# 22_ 비헤이비어 트리 본격적으로 만들기

<제목 차례>



인천대학교 컴퓨터공학부 박종승

## 1. 개요

이 장에서는 비헤이비어 트리를 본격적으로 만드는 방법을 학습한다.

비헤이비어 트리에 대해서 더 자세히 알아보자.
비헤이비어 트리에서의 노드의 종류에는 루트 노드를 제외하면 **Task** 노드와 **Composite** 노드가 있다. **Task** 노드는 보라색이고 **Composite** 노드는 회색색이다. 이들 노드에는 특별한 것을 추가할 수 있다. 바로 **Decorator**라는 것과 **서비스**라는 것을 추가할 수 있다. **Decorator**는 파란색이고 **서비스**는 녹색이다. **Decorator**는 불리언 값을 리턴하며, 부착된 노드를 실행할지의 여부를 결정한다. 보통 **Selector** 노드나 **Sequence**, 노드에 부착되어 사용된다.
**서비스**는 노드에 부착되어서 노드 이하가 실행되는 동안에 정해진 간격으로 실행을 계속한다. 틱 신호가 발생할 때마다 명시된 블루프린트 코드를 실행한다. 틱 간격은 그래픽 프레임 갱신 틱 간격과 무관하게 조절할 수 있다. **서비스**는 추격할 적군의 등장 여부 체크 등의 상태의 변화를 체크하는 목적으로 사용된다. 체크 결과는 블랙보드에 기록하여 다른 곳에서 읽어 활용할 수 있도록 한다.

<참고> 비헤이비어 트리에 대한 자세한 내용은 다음의 링크를 참조하자.
https://docs.unrealengine.com/behavior-tree-in-unreal-engine---overview/

## 2. 데코레이터의 사용

이 절에서 데코레이터의 사용에 대해서 학습한다.

먼저, 이전 예제에서의 **비헤이비어 트리**에 **Selector** 노드를 추가하여 캐릭터의 행동을 더 지능적으로 발전시켜본다.
그리고, **데코레이터**를 추가해서 추적이 가능하도록 해본다.

이제부터 예제를 통해서 학습해보자

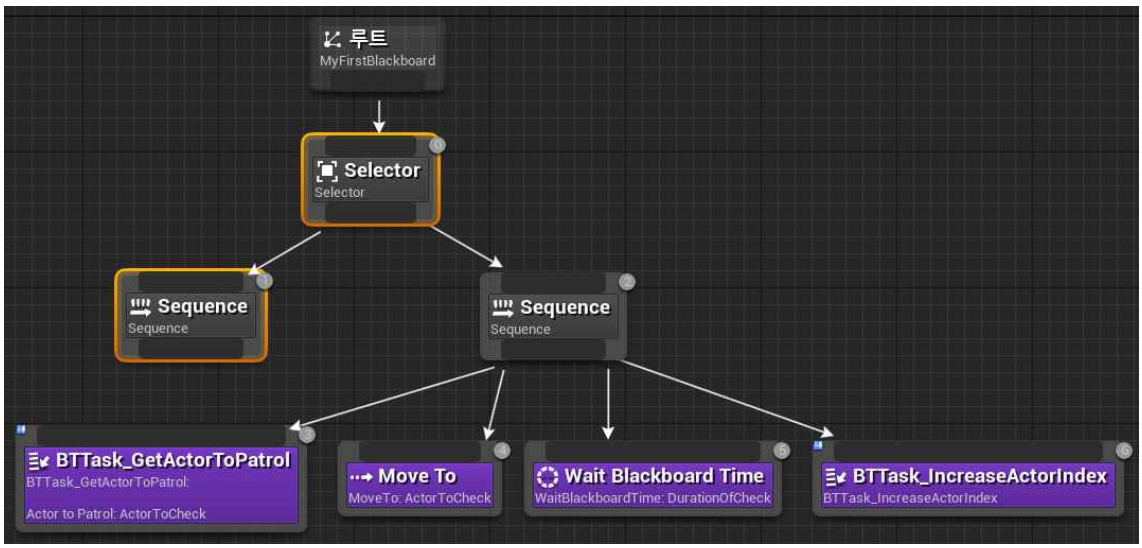
---

**1.** 새 프로젝트 Pbtchase를 생성하자. 먼저, 언리얼 엔진을 실행하고 **언리얼 프로젝트 브라우저**에서 왼쪽의 **게임** 탭을 클릭하자. 오른쪽의 템플릿 목록에서 **기본** 템플릿을 선택하자. **프로젝트 이름**은 Pbtchase로 입력하고 **생성** 버튼을 클릭하자. 프로젝트가 생성되고 언리얼 에디터 창이 뜰 것이다.

**2.** 먼저, 이전 프로젝트에서의 작업을 가져오는 일을 진행하자.
첫 번째로, 이전의 입력 처리 단원에서 만들었던 프로젝트에서 생성해둔 블루프린트 클래스를 가져오자. 윈도 파일 탐색기에서 Pinputevent 프로젝트의 **Content** 폴더로 이동하자. 그 아래에 있는 3개의 애셋 파일(BP_MyCharacter, BP_MyGameMode, BP_MyPlayerController)을 **Ctrl+C**로 복사하고 새 프로젝트의 **Content** 폴더 아래로 이동하여 **Ctrl+V**로 붙여넣자.
두 번째로, 이전 프로젝트해서 생성해둔 블루프린트 클래스를 가져오자. 이전 Pbtpatrol 프로젝트에서 Content 폴더 아래에 8개의 애셋 파일(MyMap.umap, MyMap_BuiltData.uasset, EnemyPawn.uasset, EnemyAIController.uasset, MyFirstBlackboard.uasset, MyFirstBehaviorTree.uasset, BTTask_GetActorToPatrol.uasset, BTTask_IncreaseActorIndex.uasset)이 있을 것이다. 이들을 모두 복사하여 새 프로젝트의 **Content** 폴더 아래에 붙여넣자.
세 번째로, 이전 프로젝트해서 설정한 입력 매핑을 가져오자. 우선, 새 프로젝트를 종료하자. 그다음, 이전의 프로젝트 Pinputevent 프로젝트의 **Config** 폴더로 이동하자. 그 아래에 있는 DefaultInput.ini 파일을 **Ctrl+C**로 복사하고 새 프로젝트의 **Config** 폴더 아래로 이동하여 **Ctrl+V**로 붙여넣자. 그다음, 새 프로젝트를 다시 로드하자.
네 번째로, 콘텐츠 브라우저에서 **MyMap** 애셋을 더블클릭하여 레벨을 열어보자.
그리고, **월드 세팅** 탭에서 **게임모드 오버라이드** 속성에 **BP_MyGameMode**를 지정하자.
그리고, **프로젝트 세팅** 창에서 **맵&모드** 탭에서의 **에디터 시작 맵** 속성값을 MyMap으로 수정하자.
이제, 이전 프로젝트에서의 모든 작업이 포함되도록 준비과정을 완료하였다.

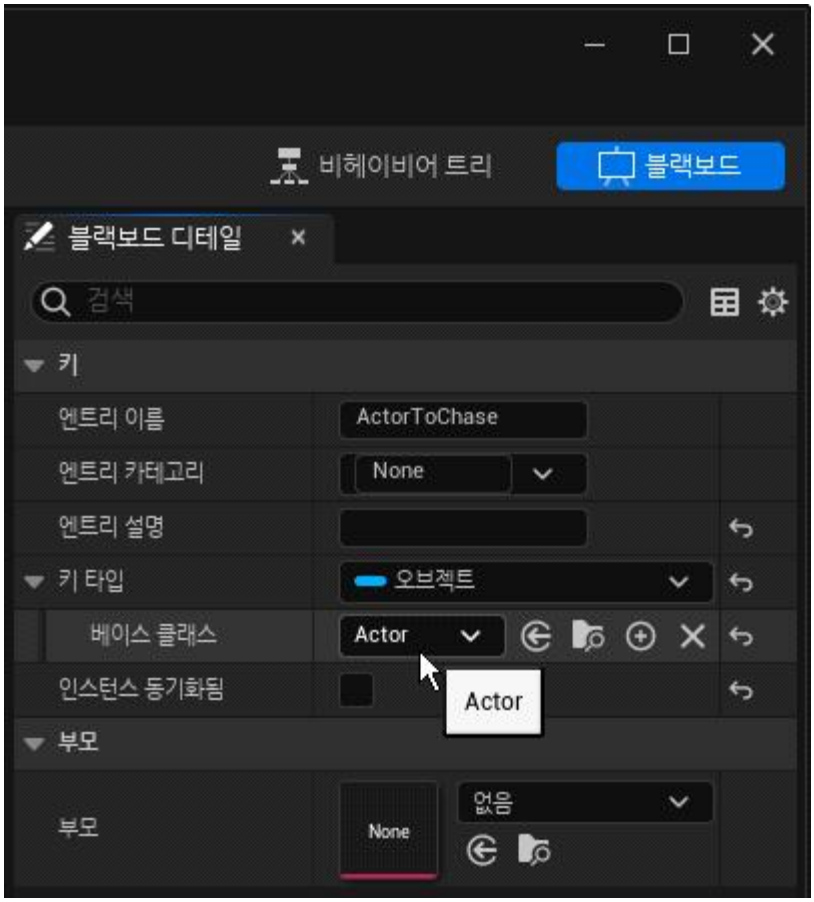
**3.** 이제부터 패트롤 행동만 수행하는 **EnemyPawn**의 행동 패턴을 약간 더 발전시켜보자.
플레이어가 보이면 플레이어 추격 행동을 하고 플레이어가 보이지 않으면 기존의 패트롤 행동을 하도록 해보자.
**콘텐츠 브라우저**에서 **MyFirstBehaviorTree**를 더블클릭하여 **비헤이비어 트리 에디터**를 열자.
먼저, **루트** 노드의 출력핀에서 와이어를 드래그해서 빈 곳에 드롭하고 선택창에서 **Composites** 아래에 있는 **Selector** 노드를 설치하자. 기존에 연결되어 있던 **Sequence** 노드와의 연결은 자동으로 끊어질 것이다.
그다음, 새로 설치된 **Selector** 노드의 출력핀에서 오른쪽 방향으로 드래그해서 기존의 끊어진 **Sequence**

노드의 입력핀으로 연결하자.
그다음, **Selector** 노드의 출력핀에서 왼쪽 방향으로 드래그하고 선택창에서 **Composites** 아래에 있는 **Sequence** 노드를 설치하자.
이제 **비헤이비어 트리**가 다음의 모양이 될 것이다.




0번 노드는 **Selector** 노드이다. 자신의 자식 노드인 1번 또는 2번 노드 중에서 우선 순위에 따라서 하나만 선택하여 실행한다. 1번 노드는 플레이어에 대한 추격 행동을 수행하는 노드이고, 2번 노드는 기존의 패트롤 행동을 수행하는 노드이다. 1번이 더 우선 순위가 높으므로 1번이 성공인 경우에는 2번은 실행하지 않는다. 우선 순위는 좌우 배치에 따라 결정되므로 우선 순위를 높게 부여할 노드를 왼쪽에 배치하면 된다.

**4.** 추격 행동을 구현하는 방법을 생각해보자.
추격할 대상이 감지되면 추격 대상을 **블랙보드**에 기록해두고, 추격 **Task**에서는 **블랙보드**를 통해서 추격 대상 정보를 확인해서 추격하도록 해보자.
먼저, **블랙보드**에서 키를 하나 추가하자. **비헤이비어 트리 에디터**의 우측 상단의 **블랙보드** 버튼을 클릭해서 **블랙보드** 에디터로 가자.
그다음, 왼쪽의 **새 키**를 클릭하고 **오브젝트(Object)** 유형의 키를 추가하자. **키**의 이름은 **ActorToChase**라고 하자. 그리고 디테일 탭에서 **키 타입(Key Type)** 속성을 펼치고 아래의 **베이스 클래스(Base Class)** 속성에서 **Object** 대신 **Actor**를 선택하자. 이렇게 하면, 최상위 **Object** 클래스 유형보다 더 구체적인 자식 클래스인 **Actor** 클래스로 범위를 좁힌다. 가급적이면 다룰 대상 유형에 가깝게 세부적으로 지정해두도록 하자.




이제 **블랙보드**에 추격할 대상을 적어둘 키인 **ActorToChase**를 추가하였다.

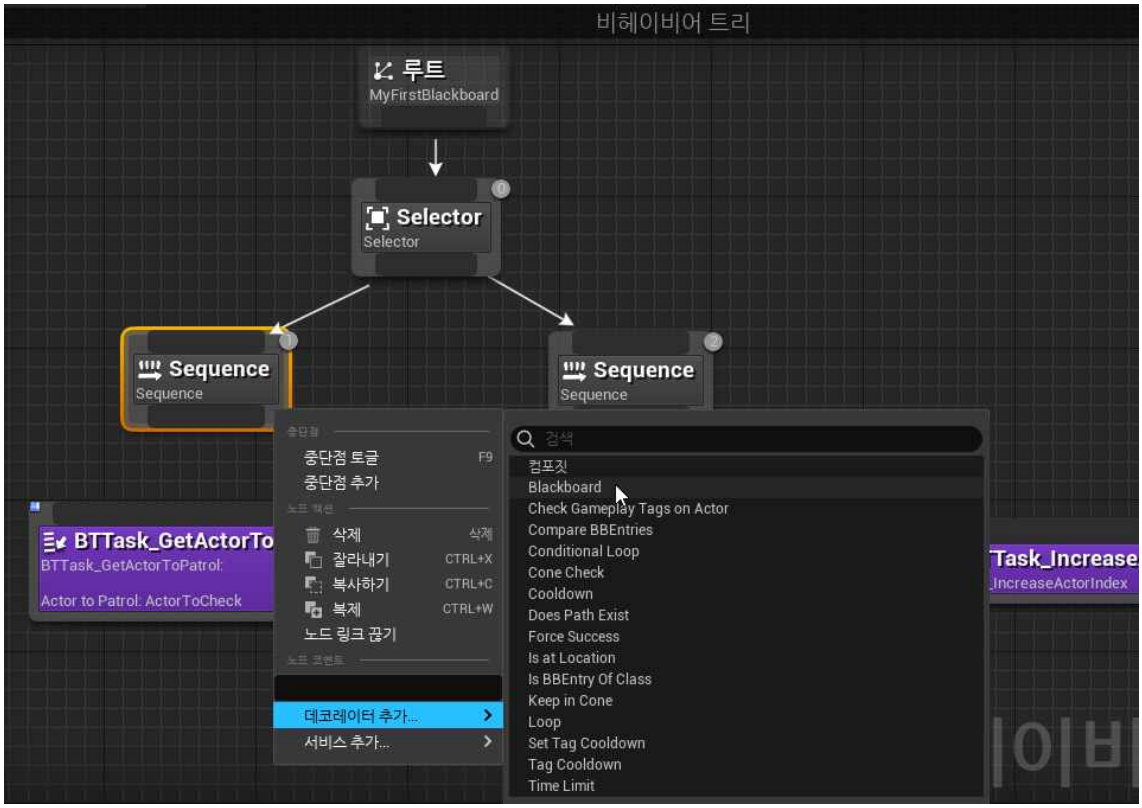
**5.** 이제부터 **데코레이터**에 대해서 알아보자.

**데코레이터**는 노드에 추가하여 노드가 더 복합한 기능을 수행할 수 있도록 한다. 대표적으로 노드의 실행 조건을 지정하는 기능을 수행하는 **데코레이터**가 있다.

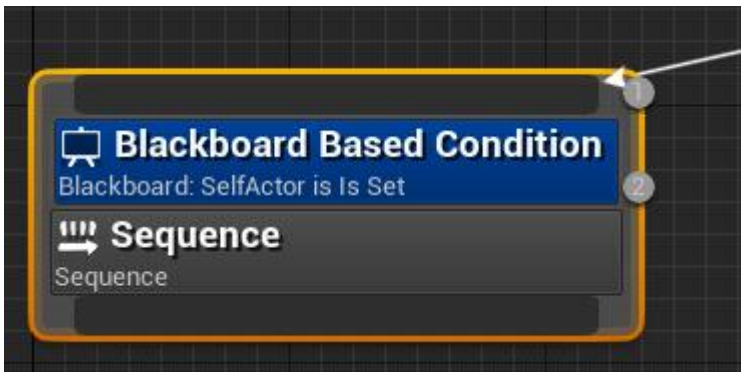
엔진은 16개의 **데코레이터**를 제공한다. 대부분의 **데코레이터**는 **블랙보드**를 통해서 정보를 교환하므로 쉽게 사용할 수 있다.

이제, **데코레이터**를 추가해보자. 먼저, **비헤이비어 트리 에디터**로 다시 돌아가자.

추격 행동을 위한 왼쪽의 **Sequence** 노드 위에서 우클릭하자. 팝업메뉴에서 **데코레이터 추가**를 선택하고 데코레이터 선택 목록에서 **Blackboard**를 선택하자.




**6.** 왼쪽의 **Sequence** 노드의 모습이 다음과 같이 바뀔 것이다.




노드에 **Blackboard Based Condition**이라는 **데코레이터**가 추가되었음을 알 수 있다. **데코레이터**도 노드 번호가 부여되어 있다. 여러 **데코레이터**가 추가된 경우에는 **데코레이터** 간의 실행 순서도 의미가 있기 때문이다.

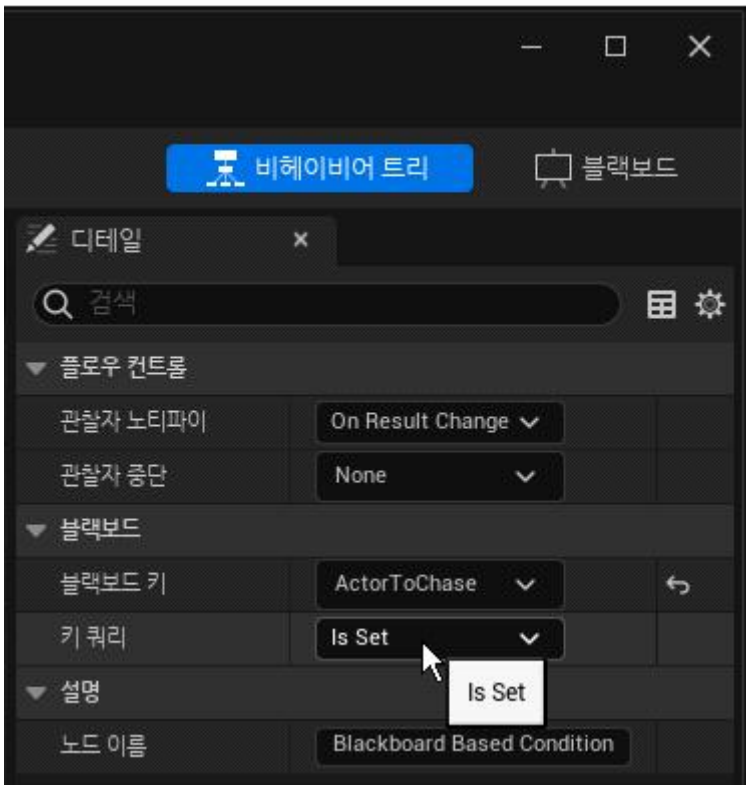
**Blackboard Based Condition 데코레이터**는 **블랙보드**의 특정 키에 값이 지정되어 있는지의 여부를 확인해서 참인 경우에 노드가 실행되도록 한다.

**7.** **Blackboard Based Condition 데코레이터**를 클릭하고 오른쪽의 **디테일** 탭에서 구체적인 조건을 명시하자.

먼저, **블랙보드** 영역의 **블랙보드 키(Blackboard Key)** 속성값에 확인할 **블랙보드**의 키를 명시하자. 우리는 **ActorToChase**를 지정하자.

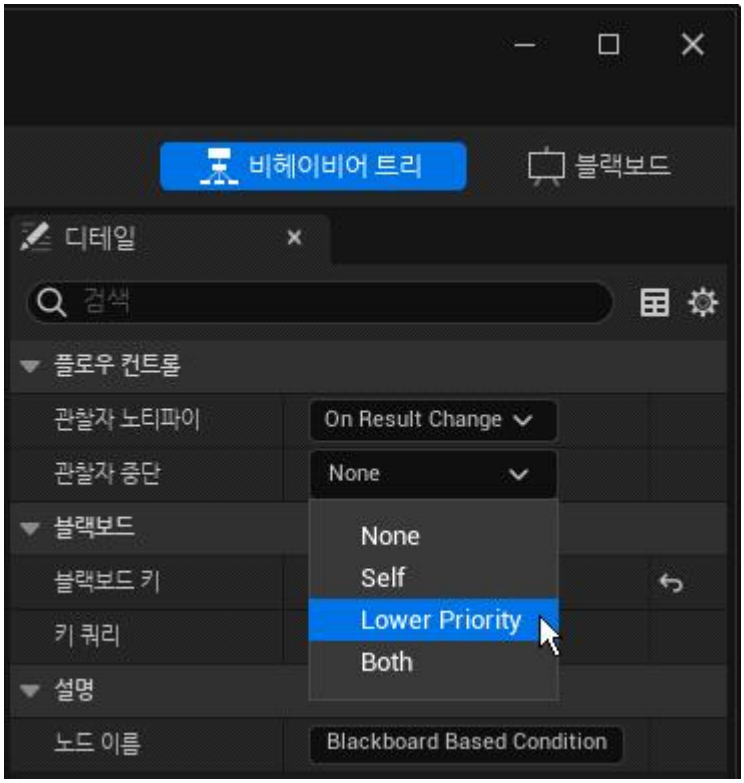
그다음, 바로 아래의 **키 쿼리(Key Query)** 속성값이 **Is Set**인지를 확인하자. **Is Set**이라면 키에 어떤 값이 설정되어 있으면 참이 된다. 반대로 **Is Not Set**으로 되어 있다면 **키**에 아무 값도 설정되어 있지

않아야 참이 된다. 우리는 **Is Set**이 되도록 하자.




이제 **블랙보드**의 **ActorToChase 키**에 아무 값도 명시되어 있지 않다면 **데코레이터**가 **false**로 되고 따라서 노드가 실행되지 않고 즉시 실패를 리턴할 것이다.

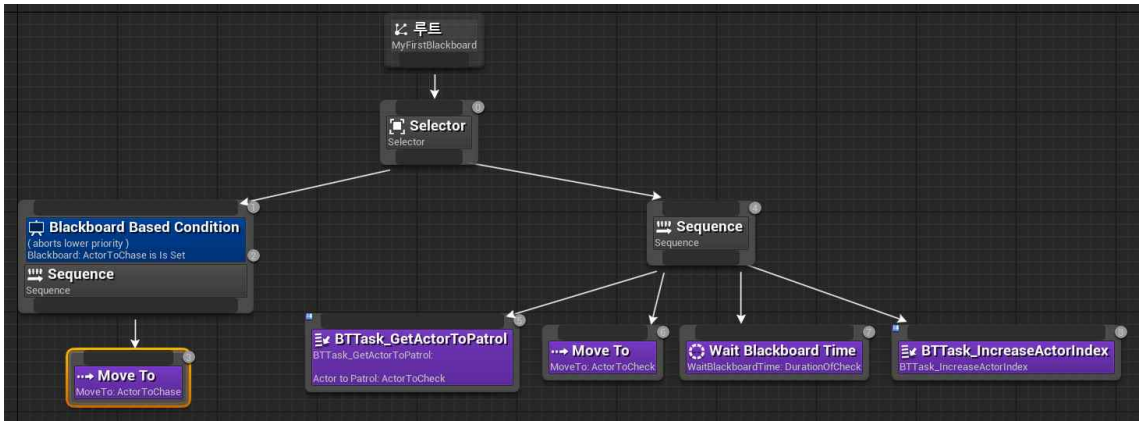
**8.** **디테일** 탭에서 또하나의 속성이 있다. **관찰자 중단** 속성값을 찾아보자. 디폴트로 **None**으로 되어 있을 것이다. 우리는 이것을 **Lower Priority**로 수정하자.




<참고> **관찰자 중단** 속성의 이해를 위해서는 약간 복잡한 개념의 이해가 필요하다.
어떤 노드는 실행될 때에 리턴은 바로 되지만 실제적인 작업은 즉시 완료되지 않고 계속해서 수행되는 노드가 있다. **관찰자 중단** 속성은 이러한 실행을 중단하는 것에 관련된 속성이다. 이전의 노드 A의 방문에서는 조건이 부합되어 자신의 노드 A 또는 다른 노드 B가 실행되기 시작했지만, 다음의 노드 A의 방문에서는 조건을 다시 평가한 결과, 조건이 부합되지 않는 경우에 이전의 실행 중인 것을 어떻게 할 것인지의 문제이다.
예를 들어서, **Selector** 노드 아래에 있는 우선 순위가 다른 두 개의 **Task** 중에서 높은 순위의 **Task**의 조건이 맞지 않아서 낮은 순위의 **Task**가 실행되고 있는 중이라고 하자. 그런데 이후에 높은 순위의 **Task**의 조건이 맞게 되면 기존의 실행중인 낮은 순위의 **Task**의 실행을 취소할 것인지 아니면 그대로 둘 것인지의 결정의 문제이다.
지정할 수 있는 옵션으로, 디폴트인 **None**인 경우에는 기존의 실행 모두 중단시키지 않고 그대로 둔다. **Self**인 경우에는 현재의 노드와 그 이하에서의 실행만 중단한다. **Lower Prority**는 오른쪽의 모든 노드에서의 실행을 중단한다. **Both**는 **Self**와 **Lower Priority**를 모두 적용하여 두 경우 모드 중단되도록 한다.

<참고> **관찰자 중단** 속성의 지정에 때라서 노드 실행이 중단될 수 있다. 노드 실행의 중단이 결정되면 중단 대상 **Task**에서 **Receive Abort AI** 이벤트가 호출된다. 따라서, 커스텀 **Task**를 구현할 때에는 **Receive Abort AI** 이벤트 그래프도 구현해서 **Task** 중단 신호가 발생했을 때에 처리해야할 작업을 구현해주도록 해주는 것이 좋다.

**9.** **데코레이터**를 붙인 노드인, 왼쪽의 **Sequence** 노드의 출력바를 드래그하고 **Task**를 배치하자. 선택창에서 **Tasks** 아래에 있는 **Move To** 노드를 배치하자. 이 노드는 **블랙보드**의 지정된 **키**의 액터 또는 위치로 이동하는 노드이다.
이 노드를 선택하고 오른쪽 **디테일** 탭에서 **블랙보드** 영역의 **블랙보드 키**(**Blackboard Key**) 속성값에 **ActorToChase**를 선택하여 지정하자.
이제 **비헤이비어 트리**는 다음의 모습이 될 것이다.




플레이해보자. 이전과 동일하게 실행될 것이다. 실행 중에 **비헤이비어 트리 에디터**에서 실행의 흐름을 관찰해보자.

아무도 **블랙보드**의 **ActorToChase 키**에 값을 지정하지 않기 때문에 왼쪽의 **Sequence** 노드는 항상 실패할 것이므로 실행되지 않을 것이다.

---

이 절에서는 **Selector** 노드와 **데코레이터**의 사용에 대해서 학습하였다.

## 3. 커스텀 서비스 만들고 사용하기

이 절에서 **서비스**의 사용에 대해서 학습한다.

**서비스**는 정보를 수집하는 센서의 역할을 한다. **서비스**는 **데코레이터**와 같이 판단을 하지는 않는다. 또한 **태스크**와 같이 행동을 취하지도 않는다. **서비스**는 단지 정보를 수집하고 수집된 결과를 **블랙보드**에 기록해서 다른 **데코레이터**나 **태스크**에게 정보를 전달해주는 역할을 한다.

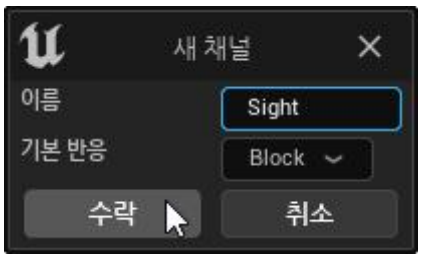
우리의 예에서는 **EnemyPawn**이 플레이어를 발견하면 이를 추격 대상으로 인식하고 발견된 플레이어 정보를 **블랙보드**의 **ActorToChase 키**에 값으로 기록하도록 하자. 이러한 기능을 수행하는 **서비스**를 만들어보자.

이제부터 예제를 통해서 학습해보자

---

**1.** 이전 예제의 프로젝트 `Pbtchase`에서 이어서 계속하자.

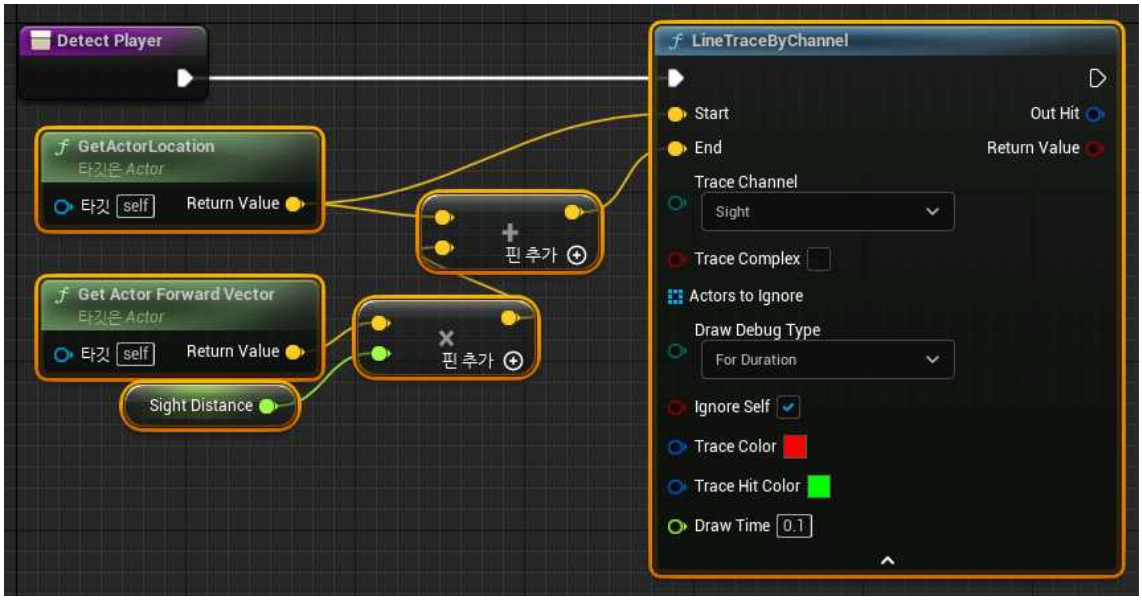
**2.** 이제부터 **EnemyPawn**에서 플레이어를 탐지하는 함수를 만들자.
탐지를 위해서 **트레이스** 기능을 사용할 것이다.
**트레이스** 기능에서 사용할 **트레이스 채널**을 추가하자. **프로젝트 세팅** 창을 열자.
**엔진** » **콜리전** 탭에서 **Trace Channels** 영역에 있는 **새 트레이스 채널** 버튼을 클릭하자. **새 채널** 창이 뜰 것이다. **이름**을 **Sight**로 하고 **기본 반응**은 **Block**으로 그대로 두자. 이제 새 **트레이스 채널**인 **Sight**가 추가되었다.




참고로, 기존의 **프리셋** 목록을 열어보면 새로운 **트레이스 채널**인 **Sight**가 추가되어있을 것이다. 콜리전 반응이 모두 블록 반응이 되도록 설정되었을 것이다. 이것이 부적절한 프리셋들을 찾아서 **블록** 대신 **무시**로 바꾸어주어야 한다. 그러나 우리는 여기에서는 이 과정을 편의상 생략하기로 하자.

**3.** **EnemyPawn** 블루프린트 에디터의 **내 블루프린트** 탭에서 함수를 추가하자. 함수명은 **DetectPlayer**라고 하자. **DetectPlayer** 함수 그래프 탭에서 함수 노드의 출력핀을 드래그해서 **LineTraceByChannel**을 설치하자.
그다음, **LineTraceByChannel** 노드의 **TraceChannel** 입력핀에서 우리가 추가해둔 **트레이스 채널**인 **Sight**를 선택하자.
그다음, **LineTraceByChannel** 노드의 **Start** 입력핀과 **End** 입력핀에 라인 트레이스의 광선의 시작과 끝 위치 벡터를 구해서 연결하자. **Start** 입력핀에는 **GetActorLocation** 노드를 배치하고 이를 연결하자. **End** 입력핀에는 **GetActorForwardVector**의 출력값을 500만큼 곱하여 스케일하고 그 결과를 시작 위치에 더한 벡터를 연결하자.

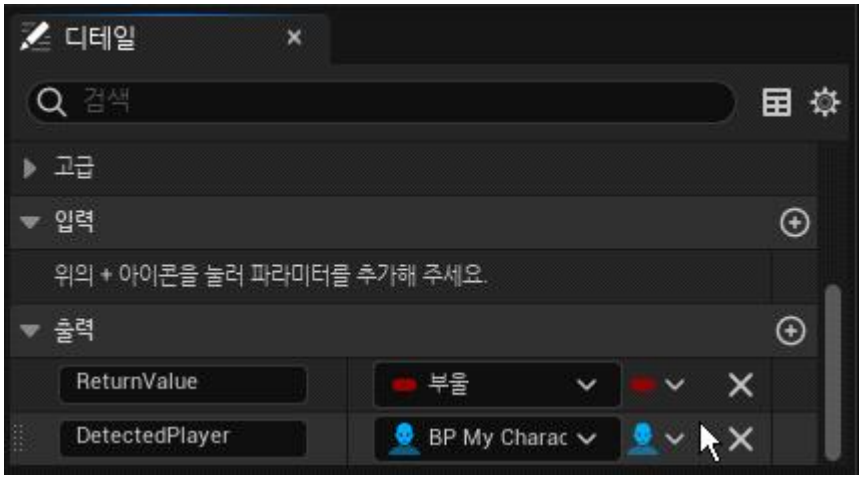
그다음, **곱하기** 노드의 입력핀에서 광선의 길이에 해당하는 상수값인 500을 왼쪽으로 드래그해서 **변수로 승격**을 선택하자. 변수 이름을 **SightDistance**라고 하자.
그다음, **LineTraceByChannel** 노드의 **DrawDebugType** 입력핀에서 **ForDuration**을 선택하자.
그다음, 그 아래의 입력핀을 펼치고 **DrawTime** 입력핀의 값을 디폴트인 5에서 0.1로 수정하자. 이렇게 하면 함수 호출에서 광선이 그려지고 이것이 0.1초 유지되도록 한다.




<참고> 벡터 요소별 곱셈 함수인 * 노드를 배치하면 노드의 두 입력핀의 유형이 모두 벡터 유형일 것이다. 만약 한 입력핀을 다른 유형으로 바꾸고자 한다면 핀 위에서 우클릭하고 **핀 변환** 메뉴를 이용하면 된다.

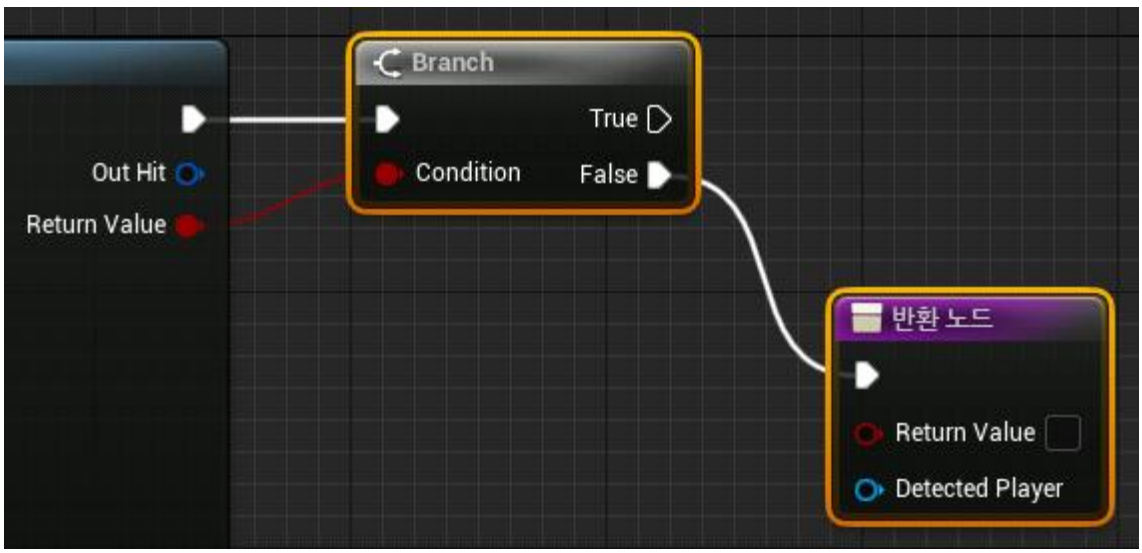
<참고> **LineTraceByChannel** 노드의 **DrawDebugType** 입력핀에 대해서 알아보자.
만약 **DrawDebugType** 입력핀의 값을 **ForOneFrame**으로 한다면 현재 프레임에 광선을 그려라는 뜻이다. 매 프레임마다 이 함수가 호출된다면 항상 광선이 보일 것이다. 한편, 매 프레임마다 이 노드가 실행되지 않고 가끔씩 호출된다면 화면에서 광선이 거의 보이지 않게 된다.
우리는 **비헤이비어 트리**에서 이 함수가 호출되도록 할 것이다. 따라서 **비헤이비어 트리**에서의 실행 반복 주기에 따라 함수가 호출될 것이다. 이 주기는 프레임 그래픽 갱신 주기보다는 매우 느린 속도일 것이다. 따라서 **ForOneFrame**으로 지정하는 것은 적당하지 않다. 우리는 **ForDuration**로 지정하고 **DrawTime**에 **비헤이비어 트리**에서의 반복 주기와 유사한 시간 간격을 지정해주자.

**4.** **DetectedPlayer** 함수의 출력 인자를 2개를 추가하자.
함수 노드를 선택하고 오른쪽의 디테일 탭에서 **출력** 영역의 + 버튼을 클릭하고 함수의 출력 인자를 추가하자. 첫 번째 출력 인자의 이름을 **ReturnValue**로 하고 유형은 **부울(Boolean)**로 두자. 이 출력 인자는 플레이어를 탐지했을 때에 **true**를 리턴하고 그렇지 않은 경우에는 **false**를 리턴하도록 하자. 두 번째 출력 인자를 추가하자. 이름을 **DetectedPlayer**로 하고 유형을 오브젝트 타입의 **BP_MyCharacter** 레퍼런스로 하자.



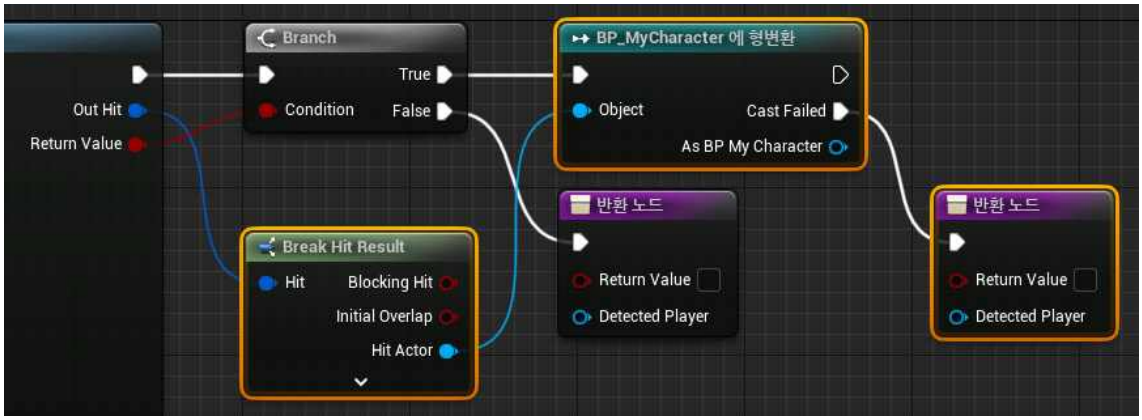

이제 그래프에 **반환 노드**가 배치될 것이다.

**5.** **LineTraceByChannel** 노드의 출력 실행핀을 당기고 **Branch** 노드를 배치하자.
그다음, **LineTraceByChannel** 노드의 **ReturnValue** 출력핀을 **Branch** 노드의 **Condition** 입력핀에 연결하자.
그다음, **Branch** 노드의 **False** 출력 실행핀을 반환 노드의 입력 실행핀에 연결하자. **ReturnValue**에는 체크되지 않은 채로 두어 **false**가 리턴되도록 하자. **DetectedPlayer**에도 아무 연결없이 그대로 두자.




이 그래프에 따라서 **라인 트레이스**에서 충돌이 없는 경우에는 함수가 **false**를 리턴하게 된다.

**6.** **LineTraceByChannel** 노드의 **OutHit** 출력핀을 드래그하고 **BreakHitResult** 노드를 배치하자.
그다음, **BreakHitResult** 노드의 **HitActor** 출력핀을 드래그하고 액션선택 창에서 **BP_MyCharacter**에 형변환 노드를 선택하여 배치하자.
그다음, **Branch** 노드의 **True** 출력 실행핀을 형변환 노드의 입력 실행핀에 연결하자.
그다음, 기존의 반환 노드를 **Ctrl+C**를 누르고 **Ctrl+V**를 눌러 복사한 후에, 형변환 노드의 **Cast Failed** 출력핀을 복사된 반환 노드의 입력 실행핀에 연결하자.
그다음, **LineTraceByChannel** 노드의 가장 아래의 접기 아이콘을 클릭하여 노드를 요약 모습으로 바꾸자.

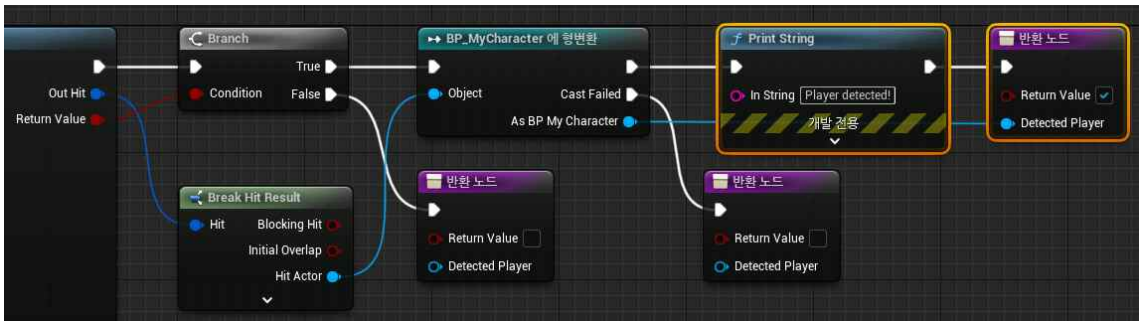



**7.** 형변환 노드의 출력 실행핀에 실행 펄스가 흐르게 되면 이는 충돌한 액터가 플레이어 액터임을 의미하는 것이다. 형변환 노드의 출력 실행핀을 드래그하여 **PrintString** 노드를 배치하자. **InString** 입력핀에 '`Player detected!`'라고 입력하자.

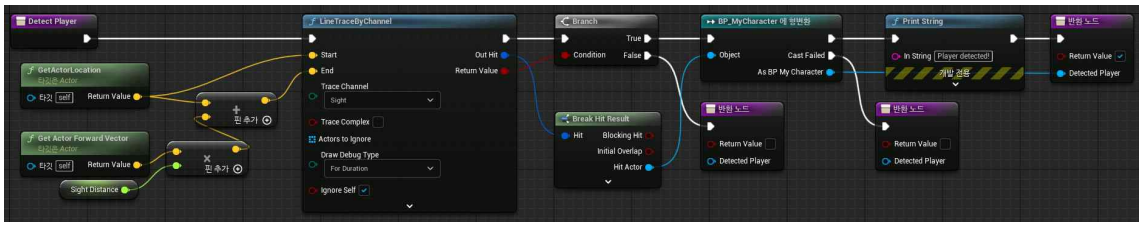
그다음, 기존의 반환 노드를 **Ctrl+C**를 누르고 **Ctrl+V**를 눌러 복사하자. **PrintString** 노드의 출력 실행핀을 복사된 반환 노드에 연결하자.

그다음, 반환 노드의 **ReturnValue** 입력핀에 체크하여 **true**가 리턴되도록 하자.

그다음, 형변환 노드의 **As BP_MyCharacter** 출력핀을 드래그하여 반환 노드의 **DetectedPlayer**에 연결하자.




**8.** 이제 **DetectPlayer** 함수가 완성되었다.



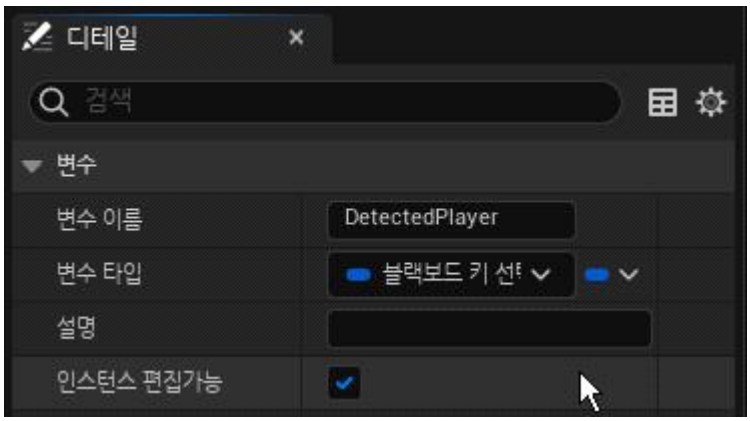
**9.** 지금부터 **서비스**를 만들어보자.

먼저, **MyFirstBehaviorTree**의 **비헤이비어 트리 에디터**로 이동하자. 툴바의 새 **서비스** 버튼을 클릭하자.


저장할 애셋 이름을 묻는 창이 뜰 것이다. 이름이 디폴트로 **BTService_BluePrintBase_New**로 나타날 것이다. 우리는 애셋의 이름을 **BTService_PlayerDetector**로 수정하자.
**콘텐츠 브라우저**의 현재 폴더로 가보면 **서비스** 블루프린트 애셋 파일이 생성되어 있을 것이다.

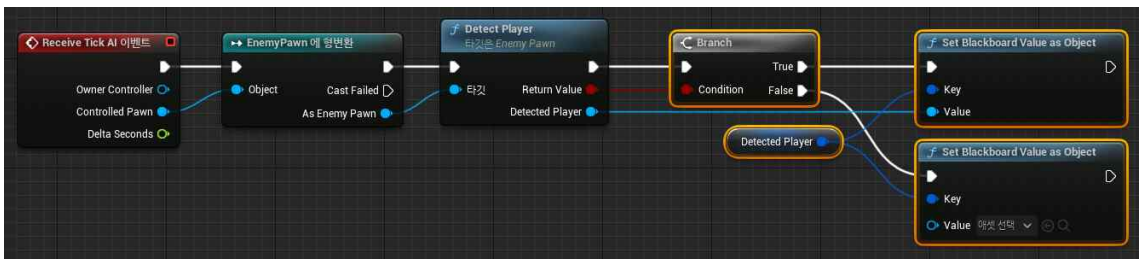
<참고> 모든 **서비스** 블루프린트 클래스의 가장 상위의 기반 클래스는 **BTService_BlueprintBase**이다.아직 생성한 **서비스**가 없는 상태이므로 부모 클래스 선택 창이 뜨지 않고 디폴트로 바로 생성해준다. 다음에 **서비스**를 만들 때에 부모 선택 창이 뜨는 경우에 특별한 경우가 아니라면 매번 **BTService_BlueprintBase**를 부모 클래스로 선택하면 된다.

**10.** 지금부터 **BTService_PlayerDetector 서비스** 블루프린트를 작성해보자.
**서비스**가 생성되면 **서비스** 블루프린트 에디터 창이 열릴 것이다. 열리지 않는다면 서비스 애셋 아이콘을 더블클릭하면 된다.
먼저 왼쪽 **내 블루프린트** 탭에서 변수를 추가하자.
변수 이름은 **DetectedPlayer**로 하자. 변수 유형은 **구조체** 아래의 **블랙보드 키 선택 툴(Blackboard Key Selector)**로 하자.
**인스턴스 편집 가능**에 체크하자. 이렇게 하면 **비헤이비어 트리 에디터**에서 디폴트 값을 지정할 수 있게 된다.



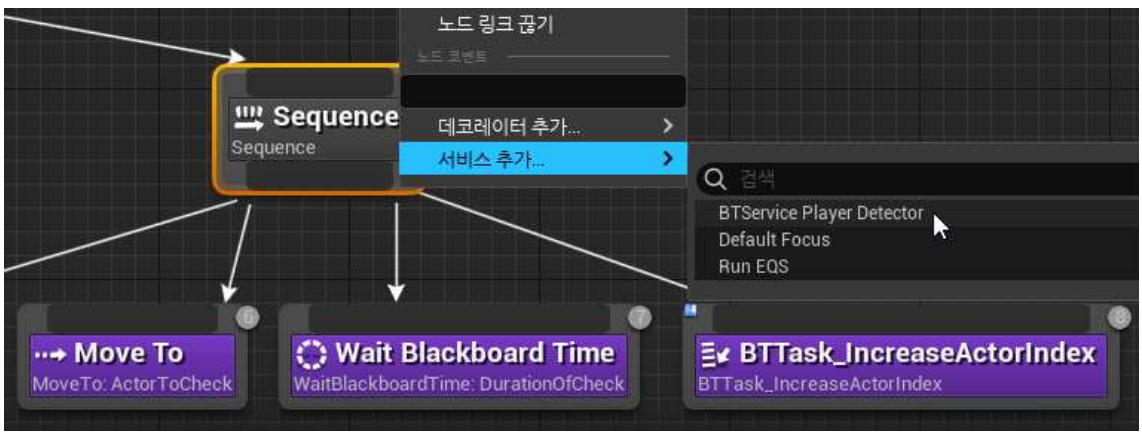

**11.** **BTService_PlayerDetector 서비스** 블루프린트에서 이벤트 그래프를 작성해보자.
격자판의 빈 곳에서 우클릭하고 **ReceiveTickAI 이벤트** 노드를 배치하자.
그다음, **ReceiveTickAI 이벤트** 노드의 **ControlledPawn** 출력핀을 드래그하고 **EnemyPawn에 형변환** 노드를 배치하자.
그다음, 형변환 노드의 **As EnemyPawn** 출력핀을 드래그하고 **DetectPlayer** 함수 호출 노드를 배치하자.


**12.** 그다음, **DetectPlayer** 함수 노드의 **ReturnValue** 출력을 드래그하고 **Branch** 노드르 배치하자.
그다음, **Branch** 노드의 **True** 출력 실행핀을 드래그하고 **SetBlackboardValue asObject** 함수 호출 노드를 배치하자.
그다음, **내 블루프린트** 탭에서 **DetectedPlayer** 변수를 드래그해서 **Get** 노드를 배치하고, **Get** 노드의 출력을 **SetBlackboardValue asObject** 노드의 **Key** 입력핀에 연결하자.
그다음, **DetectPlayer** 노드의 **DetectedPlayer** 출력핀을 **SetBlackboardValue asObject** 노드의 **Value** 입력핀에 연결하자.
그다음, **SetBlackboardValue asObject** 노드를 **Ctrl+C**를 누르고 **Ctrl+V**를 눌러 복사하자. 그리고, **Branch** 노드의 **False** 출력 실행핀을 복사된 노드의 입력 실행핀에 연결하자.
그리고, **DetectedPlayer** 변수의 **Get** 노드를 복사된 노드의 **Key** 입력핀에 연결하자. 복사된 노드의 **Value** 입력핀에는 아무 연결없이 그대로 두자.



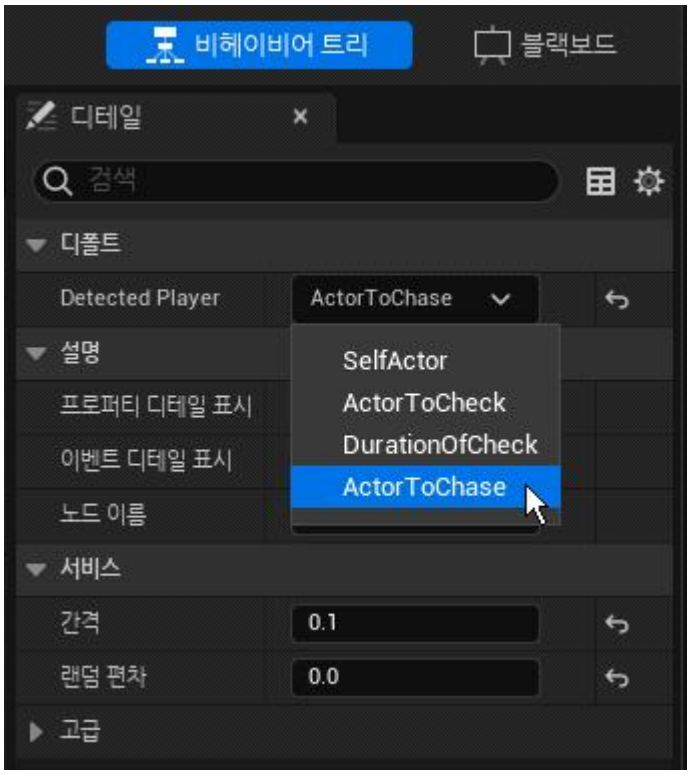

이제 **BTService_PlayerDetector 서비스** 블루프린트가 모두 완성되었다.
컴파일하고 저장하자.

**13.** **비헤이비어 트리 에디터**로 가자.
오른쪽 **Sequence** 노드에서 우클릭하고 **서비스 추가**를 선택하자. 목록에서 **BTService_PlayerDetector**를 선택하자.



Sequence 노드 안에 녹색의 서비스가 추가될 것이다.

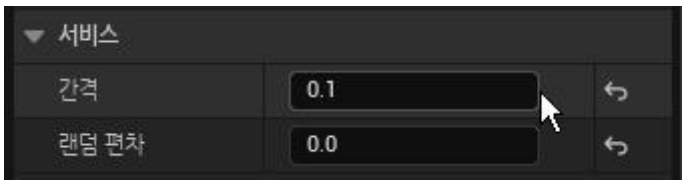
**14.** 추가된 **서비스**를 선택하고 오른쪽 **디테일** 탭에서 **디폴트** 영역의 **DetectedPlayer** 속성을 찾자. 속성 값을 디폴트인 **SelfActor**에서 **ActorToChase**로 수정하자.



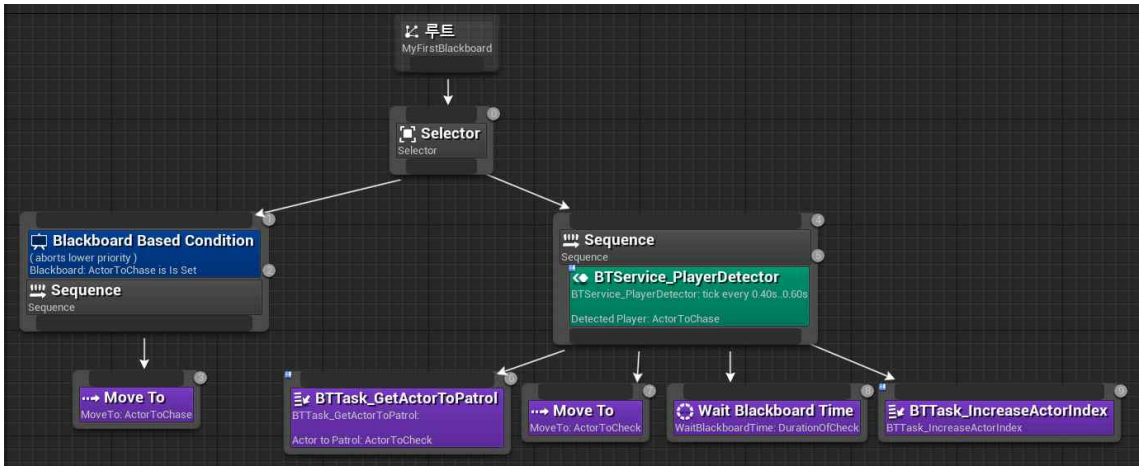

이제 **서비스**에서 **DetectedPlayer** 변수에 기록되는 값은 **블랙보드**의 **ActorToChase**에 기록될 것이다.

**15.** 또한, 그 아래의 **서비스** 영역에 **간격(Interval)** 속성이 있다. 디폴트값인 0.5를 0.1로 수정하자. 이것은 **서비스**의 틱이 발생하는 시간 간격이다. 얼마나 자주 **서비스**를 실행시켜 탐지 작업을 반복할지를 명시한다.

그리고 그 아래의 **랜덤 편차(RandomDeviation)** 속성값이 있다. 이것은 틱 간격에 임의의 편차를 주기 위한 속성값이다. 틱 간격이 주어진 범위 내에서 랜덤하게 되도록 하여 시간 간격이 불규칙하게 되도록 한다. 우리는 테스트를 위하여 쉽게 확인되도록 디폴트값인 0.1을 0으로 수정하자.




**16.** 이제 **비헤이비어 트리**가 완성되었다.




**17.** 플레이해보자.

플레이하면서 **비헤이비어 트리 에디터** 창에서의 실행 펄스 흐름을 관찰해보자.

**EnemyPawn**의 이동 동선으로 움직여서 탐지되도록 해보자.

오른쪽 **Sequence** 노드의 서비스에서 플레이어 액터가 탐지되면 **ActorToChase**에 기록될 것이다. 그리고 다음번 반복에서 왼쪽의 **Sequence** 노드의 **데코레이터**가 만족되어 왼쪽 아래의 **태스크** 노드가 실행될 것이다.

---

이 절에서는 **서비스**를 생성하고 사용하는 방법을 학습하였다.

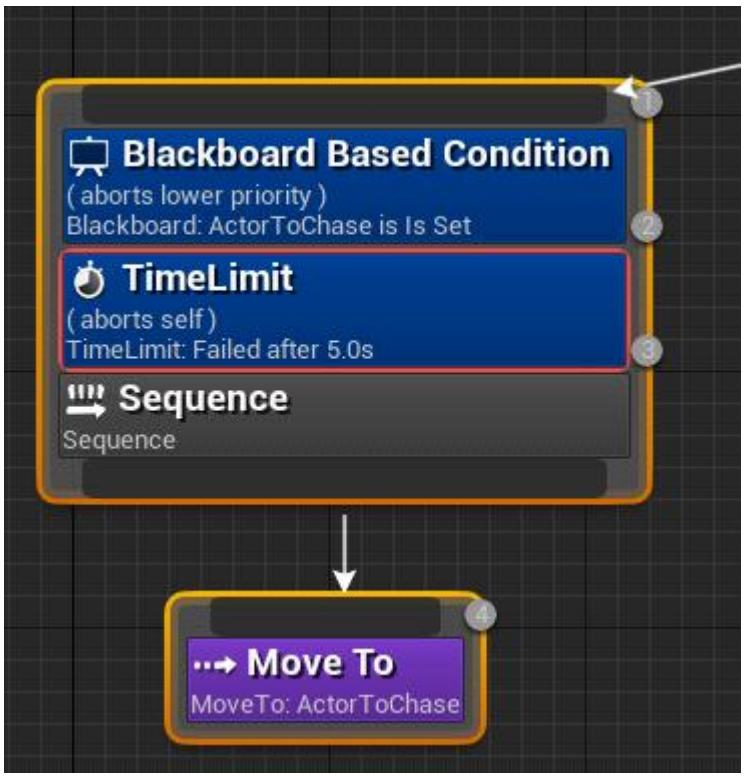
## 4. 커스텀 데코레이터 만들고 사용하기

이 절에서 커스텀 **데코레이터**를 만들고 사용하는 방법을 학습한다.
이제부터 예제를 통해서 학습해보자

---

**1.** 이전 예제의 프로젝트 `Pbtchase`에서 이어서 계속하자.

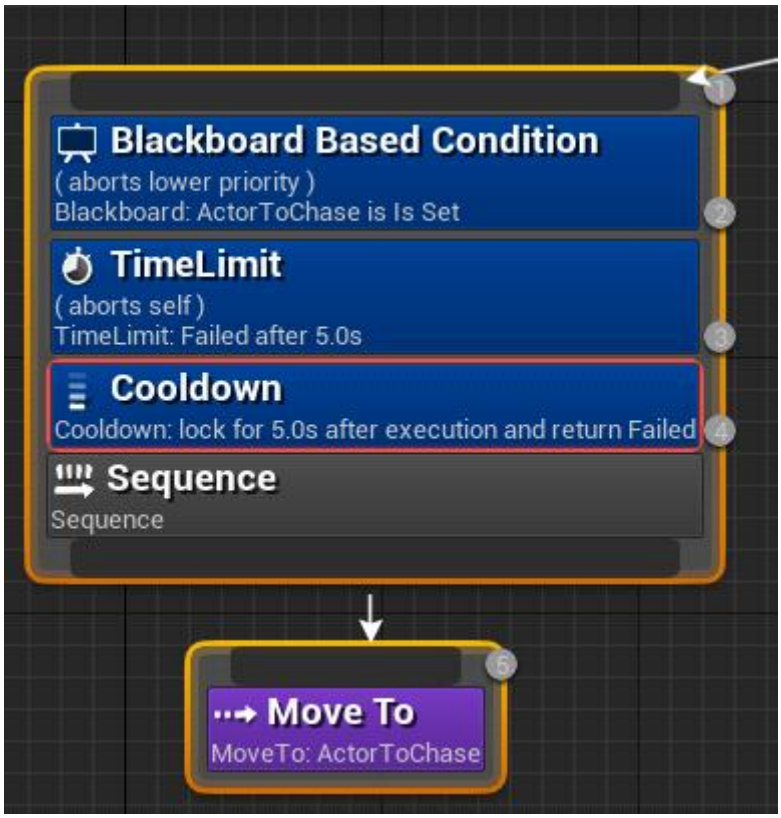
**2.** 커스텀 **데코레이터**를 만들어보기 전에 엔진이 제공하는 **데코레이터**에 대해서 더 알아보자.
**데코레이터** 중에 **Time Limit**가 있다. 이 **데코레이터**를 노드에 부착해두면 지정된 시간이 경과된 후에 해당 노드가 강제로 실패되도록 만든다.
이제 실제로 적용해보자.
**콘텐츠 브라우저**에서 **MyFirstBehaviorTree**를 더블클릭하여 **비헤이비어 트리 에디터**를 열자.
그다음, 왼쪽의 **Sequence** 노드에서 우클릭하고 **데코레이터 추가**를 선택하고 목록에서 **Time Limit**를 선택하여 배치하자.
그다음, **Time Limit** 데코레이터를 클릭하여 선택하고 오른쪽의 **디테일** 탭의 **시간 제한(TimeLimit)** 속성을 찾아보자. 속성값이 디폴트로 5로 되어 있을 것이다. 따라서 **Squence** 노드의 실행이 시작되어 **EnemyPawn**이 플레이어를 추적하기 시작하고 5초가 경과한 후에는 **Sequence** 노드가 실패하게 된다.




한편, 이 **Sequence** 노드가 실패하여 부모 노드로 리턴하게 되더라도 다시 바로 이 **Sequence** 노드가 실행된다. 왜냐하면 **ActorToChase**에 명시되어 있는 액터가 있어서 **Blackboard Based Condition** 데코레이터가 여전히 참이 되기 때문이다.

**3.** 이번에는 다른 **데코레이터**를 알아보자.
**Cooldown** 데코레이터가 있다. 이 **데코레이터**를 노드에 부착해두면 노드가 성공적으로 실행되고 리턴

된 후에는 지정된 시간 동안에는 다시 진입할 수 없도록 막는다.
이제 실제로 적용해보자.
왼쪽의 **Sequence** 노드에서 우클릭하고 **데코레이터 추가**를 선택하고 목록에서 **Cooldown**을 선택하여 배치하자.
그다음, **데코레이터**를 클릭하여 선택하고 오른쪽의 **디테일** 탭의 **CoolDownTime** 속성을 찾아보자. 속성 값이 디폴트로 5로 되어 있을 것이다. 이는 **Sequence** 노드가 실행 조건을 만족해서 성공적으로 실행되고 리턴된 후에는 이 시간 동안 다시 진입할 수 없도록 막는다. 또한, **Sequence** 노드가 성공적으로 실행되고 리턴될 때에 **Blackboard Based Condition** 데코레이터가 체크하는 조건도 성공되지 못하도록 **블랙보드**의 값을 리셋한다. 따라서 **ActorToChase**의 값이 리셋된다.

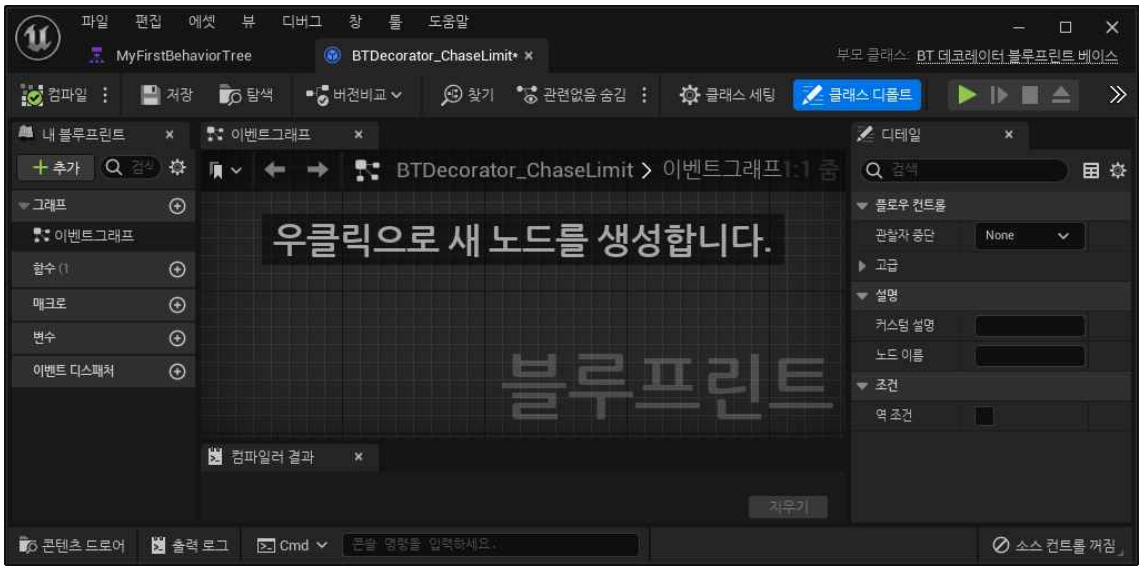



**4.** 이제부터 커스텀 **데코레이터**를 만들어보자.
적이 플레이어를 끝까지 추격하도록 하는 것은 적절하지 않다. 적이 추격하다가 어느 정도 후에는 포기하도록 해보자. 적이 추격을 시작할 때에 추격 시작 시의 위치를 기억해두자. 추격 시작 위치로부터 일정 거리 이상으로 멀어진 위치에 오게 되면 추격을 포기하도록 해보자. 즉 거리를 계산하고 거리가 일정 수준이 넘는 경우에 실패하도록 하는 데코레이터를 만들어보자.
먼저, **비헤이비어 트리 에디터**의 툴바에서 **새 데코레이터** 버튼을 클릭하자.

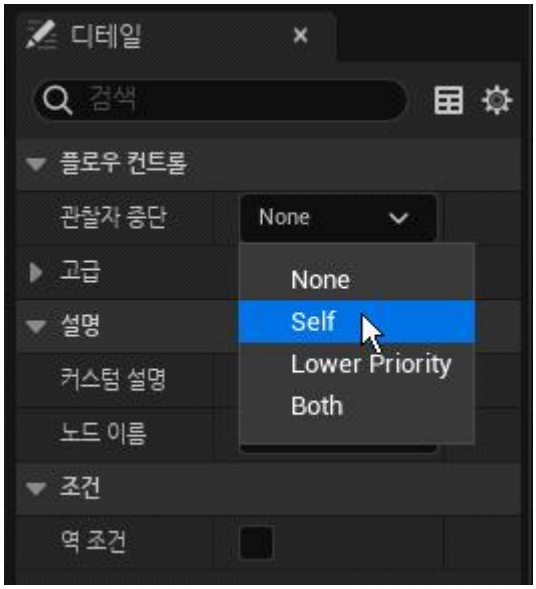

**새 데코레이터** 버튼을 클릭하면 저장할 애셋 이름을 묻는 창이 뜬다.
이름이 디폴트로 **BTDecorator_BlueprintBase_New**로 되어 있을 것이다. 우리는 이름을 **BTDecorator_ChaseLimit**로 수정하자.

<참고> 모든 **데코레이터** 블루프린트 클래스의 가장 상위의 기반 클래스는 **BTDecorator_BlueprintBase**이다.아직 생성한 **데코레이터**가 없는 상태이므로 부모 클래스 선택 창이 뜨지 않고 디폴트로 바로 생성해준다. 다음에 **데코레이터**를 만들 때에 부모 선택 창이 뜨는 경우에 특별한 경우가 아니라면 매번 **BTDecorator_BlueprintBase**를 부모 클래스로 선택하면 된다.

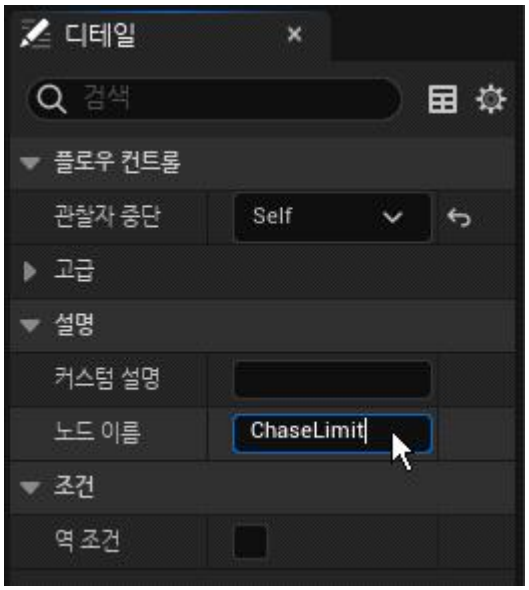
**5.** **BTDecorator_ChaseLimit** **데코레이터**가 생성되면 **데코레이터**의 블루프린트 에디터 창이 뜬다.
만약 창이 뜨지 않는다면 **콘텐츠 브라우저**에 가서 데코레이터 애셋 아이콘을 더블클릭하면 된다.




**6.** **BTDecorator_ChaseLimit** 블루프린트 에디터에서 디테일 탭의 **플로우 컨트롤** 영역의 **관찰자 중단**을 디폴트값인 **None**에서 **Self**로 수정하자. 이렇게 **Self**로 설정하면 해당 **데코레이터**의 부착된 노드의 실행을 도중에 중단시킨다. 우리의 예제에서 플레이어를 추격하는 **MoveTo** 태스크 노드의 실행을 중도에 중단시키게 된다.

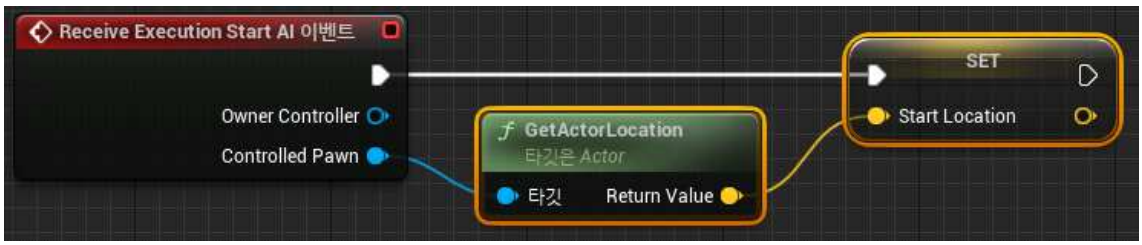



**7.** 또한, **디테일** 탭의 아래에 **설명** 영역에 **Node Name** 속성이 있다. 이 속성은 **비헤이비어 트리**에서 **데코레이터**가 노드 내에서 표시되는 이름이다. 만약 이 속성을 비워두게 되면 디폴트로 클래스 이름인 **BTDecorator_ChaseLimit**가 그대로 사용되어서 복잡해 보이게 된다. 우리는 **ChaseLimit**이라고 입력하자.

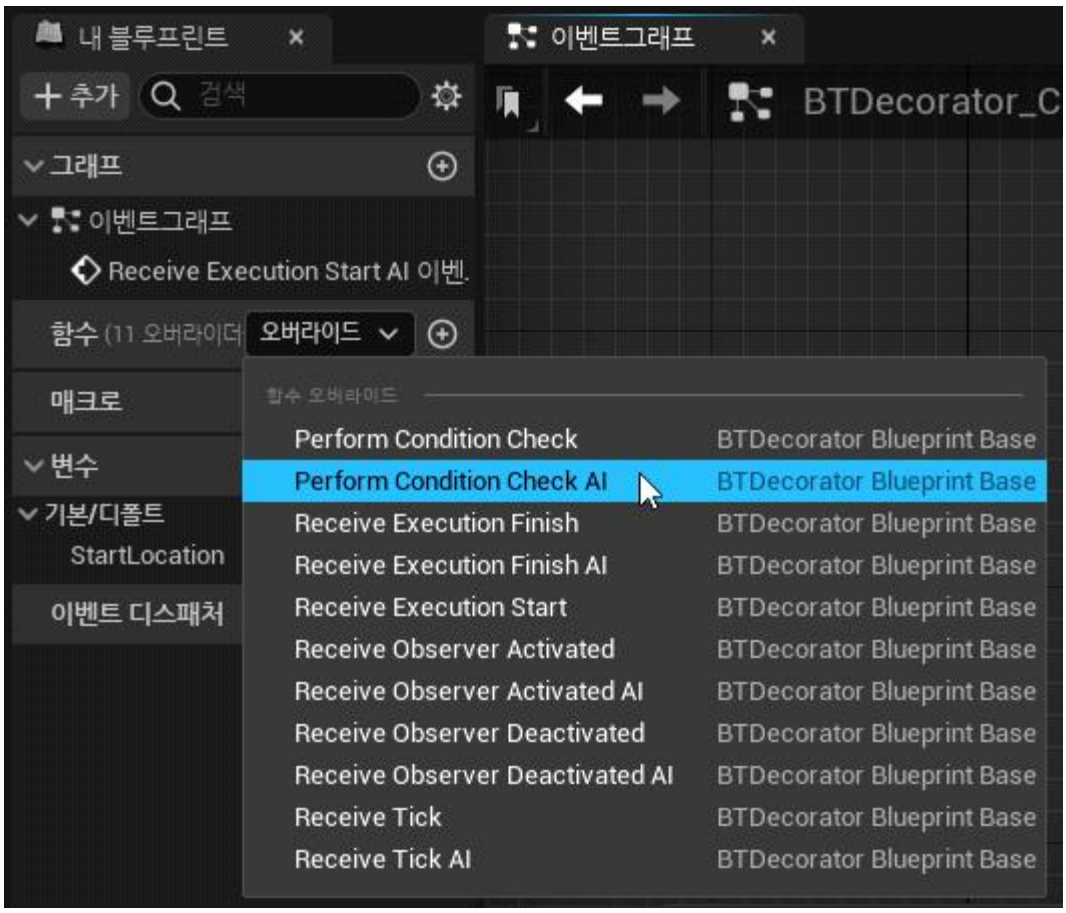



**8.** **내 블루프린트** 탭에서 변수를 추가하자. 변수명을 **StartLocation**으로 하고 변수 유형을 **벡터(Vector)**로 하자.
그다음, 이벤트 그래프에서, **ReceiveExecutionStartAI** 이벤트 노드를 추가하자. 이 이벤트 노드는 노드의 실행이 시작될 때에 호출된다. 즉 추격이 시작될 때에 호출되므로 이 이벤트 그래프에서 자신이 제어하는 폰의 현재의 위치를 기억해두면 된다.
그다음 이벤트 노드의 **ControlledPawn** 출력핀을 드래그하고 **GetActorLocation** 노드를 배치하자.
그다음, 변수 **StartLocation**의 **Set** 노드를 배치하고 **GetActorLocation** 노드의 출력핀을 **Set** 노드에 연결하자. 그리고, 이벤트 노드의 실행핀을 **Set** 노드에 연결하자.

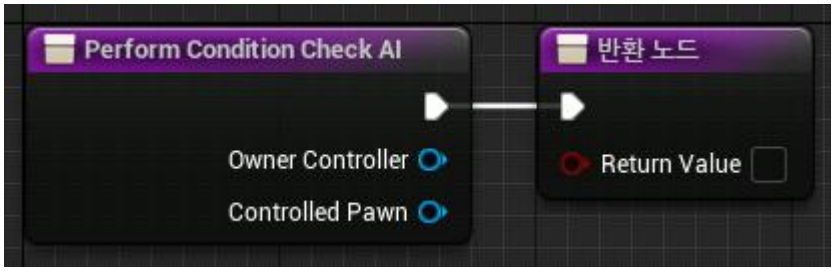
다음과 같은 그래프가 될 것이다.




**9.** 이제 추격을 중단할지를 판단하는 구현을 해보자.
**데코레이터**에서는 자신이 부착된 노드가 실행되는 중에 노드의 실행을 계속할지 또는 중단할지를 판단하는 함수를 제공한다. 이 함수의 이름은 **PerformConditionCheckAI** 함수이며, **데코레이터**의 부모 클래스인 **BTDecorator_BlueprintBase**의 가상함수로 구현되어 있다. 따라서, 우리는 부모 클래스의 가상 함수를 재정의해서 우리의 커스텀 **데코레이터**에서의 **PerformConditionCheckAI** 함수를 구현하면 된다. **내 블루프린트** 탭에서 함수 추가 아이콘 바로 왼쪽으로 마우스를 옮기면 **오버라이드** 리스트박스가 있다. 이것을 클릭하고 **PerformConditionCheckAI** 함수를 선택하여 배치하자.




**10.** **PerformConditionCheckAI** 함수 그래프 창이 보일 것이다. 다음과 같은 그래프가 보일 것이다. 함수에는 **Bool** 값을 리턴하는 반환 노드가 있다.




**11.** 그래프를 작성해보자.

먼저, **PerformConditionCheckAI** 함수 노드에서 **ControlledPawn** 출력핀을 드래그하고 **GetActorLocation** 노드를 배치하자.

그다음, 변수 **StartLocation**의 **Get** 노드를 배치하자.

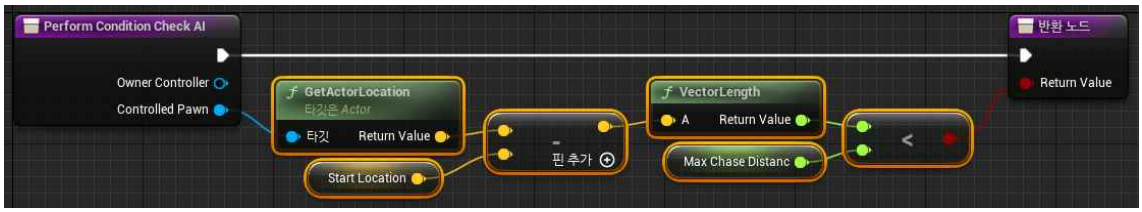
그다음, **GetActorLocation** 노드의 출력값과 **Get** 노드의 출력값의 차이를 구하는 **-** (**빼기**) 노드를 배치한다.

그다음, **VectorLength** 노드를 통해 벡터 길이를 계산하자.

그다음, 비교 연산자 **<** (**작음**) 노드를 배치하고, 길이가 200보다 작으면 **true**를 리턴하도록 하고 그렇지 않으면 **false**를 리턴하도록 하자. 길이에 더 큰 값을 설정해야 하겠지만 우리는 테스트를 쉽게 하도록 짧게 명시하였다.

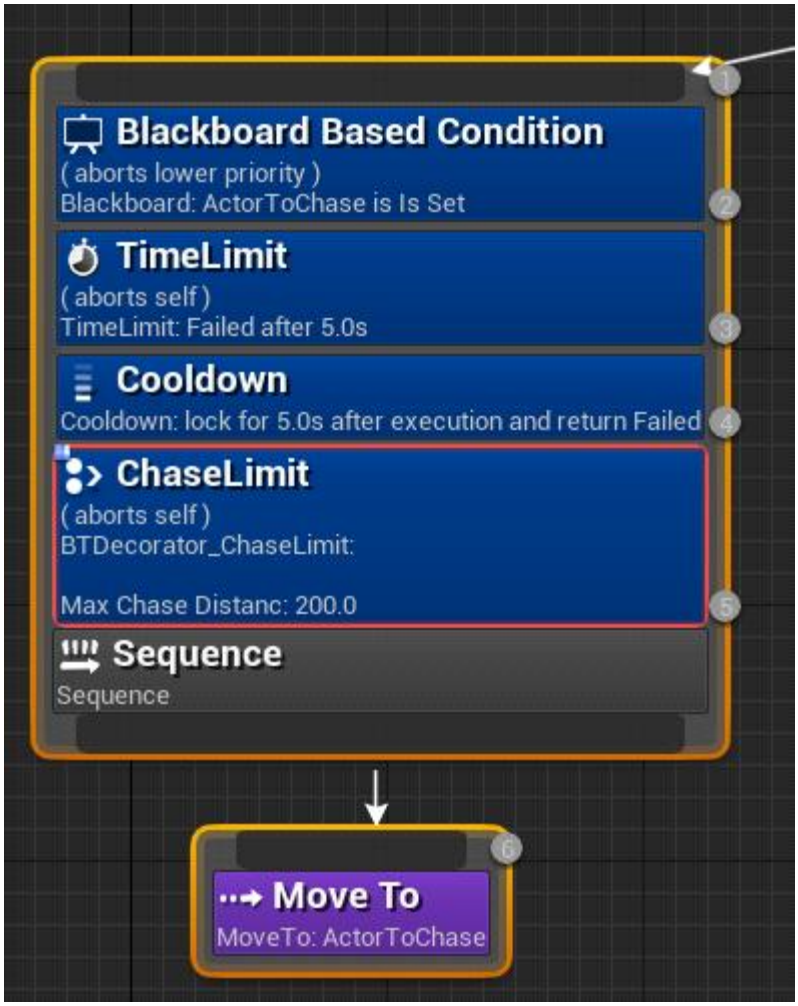
그다음, 상수값 200을 왼쪽으로 드래그하고 변수로 승격을 선택하여 변수를 추가하자. 변수 이름은 **MaxChaseDistance**로 지정하자. 그리고, **디테일** 탭에서 **인스턴스 편집 가능**에 체크하자. 체크해두면 **비헤이비어 트리 에디터**에서 디폴트 값을 설정할 수 있게 된다.

그래프가 다음과 같이 작성되었을 것이다.

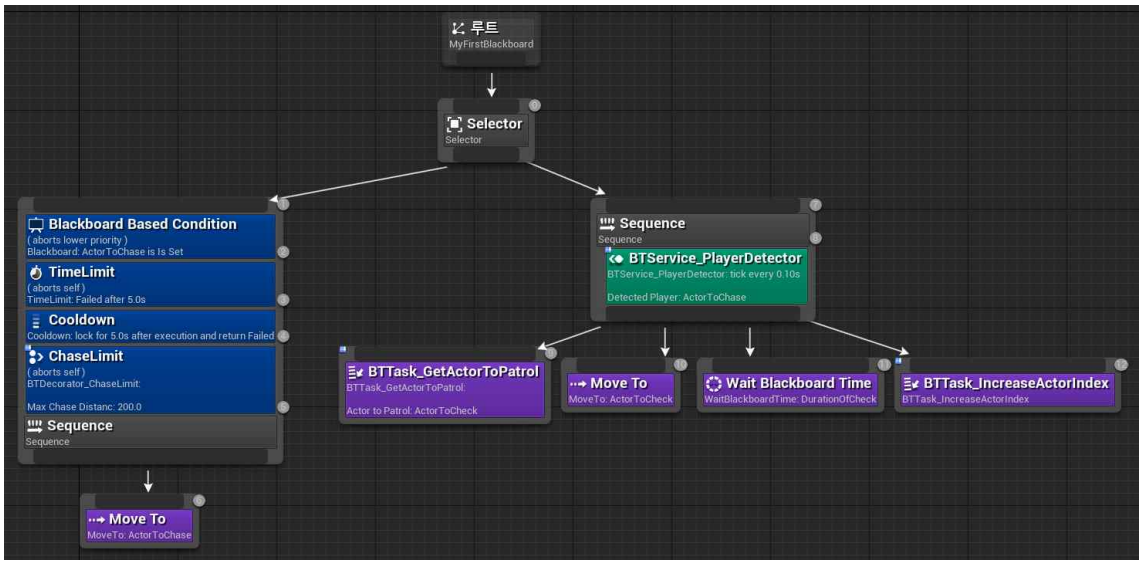



**12.** **비헤이비어 트리 에디터**로 가자.

플레이어를 추격하는 왼쪽의 **Sequence** 노드에 우리가 만든 **데코레이터**인 **BTDecorator_ChaseLimit**를 추가하자. 우리가 노드 이름을 **ChaseLimit**로 입력해두었으므로 입력된 노드 이름이 보일 것이다.

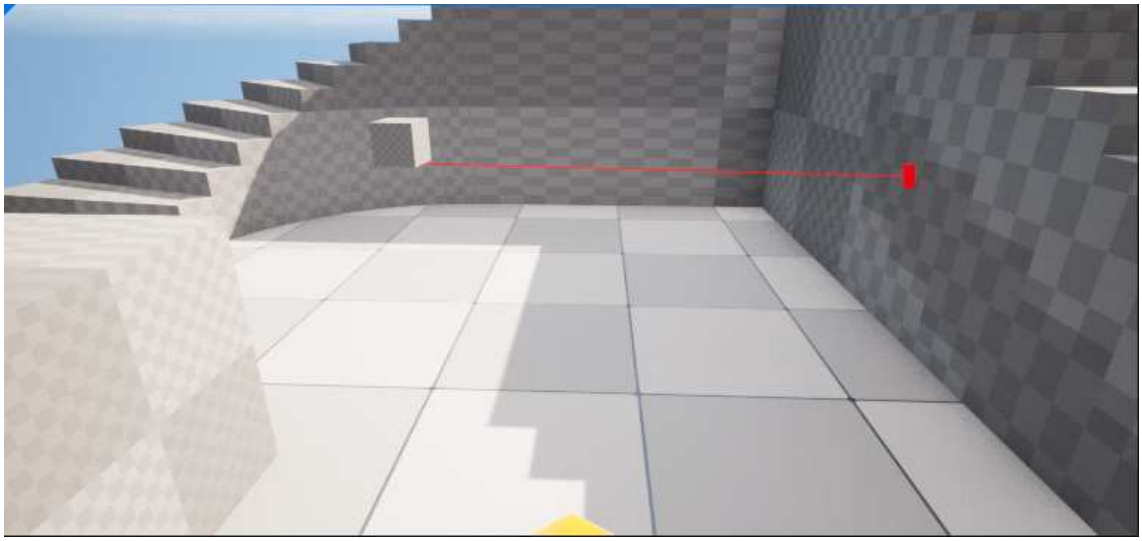



**13.** 이제 전체적으로 다음과 같은 **비헤이비어 트리**가 완성되었다.




**14.** 플레이해보자.

적이 추격해올 때에 달아나면 적이 추격을 포기할 것이다.



이 절에서는 커스텀 **데코레이터**를 만들고 사용하는 방법을 학습하였다.

□