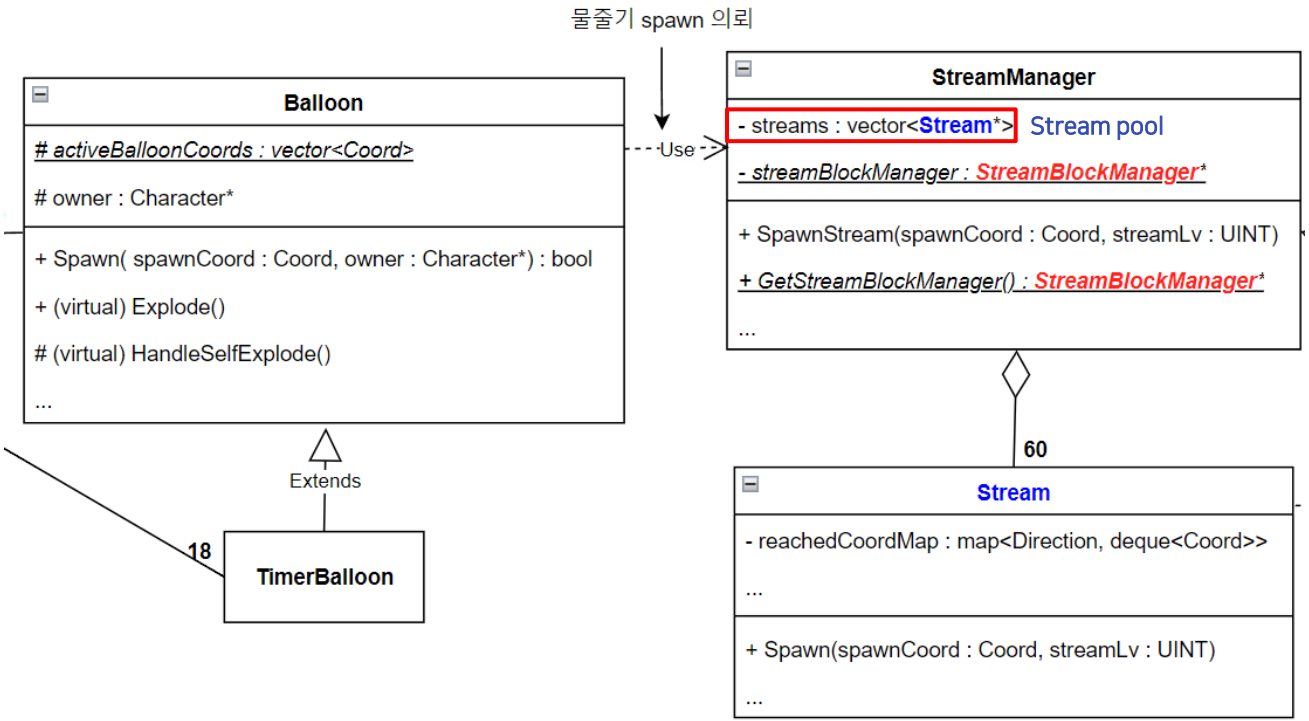
# 5. Object pooling

## “ Balloon Pooling “



# 5. Object pooling

## “Stream Pooling “



# 5. Object pooling

## “StreamBlock Pooling “