

Đưa ra các lựa chọn cho AI

Giải thích cách giới thiệu hành vi tự trị cho các nhân vật của chúng ta bằng cách sử dụng Behavior Trees. Behavior Tree là một phương pháp cho phép bạn xây dựng logic AI của mình một cách trực quan trong cấu trúc cây và có thể được sử dụng lại trong các ký tự khác nhau.

Tags: Đưa ra các lựa chọn cho AI

Trong chương này, bạn sẽ học cách giới thiệu hành vi tự chủ cho các nhân vật của mình bằng Behavior Tree. Behavior Tree là một phương pháp cho phép bạn xem logic AI của mình một cách trực quan. Behavior Tree là một loại mạng tác vụ phân cấp cho thiết kế hướng trạng thái. Vì vậy, mỗi trạng thái sẽ ra lệnh cho nhiệm vụ hiện tại của chúng ta thay vì một mục tiêu.

Chương này sẽ bao gồm:

- Behavior Tree
- Blackboard
- Các component của Behavior Tree, bao gồm Selector, Decorator, Service, v.v.
- Xây dựng Behavior Tree để chạy trên nhân vật con chó.

Download tập tin full source của chương này, [tại đây](#).

Behavior Tree trong AIController

Trong chương này, chúng ta sẽ sử dụng Behavior Tree và các script lệnh để tạo hành vi định hướng trạng thái tự trị của chúng ta. Tuy nhiên, trước khi chúng ta kiểm soát AI ở cấp độ cao hơn, hãy hiểu một số thành phần cơ bản cho phép chúng ta kiểm soát AI của mình. Vì vậy, ngay từ đầu, chúng ta có AIController, tương tự như PlayerController; Controller này chịu trách nhiệm diễn giải tất cả đầu vào AI của chúng ta. Đầu vào này được thế giới áp dụng khi chúng ta yêu cầu nó được di chuyển.

Với suy nghĩ này, chúng ta có thể giới thiệu influence thông qua nhiều con đường trong mã code. Chúng ta có thể yêu cầu AIController di chuyển đến một vị trí hoặc chúng ta có thể yêu cầu AIController chạy Behavior Tree. Điều nữa rất quan trọng cần hiểu là chuyển động được áp dụng với component CharacterMovement. Nếu bạn đã tạo một subclass từ CharacterMovement, bạn có thể mở rộng và tiếp tục sử dụng cùng một Behavior Tree để giới thiệu chuyển động theo lý thuyết.

Component chuyển động bên trong chịu trách nhiệm thực hiện yêu cầu *Move To* từ Behavior Tree. Vì vậy, điều này cho phép tôi ra lệnh cho ô tô của mình đi đến một số địa điểm nhất định, biết khi nào nó kết thúc và chỉ cần lặp lại quy trình. Điều này rất hiệu quả nếu bạn đang cố gắng tạo AI vì phần lớn những gì bạn cần làm là di chuyển.

Tạo ra vài thứ Behavior Tree cần

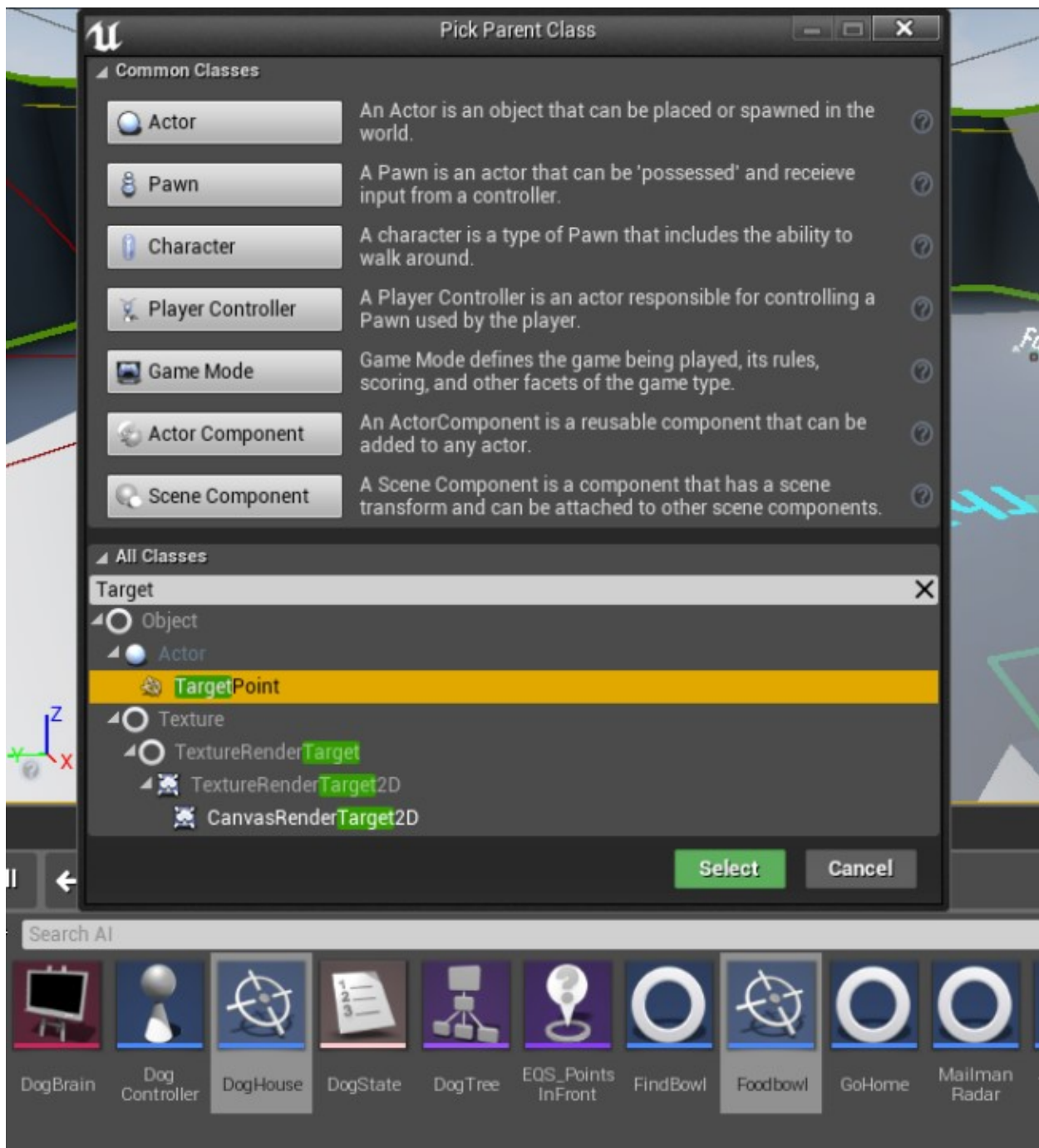
Chuyển sang Behavior Tree, hãy bắt đầu xây dựng cây đầu tiên của chúng ta! Trong dự án này, chúng ta sẽ tạo ra AI giống với hành vi của một con chó hàng xóm. Nó sẽ thay đổi trạng thái hành vi của mình một cách ngẫu nhiên thường xuyên và tiếp tục tìm kiếm bất kỳ người

đưa thư nào có thể lảng vảng gần chỗ ở của con chó.

Điều này sẽ đề cập đến việc sử dụng **Environment Query System** (EQS), đây vẫn là một tính năng thử nghiệm nhưng mạnh mẽ trong Unreal Engine 4. Nó sẽ chịu trách nhiệm giao cho chú chó một vị trí mới để tìm kiếm người đưa thư. Vì vậy, nếu bạn đã sẵn sàng, hãy mở một dự án Third Person mới.

Hãy tạo một vài thứ để con chó của chúng ta định tuyến. Sau đây là các bước để thực hiện:

1. Trước tiên, hãy tạo một class **Target Point** tự tạo mới và đặt tên là *Foodbowl*.
2. Tiếp theo, hãy tạo một class **Target Point** tự tạo mới và đặt tên là *DogHouse*:



3. Đặt hai actor này một cách phù hợp trong màn chơi. Chúng ta sẽ sử dụng những thông tin này để tham khảo nơi chú chó của chúng ta sẽ chuyển đến khi có trạng thái thích hợp. Vì vậy, về cơ bản, chúng ta sẽ sử dụng **Target Point** hoặc Waypoint để tham chiếu vị trí vector của actor này trong blueprint sau này.
4. Bây giờ, chúng ta cần một class **Enumeration** mới trong context-menu **Blueprint** > **Enumeration** gọi là *DogState*; Nó sẽ giữ trạng thái hiện tại của chú chó của chúng ta. Giá trị này rất quan trọng để thực hiện đúng cây Behavior Tree cho các phản hồi thích hợp mà chúng ta gửi tới AI, chẳng hạn như phát hiện người đưa thư gần đó.
5. Class *DogState* sẽ có ba trạng thái có sẵn là *Hungry*, *Barking* và *Idle*. *Hungry* sẽ chịu trách

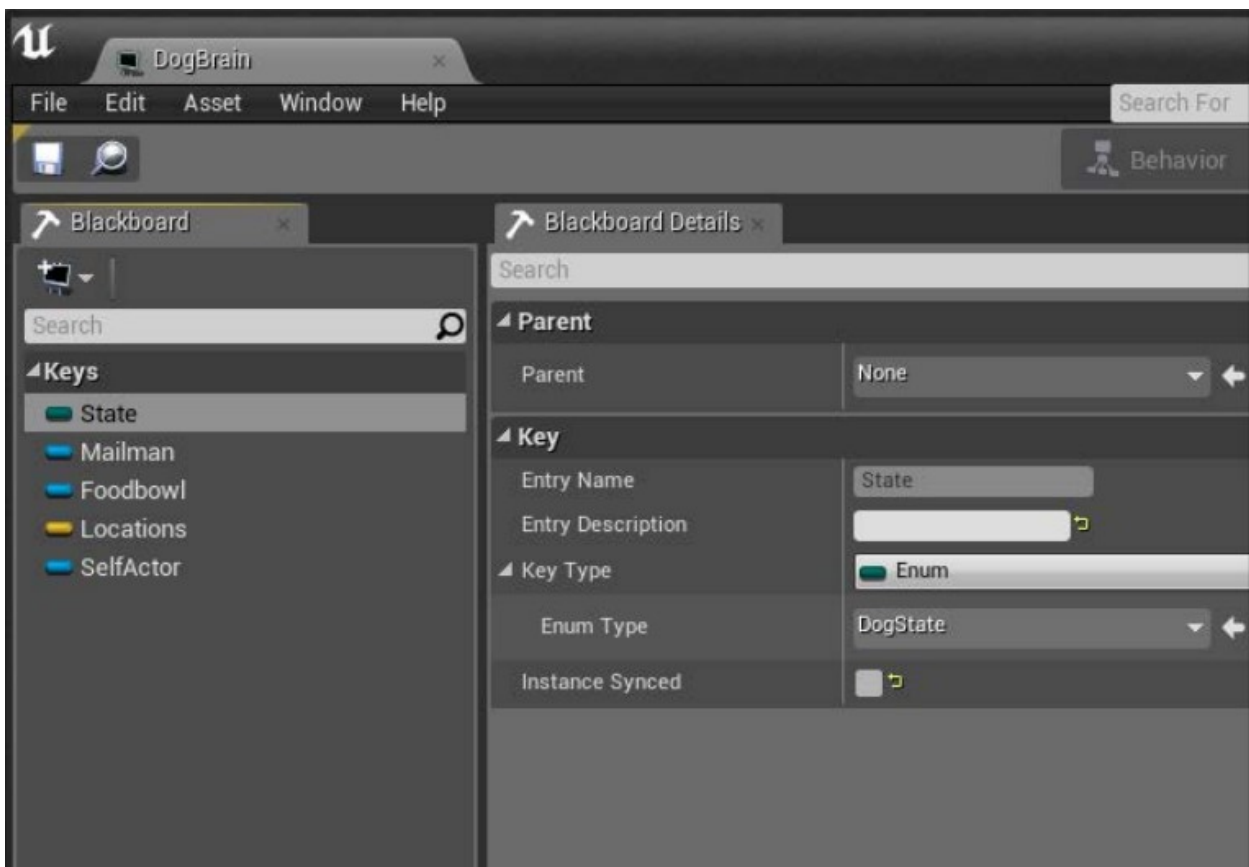
nhiệm nói cho chú chó tìm đến bát của chó để ăn thức ăn. Trạng thái *Barking* sẽ bảo chú chó đi tìm người đưa thư. Trạng thái *Idle* sẽ bảo chú chó đi đến chuồng chó của nó. Bất cứ lúc nào, chú chó sẽ bắt đầu đuổi theo người đưa thư nếu anh ta đến quá gần:



Tạo ra Blackboard

Blackboard là một trong những thứ Behavior Tree cần dùng. Trong đó **Blackboard** sẽ lưu bộ nhớ cho các agent hoặc một agent riêng lẻ nào đó. Nó hoạt động song song với **Behavior Tree**. Nó cho phép truy cập dễ dàng và trực tiếp vào các biến từ bất kỳ node **Task** nào của bạn. Hãy tưởng tượng rằng **Behavior Tree** của bạn là **EventGraph** và **Blackboard** là biến mà bạn sử dụng trong **EventGraph**. Với chức năng đồng bộ **Instance Synced**, bạn có thể sao chép biến thành mọi đối tượng của class **Blackboard** trong màn chơi. Bây giờ, hãy bắt đầu tạo **Blackboard** của chúng ta:

1. Chúng ta cần tạo một **Blackboard** mới (nằm dưới tài nguyên **Artificial Intelligence** khi bạn nhấp chuột phải vào **Content Browser**) và đặt tên cho nó là *DogBrain*. Nó sẽ lưu trạng thái **State** của chúng ta dưới dạng *Enum* của *DogState* cho các nhánh của **Behavior Tree**, *Mailman* kiểu **Object** sẽ nhớ **ThirdPersonCharacter**, *Foodbowl* kiểu **Object** sẽ nhớ chén thức ăn và cuối cùng là *Locations* tìm kiếm mới nhất của chúng ta, nó có kiểu vector cho EQS. Bạn có thể thấy các phần tử chúng ta đã tạo như sau:

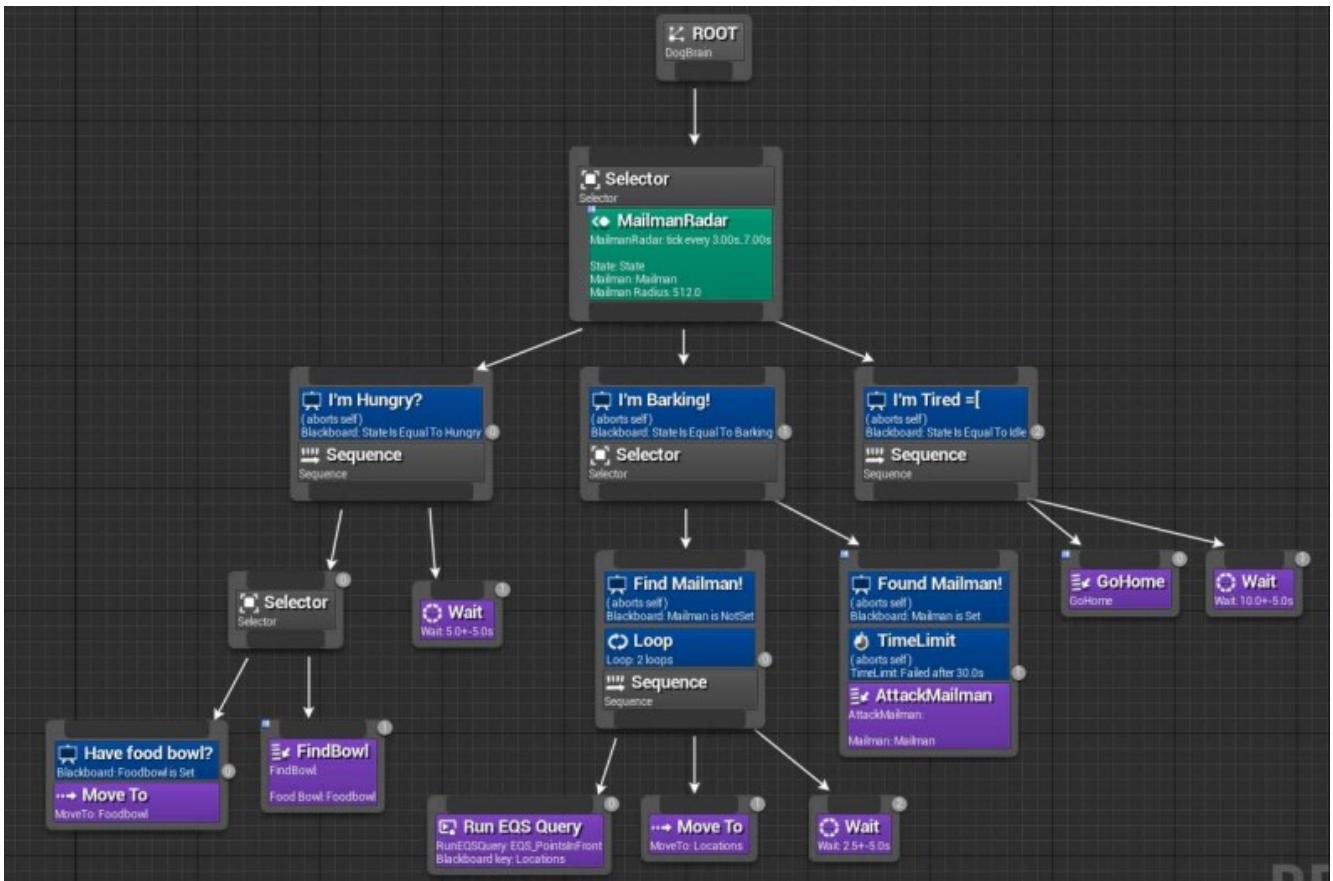


2. Hãy lưu lại tất cả. Bây giờ chúng ta phải tạo một **Behavior Tree** vì chúng ta có các component cơ bản tạo nên **Behavior Tree** của mình. Hãy xem qua những component này một cách nhanh chóng:
- *DogState*: Nó sẽ lưu trữ trạng thái hiện tại của chúng ta vì đây là cây hướng trạng thái
 - *DogHouse*: Điều này sẽ đại diện cho nơi đặt chuồng chó của chúng ta cho chú chó ngủ
 - *Foodbowl*: Tương tự như *DogHouse*, đây sẽ là vị trí đặt bát thức ăn khi chó đói

Thiết kế Behavior Tree

Hãy đặt tên cho Behavior Tree của chúng ta là *DogTree* và đảm bảo rằng tài nguyên **BlackBoard** *DogBrain* của chúng ta được cắm vào node khởi đầu **ROOT**. Tài nguyên **Blackboard** được thiết lập ở đây có các biến có thể được truy cập bởi các hàm trong cây trong khi thực thi. Khi các biến được kích hoạt và đồng bộ hóa, bạn có một biến toàn cầu cho *Mailman* mà tất cả những chú chó khác có cùng nội dung Blackboard đều có thể nhìn thấy.

Cây hành vi của chúng ta sẽ được thiết lập theo cách tương tự như ảnh chụp màn hình trực quan sau:

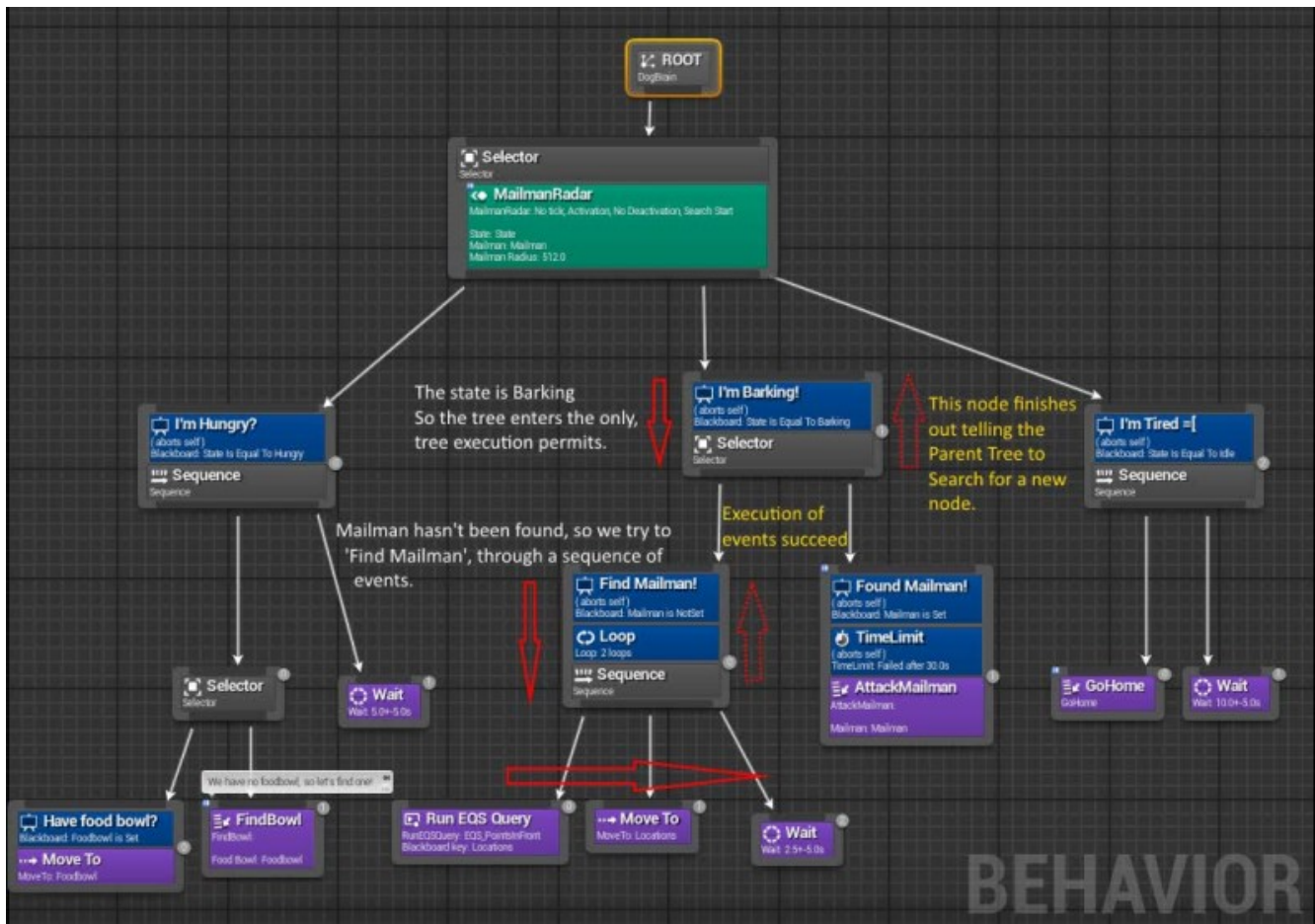


Những gì bạn có thể lưu ý ở đây là chúng ta sẽ bắt đầu với một **Selector** node. Điều này là do chúng ta không muốn thoát khỏi cây nếu node con của chúng tôi bị lỗi. Nếu thất bại, chúng ta muốn tiếp tục đến node tiếp theo cuối cùng sẽ thành công. Trong các cây riêng lẻ, chúng ta sẽ làm điều gì đó khác biệt. Bây giờ chúng ta đang ở trong một trạng thái, đôi khi chúng ta muốn kiểm soát chuỗi sự kiện để sao chép một hành vi cụ thể cho trạng thái này. Vì vậy, đối với trạng thái *Hungry* của chúng ta, trước tiên chúng ta cần có một cái bát. Khi chúng ta có bát của mình, chúng ta có thể đi ăn. Chúng ta cũng sẽ bao gồm một node chờ đợi waiting và điều này làm cho chú chó của chúng ta có vẻ như đang tìm kiếm chiếc bát dành cho chó của nó. Điều này chỉ xảy ra lần đầu tiên. Lần tới, kiểm tra điều kiện đối với bát được đặt sẽ đúng và chúng ta sẽ ngay lập tức chuyển sang *Foodbowl* của chúng ta.

Ở trạng thái *Barking* của chúng ta, chúng ta sẽ ngay lập tức đi vào node **Selector** vì chúng ta muốn thực hiện lại bất kỳ tác vụ nào ở đó và đợi cho đến khi một tác vụ thành công. Điều này có nghĩa là tác vụ trong đó điều kiện được đáp ứng trả về thành công và dẫn đến lỗi đối với node **Selector Composite**. Đối với điều này, chúng ta cần thực thi một vài chức năng và tất cả phải trả về thành công.

Điều này rất quan trọng để hiểu vì logic cơ bản này là thứ cho phép bạn xây dựng một hành vi thực thi mong muốn trong Behavior Tree. Vì vậy, trong phần phân tích sau đây, tôi sẽ phác thảo quy trình theo thứ tự thực hiện và cách xây dựng cây phụ thuộc vào hướng của

luồng thực thi:



Event tìm kiếm **Tree Search** nằm trong **Service** có tên *MailmanRadar* và chúng ta sẽ sử dụng event này để di chuyển các trạng thái chuyển tiếp.

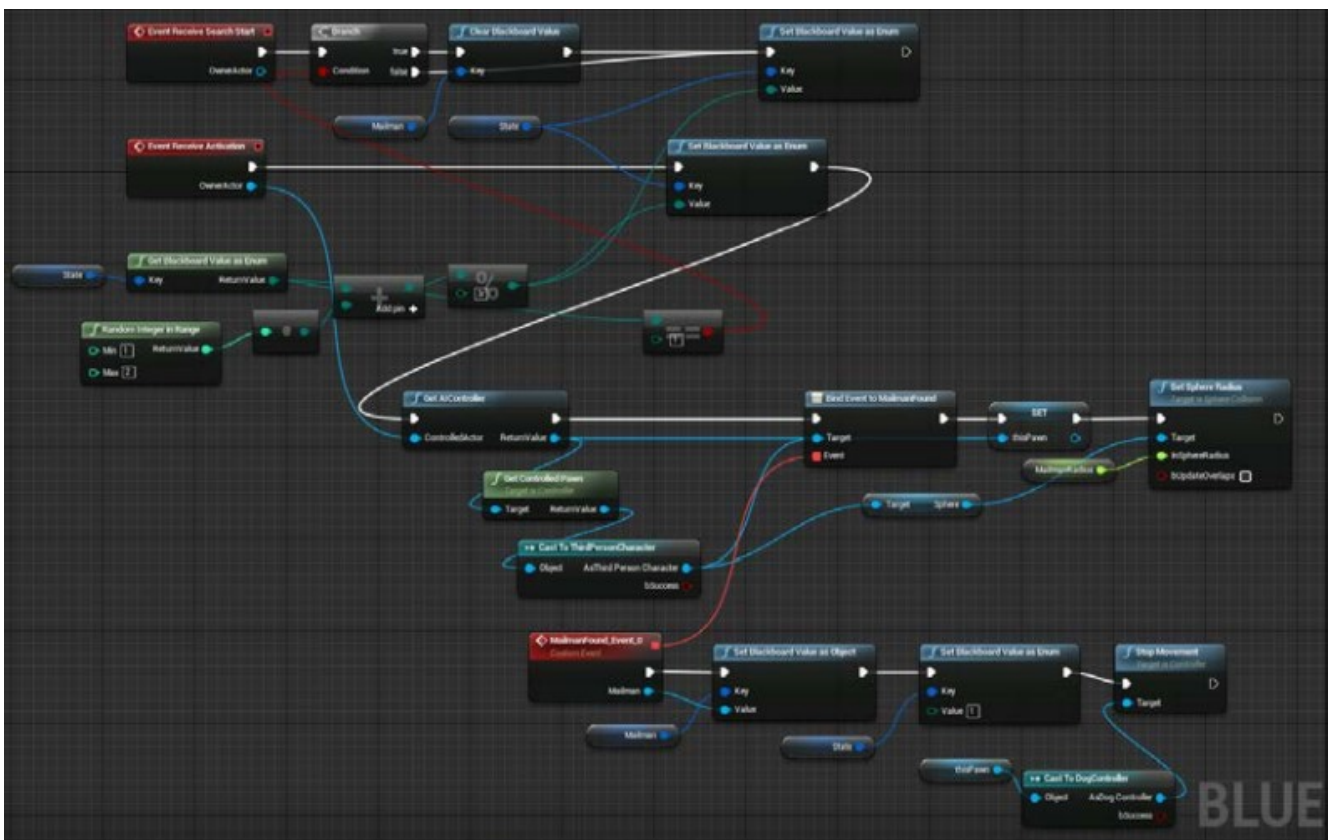
Cuối cùng, chúng ta có trạng thái *Idle*, trạng thái này chỉ đơn giản là khiến chú chó quay trở lại chuồng chó.

Vì vậy, với điều này, hãy tiếp tục chuẩn bị cho chú chó AI của chúng ta. Bây giờ chúng ta cần tạo một **AIController** cho chú chó của mình và điều này là do **AIController** chịu trách nhiệm thực thi Behavior Tree. Chúng tôi sẽ chỉ thực hiện các bước sau:

1. Hãy tạo một Blueprint mới và chọn **Custom Classes**. Sau đó, chúng tôi sẽ tìm kiếm **AIController**.
2. Bây giờ, chúng ta có thể đặt tên cho nó là **DogController** và tạo nó.
3. Chúng ta cũng cần tạo **Service**, **Service** này sẽ xác định trạng thái đang có. Để thực hiện việc này, nhấp chuột phải và tạo một blueprint mới. Trong **Custom Classes**, chúng ta cần tìm kiếm **BTService_BlueprintBase**. Hãy đặt tên cho nó là **MailmanRadar**.

Service của Behavior Tree

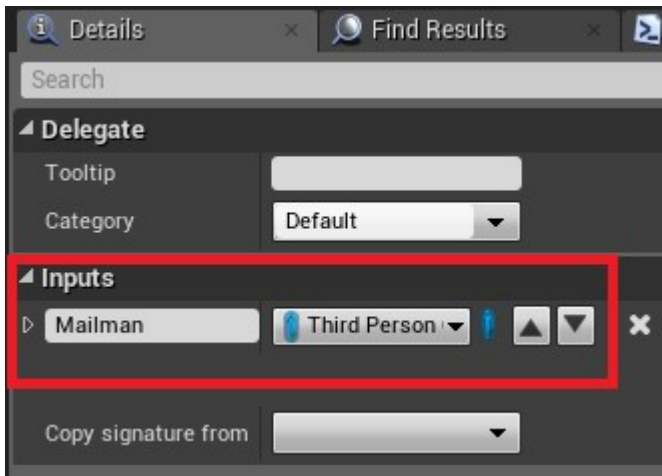
Điều làm cho node này trở nên độc đáo là nó được thiết kế để chạy và giám sát nhánh mà nó được gắn vào. Nó sẽ thực thi ở tần suất xác định để kiểm tra và cập nhật **Blackboard** miễn là nhánh của nó được thực thi. Hôm nay, chúng ta sẽ sử dụng các chức năng này. Node Kích hoạt **Event Receive Activation** đầu tiên thông báo cho chúng ta rằng node **Branch** đã được kích hoạt và nó là điều hoàn hảo để khởi tạo các biến trong **Blackboard** liên quan đến nhánh này. Node tiếp theo mà chúng ta sẽ sử dụng là **Event Receive Search Start** và node này sẽ thông báo cho chúng ta khi một nhánh được chọn. Chúng ta sẽ tận dụng điều này và thay đổi trạng thái của chúng ta, nó báo cho cây của chúng ta chọn một luồng dẫn thực hiện mới. Ảnh chụp màn hình sau đây hiển thị blueprint tổng thể cho service *MailmanRadar*:



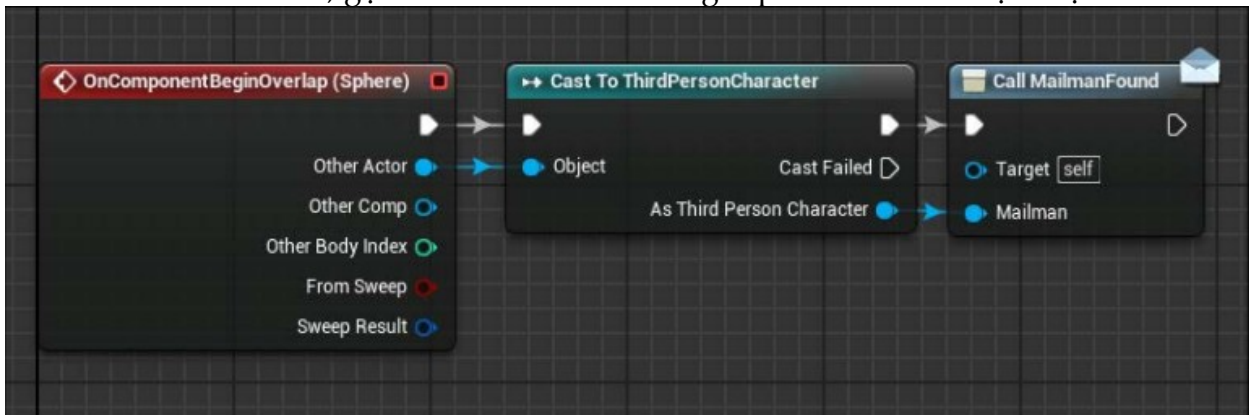
Bây giờ, để radar của chúng ta hoạt động, chúng ta phải thiết lập vài thứ cần thiết lên con tốt. Nó sẽ chịu trách nhiệm chuyển tiếp thông tin đến Behavior Tree bất cứ khi nào có một con tốt nào khác chồng lên nó. Vì vậy, hãy tìm *ThirdPersonCharacter* của chúng ta và mở nó ra.

1. Chúng ta cần vào Viewport và tạo một component **Sphere Collision** mới. Bán kính hình cầu của component này sẽ được cập nhật bởi chức năng radar của chúng ta trong cây. Hiện tại chúng ta sẽ thiết lập **Sphere Radius** dưới phần **Shape** giá trị là **512** đơn vị.
2. Bây giờ, hãy thêm event **OnComponentBeginOverlap** vào sơ đồ **EventGraph**.
3. Hãy tạo **Event Dispatcher** bằng cách thêm vào panel **My Blueprint**. Kéo nó vào sơ đồ

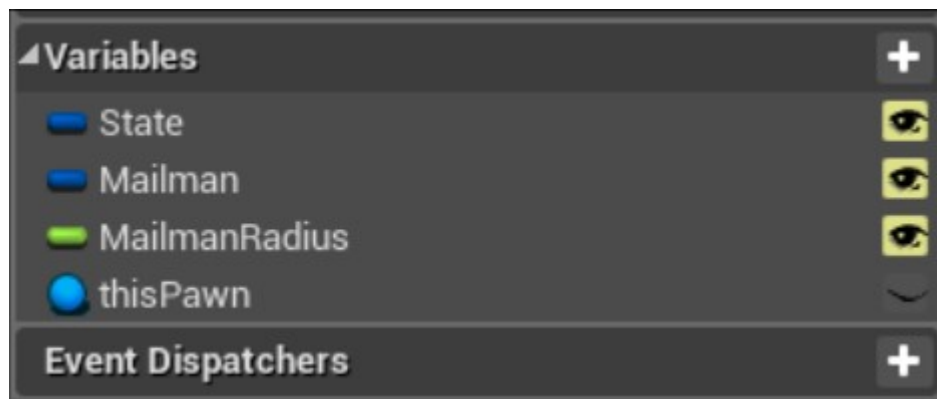
EventGraph và chọn **Call. Event Dispatcher** này sẽ chịu trách nhiệm thông báo cho cây về những thay đổi. Hãy đặt tên cho event này là *MailmanFound* và cung cấp cho nó một loại đầu vào **ThirdPersonCharacter** có tên là *Mailman*. panel **Details** của *MailManFound* sẽ như sau.



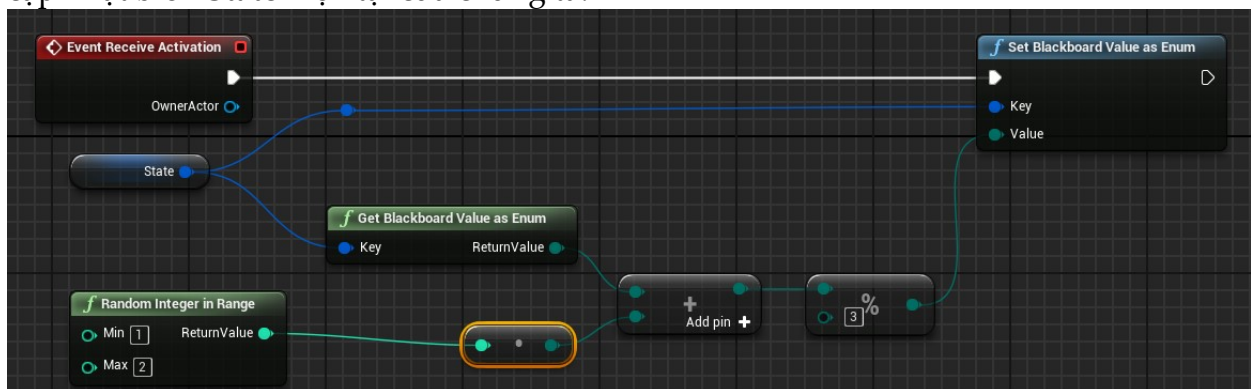
4. Từ event **Overlap** của chúng ta, chúng ta sẽ cast **Other Actor** sang **ThirdPersonCharacter**, gọi *MailmanFound* và cung cấp actor mà sẽ được chọn:



5. Bây giờ, hãy chuyển qua *MailmanRadar* và thiết lập một số biến khác để đáp ứng event này. Chúng ta cần thêm một vài biến để giữ tham chiếu trong khi radar này quét các mục tiêu có thể là người đưa thư:
- Đầu tiên, chúng ta cần một biến kiểu **Blackboard Key Selector** có tên là *State*, có thể chỉnh sửa được editable.
 - Tiếp theo, chúng ta cần một biến kiểu **Blackboard Key Selector** tên là *Mailman*, biến này cũng chỉnh sửa được editable.
 - Tiếp theo, chúng ta cần một biến **Float** có tên là *MailmanRadius*, biến này có thể chỉnh sửa được editable và có giá trị mặc định là 512.
 - Cuối cùng, chúng ta cần tạo một biến kiểu **Actor** và gọi nó là *thisPawn*



6. Bắt đầu với event khởi tạo của chúng ta, **Event Receive Activation**.
7. Chúng ta sẽ sử dụng nó để khởi tạo các biến **Blackboard** của chúng ta. Vì vậy, hãy lấy biến **State** của chúng ta sao cho chúng ta có thể thiết lập trạng thái ban đầu.
8. Kéo từ biến **Get State**, chúng ta sẽ tìm **Set Blackboard Value as Enum**. Nhận trạng thái từ các biến và lần này, chúng ta muốn **Get Blackboard Value as Enum**. Chúng ta sẽ chọn một trạng thái ngẫu nhiên mới cho AI của chúng ta. Chúng ta sẽ làm điều này bằng cách tăng biến **State** của chúng ta một cách ngẫu nhiên một hoặc hai lần. Vì vậy, hãy kéo ra từ đây, và tạo node **Byte + Byte**, sau đó tạo **modulo (%)**. **Modulo** này sẽ lấy kết quả của chúng ta từ node cộng (+), nhưng chúng ta muốn 3 được thiết lập ở vị trí cuối cùng. Vì vậy, chúng ta sẽ có một cái gì đó tương tự như $Input \% 3 = Range (0-2)$ (có nghĩa là Hungry, Barking và Idle).
9. Bây giờ, hãy tạo **Random Integer in Range** và đặt giá trị **Min** thành 1 và giá trị **Max** thành 2. Sau đó, chúng ta muốn bơm giá trị này vào node bổ sung **Byte + Byte**. Giá trị **False** từ node **Branch** trước đó của chúng ta sẽ cắm vào node **Set Blackboard Value as Enum**.
10. Cuối cùng, cắm kết quả của **modulo (%)** vào node **Set Blackboard Value as Enum** để cập nhật biến **State** hiện tại của chúng ta:



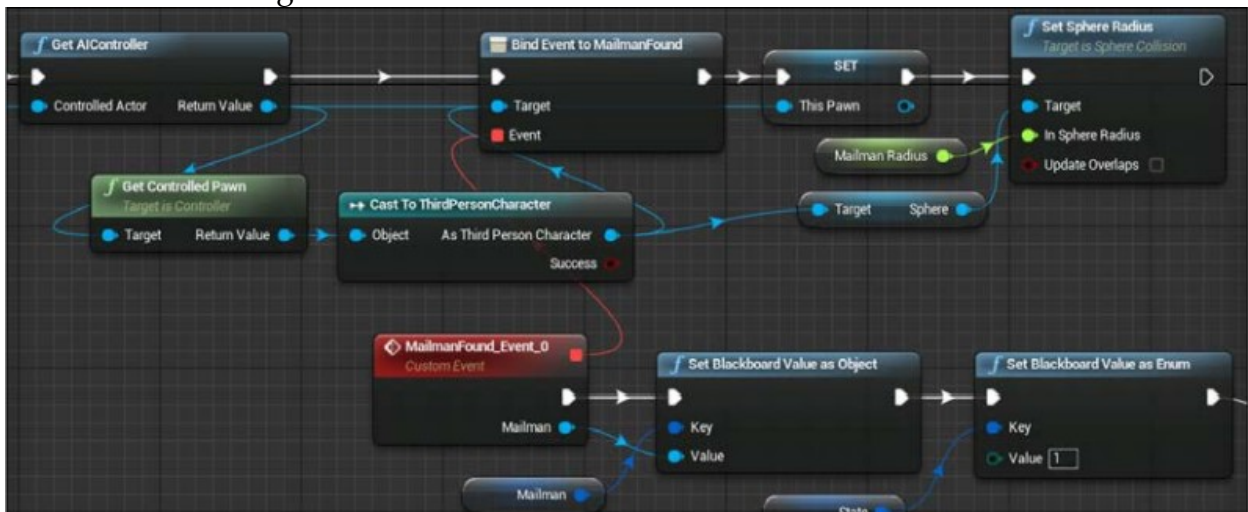
Chuyển đổi trạng thái

Những gì chúng ta sẽ làm ở đây là tăng ngẫu nhiên trạng thái của chúng ta về phía trước. Điều này chỉ đơn giản là thể hiện sự chuyển đổi trạng thái. Thao tác này rất quan trọng vì

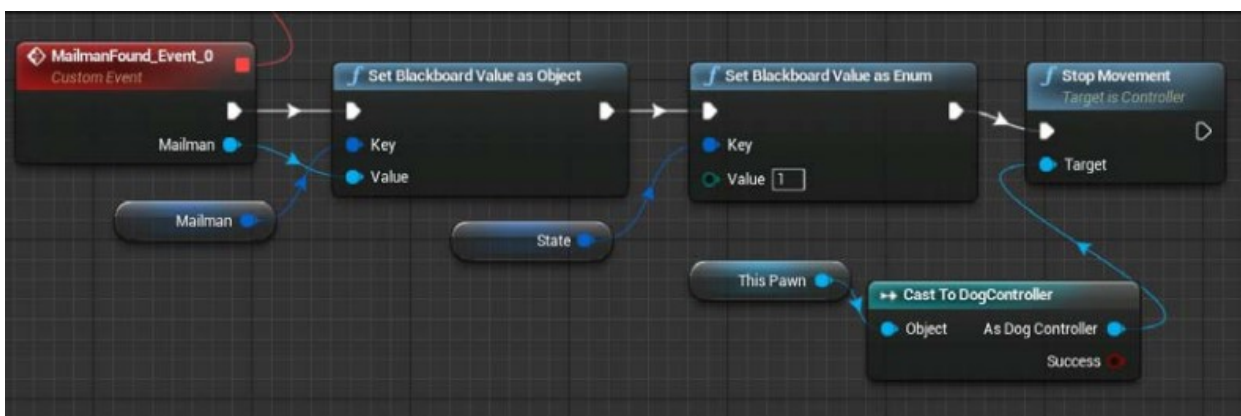
cách bạn xử lý các chuyển đổi có thể tạo ra AI hầu như mạch lạc. Điều này được thiết lập khi bạn có AI định hướng mục tiêu. Thay vì hoàn toàn phản ứng, nó chủ động chọn trạng thái ghi bàn tốt nhất để đạt được mục tiêu đã nói. Điều này làm cho có vẻ như AI có một mức độ thông minh.

AI của chúng ta chọn ngẫu nhiên trạng thái tiếp theo, nhưng nếu AI có thể đưa ra quyết định về trạng thái mới dựa trên thông tin chúng tôi thu thập được, thì AI của chúng ta có vẻ thông minh. Những ví dụ này sẽ cung cấp cho bạn kiến thức để đảm nhận các dự án khó khăn hơn:

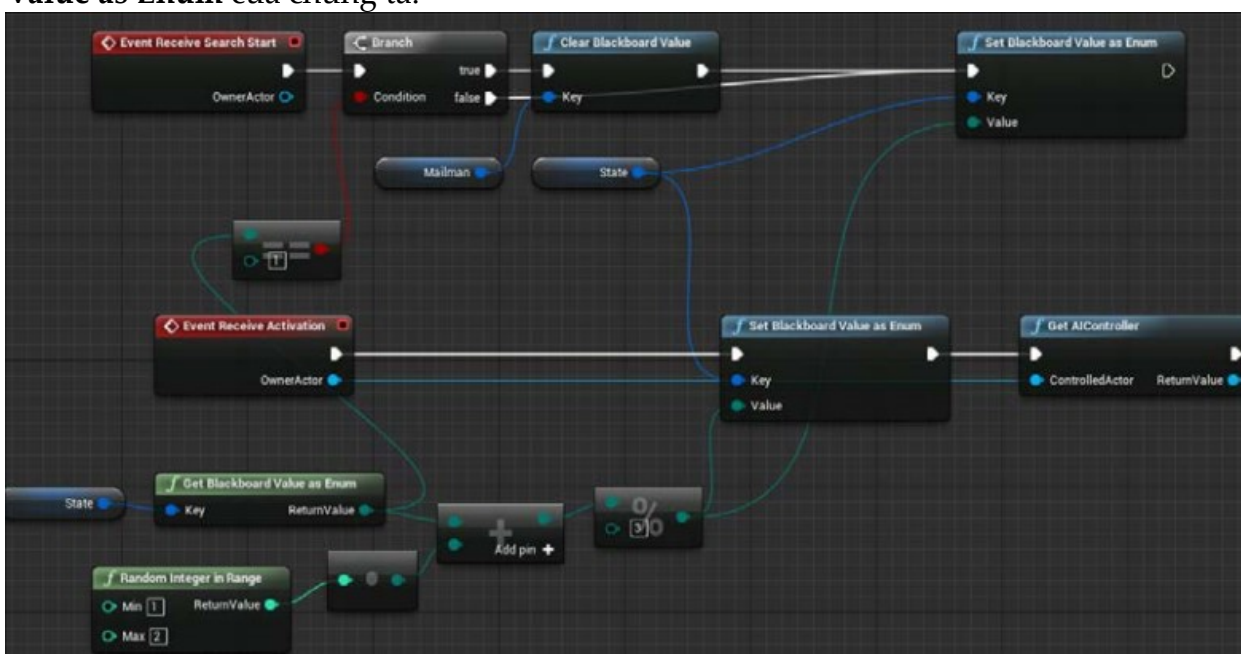
1. Nhìn lại **Event Receive Activation**, hãy kéo ra node **Get AIController** từ chân **Owner Actor**. Sau đó, chúng ta sẽ kéo **Get Controlled Pawn** để chỉ định node **Event Dispatcher** mà chúng ta đã tạo trước đó.
2. Cast từ node **Get Controlled Pawn** cho **ThirdPersonCharacter** và sau khi chiếu, hãy gán assign *MailmanFound*.
3. Chúng ta cũng sẽ lấy **Owner Actor** và đặt nó trong biến **thisPawn** của chúng ta. Cuối cùng, chúng ta nên cập nhật biến **Sphere** của con tốt của mình để khớp với **Radius** của node biến mà chúng ta muốn kiểm tra:



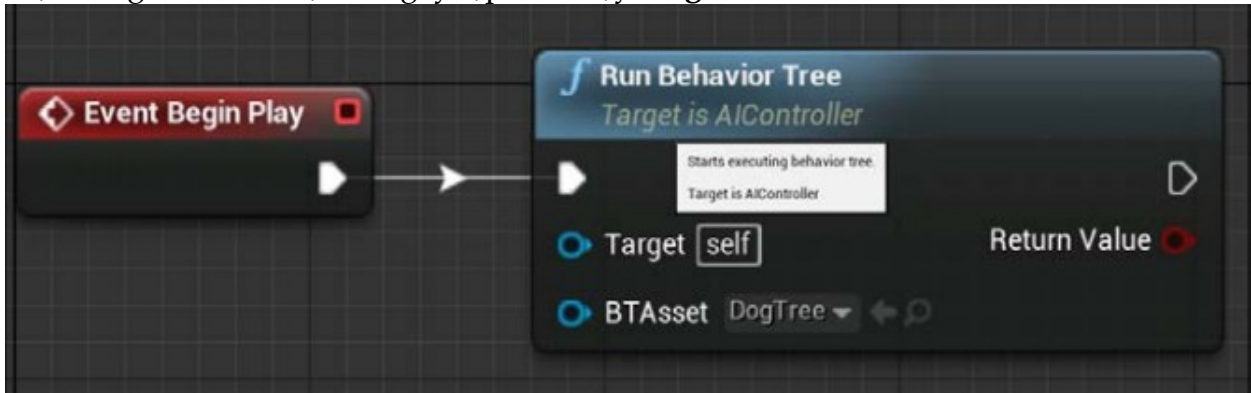
4. Từ event **MailmanFound** mới được tạo của chúng ta, chúng ta sẽ tiến hành cài đặt cho biến **selector Blackboard Mailman** của mình. Vì vậy, hãy lấy nó từ danh sách biến vào sơ đồ, thực thi **Set Blackboard Value as Object** và cắm *Mailman* được trả về thông qua sự kiện vào chân **Value**.
5. Tiếp theo, đặt biến **State** của chúng ta với *Barking* để chúng ta ngay lập tức hủy bỏ các trạng thái khác và tiếp tục với *Barking*. Lấy biến **State** từ danh sách biến vào sơ đồ và thực thi node **Set Blackboard Value as Enum** từ node **Set Blackboard Value as Object**. Chúng tôi muốn đặt giá trị này là **1**, đại diện cho trạng thái *Barking* của chúng ta.
6. Cuối cùng, hãy lấy biến *This Pawn* từ danh sách biến vào sơ đồ của chúng ta. Cast node này thành **DogController** và sau đó gọi **Stop Movement**. Thao tác này sẽ dừng bất kỳ chuyển động hiện tại nào và bắt đầu chuyển động được yêu cầu tiếp theo:



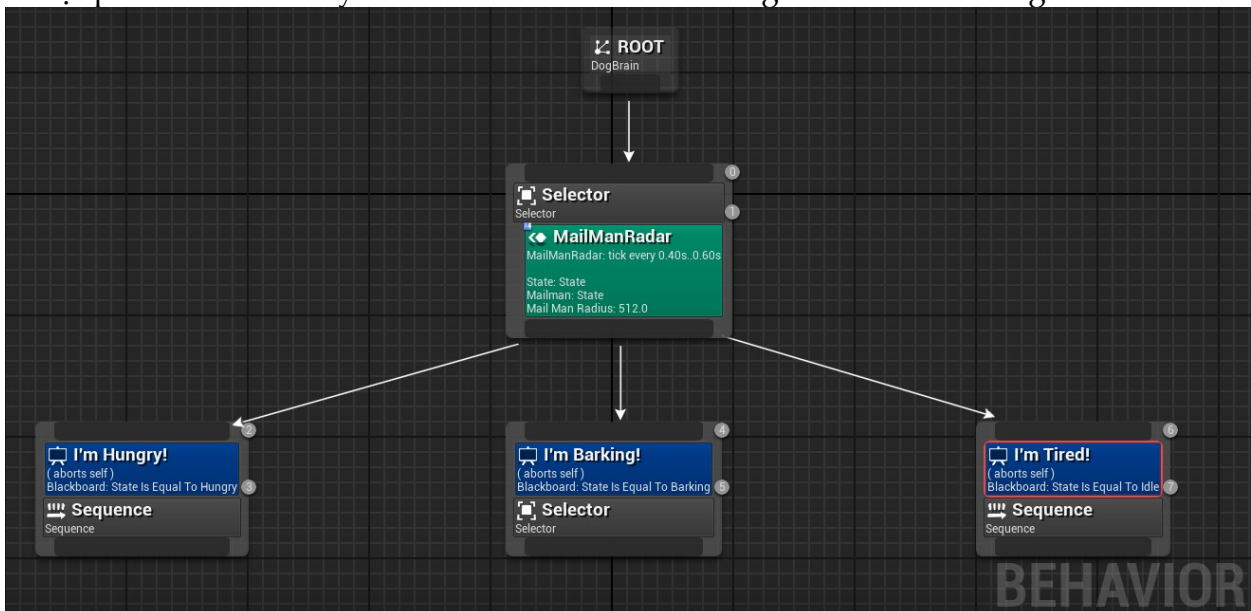
7. Bây giờ, tới **Event Receive Search Start**, cây (Behavior Tree) sẽ tìm kiếm một nhánh mới để bắt đầu. Vì vậy, chúng ta sẽ sử dụng cái này để thiết lập các trạng thái mới. Bây giờ, hãy nhấp chuột phải vào sơ đồ của chúng ta và kéo event ra.
8. Từ đây, chúng ta cần tạo một điều kiện bằng **1** với **Return Value** của node **Get Blackboard Value as Enum** và sau đó kết nối nó với một node **Branch**. Những gì chúng ta đã làm là so sánh trạng thái với 1, nó có nghĩa là chúng ta đã nói rằng biến **State** của chúng ta bằng với **Barking**, cho phép chúng ta thực hiện các thao tác cụ thể để rời khỏi trạng thái này. Nhánh này sẽ là node được thực hiện tiếp theo.
9. Bây giờ, chúng ta cần lấy biến **Mailman** từ danh sách biến vào sơ đồ và gọi một hàm để reset các giá trị của Blackboard Key. Chúng ta sẽ thực hiện việc này bằng cách kéo từ chân **Mailman** và tìm kiếm node **Clear Blackboard Value**. Nó sẽ được gọi nếu node **Branch** mà chúng ta đã tạo trả về giá trị **True**.
10. Chúng ta cũng muốn đặt một trạng thái mới nếu chúng ta nhận được biến **State** của mình và kéo **Set Blackboard Value as Enum**. Chúng ta có thể cập nhật biến **State** hiện tại của mình bằng phép tính mà chúng ta đã thực hiện trên **Event Receive Activation**. Vì vậy, hãy lấy từ **modulo (%)** này và đặt nó vào chân **Value** của node **Set Blackboard Value as Enum** của chúng ta:



11. Hãy mở class *DogController* này và tìm node **Event Begin Play**.
12. Nhấp chuột phải vào sơ đồ và tìm node **Run Behavior Tree**.
13. Sau đó, trong chân **BTAsset**, hãy tìm **DogTree** mà chúng ta đã tạo trước đó. Sau đó, trong trò chơi, chúng ta sẽ gán class **DogController** cho nhân vật của mình. Nó sẽ xuất hiện cùng với nhân vật và ngay lập tức chạy **DogTree**:



14. Quay lại **Behavior Tree** của chúng ta, hãy kéo một mũi tên xuống và tạo một **Selector** tổng hợp mới trong sơ đồ của chúng ta. Đây là điểm bắt đầu trong cây, nó được hiển thị trong ảnh chụp màn hình trước đó. Cài đặt Service vào node bằng cách click chuột phải lên node và chọn **Service** > *MailManRadar*. Nó sẽ chạy node *MailmanRadar* **Service** đã được tạo của chúng ta. Điểm độc đáo của các node **Service** là chúng chỉ được sử dụng trên điều kiện hỗn hợp và chịu trách nhiệm xử lý các tác vụ yêu cầu chạy với cây.
15. Bây giờ, hãy xây dựng ba tùy chọn node tiếp theo vào trong cây. Đây sẽ là ba node Sequence có **Decorators** là **Blackboard Based Condition** trên chúng. Decorator có thể áp dụng các điều kiện cho các node Composites (Hỗn hợp) hoặc Task (Tác vụ) của chúng ta. Các decorator này sẽ kiểm tra trạng thái truy cập thích hợp để bắt đầu thực thi.
16. Vì vậy, hãy tạo các node Composite của chúng ta từ đầu cây này. Sau đó, hãy nhấp chuột phải vào node này và thêm **Decorator** mà chúng ta muốn tìm trong **Blackboard**:



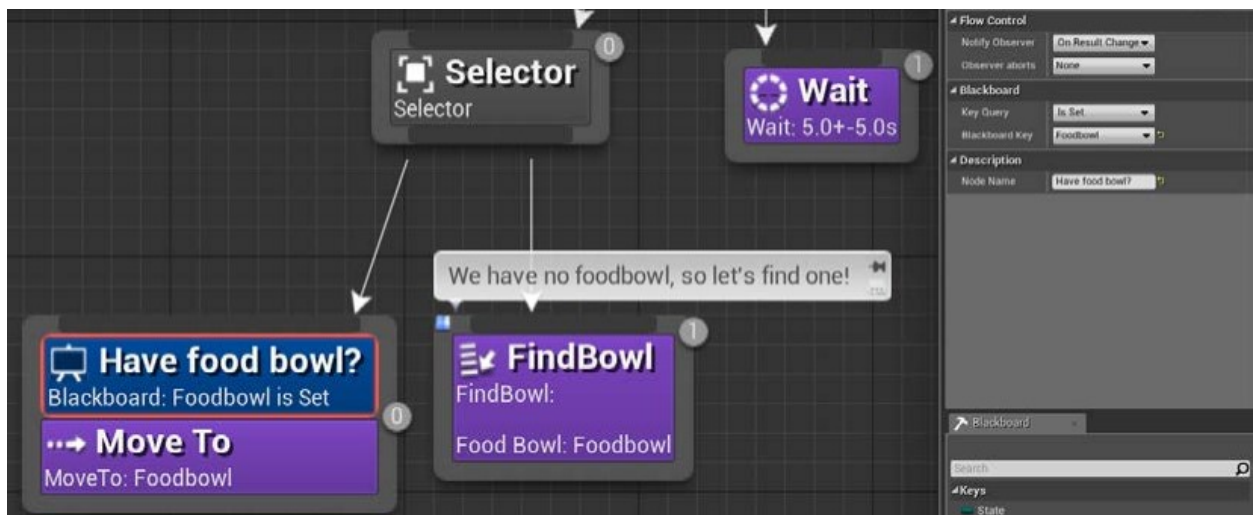
Blackboard Compare Decorator

Một vài điều cần chú ý:

- Hãy bắt đầu với **Flow Control**. Nó có hai thuộc tính: **Notify Observer** và **Observer aborts**. **Notify Observer** nói cho **Flow Control** biết hoạt động nào sẽ được kích hoạt. **On Result Change** kích hoạt khi điều kiện được tính toán để nhập vào cây này thay đổi. Vì vậy, nếu ban đầu chúng ta barking nhưng hiện không hoạt động, thì **Flow Control** của cây sẽ kích hoạt sự kiện **Abort**. Phần nào của cây bị hủy bỏ được kiểm soát bởi **Observer abort**. Tùy chọn đầu tiên là **Selft**. Điều này đơn giản ở chỗ nó hủy bỏ việc thực hiện thêm trong cây này. Tùy chọn thứ hai là **Low Priority**. Tùy chọn này về cơ bản cho biết phần còn lại của cây sẽ bị hủy bỏ. Điều này là tốt nếu bạn muốn toàn bộ cây bắt đầu lại. Tùy chọn cuối cùng chỉ đơn giản là sự kết hợp của hai tùy chọn đầu tiên mà tôi đã đề cập.
- Tiếp theo là **Blackboard** và trường này chịu trách nhiệm thông báo cho Decorator biết điều kiện nào trả về **True**. Trong trường hợp này, chúng ta muốn đặt **Blackboard Key** vào **State**. Lưu ý các thuộc tính thay đổi. Thay đổi giá trị **Key Query** thành điều kiện mà chúng ta áp dụng cho giá trị **Key**. Sau đó, cuối cùng, đặt giá trị **Key** thành giá trị mà chúng tôi đang kiểm tra điều kiện của mình.

Hãy quay trở lại **Behavior Tree** của chúng ta:

1. Node đầu tiên trong nhánh Sequence là node **Selector**. Chúng ta muốn hành vi này bởi vì chúng ta có hai điều kiện mà chúng ta muốn được kiểm tra, nhưng chúng ta biết rằng chỉ có thể đáp ứng một điều kiện tại một thời điểm. Node tiếp theo chúng ta cần tạo trong nhánh Sequence này là node **Wait**. Điều này sẽ mô phỏng hoạt động tìm kiếm hoặc hoạt ảnh của chú chó trong khi nó đánh hơi hoặc ăn thức ăn của nó.
2. Node đầu tiên trong node Selector của chúng ta là node sẽ chịu trách nhiệm di chuyển đến bát của chúng ta, giả sử chúng ta có nó, thì sẽ sử dụng **Move To**. Node **Move To** nhận một đối số **Blackboard Key** và đối số này tương thích với Actors hoặc Vectors. Nó cung cấp một cách hạn chế nhưng trực tiếp để thực hiện chuyển động từ cây:



3. Node tiếp theo trong node **Selector** trên là node mà chúng ta cần phải tạo. Vì vậy, hãy quay lại **Content Browser** của chúng ta và tạo một Blueprint mới. Trong **Custom Classes**, chúng ta muốn tìm kiếm **BT** và chúng tôi sẽ tạo **BTTask Blueprint Base**. Cái này sẽ được gọi là *FindBowl*, và nó sẽ chịu trách nhiệm tìm bát thức ăn cho chú chó của chúng ta.
4. Sau đó, chúng ta sẽ lưu lại tài nguyên này mới này, và quay lại cây chú chó của chúng ta bên dưới nút **Selector** và tạo tác vụ *FindBowl* mới sau node **Move To**. Bây giờ chúng ta đã nói rằng node này sẽ thực thi nếu các điều kiện con khác, chẳng hạn như *Foodbowl*, được đặt thành **false**.
5. Bây giờ chúng ta đã có cây **State** đầu tiên của mình, chúng ta nên cập nhật nhận xét trên các nút của cây để hiểu ý nghĩa của nó khi viết tắt. Hãy xem cách tôi đã thực hiện trong ảnh chụp màn hình sau:



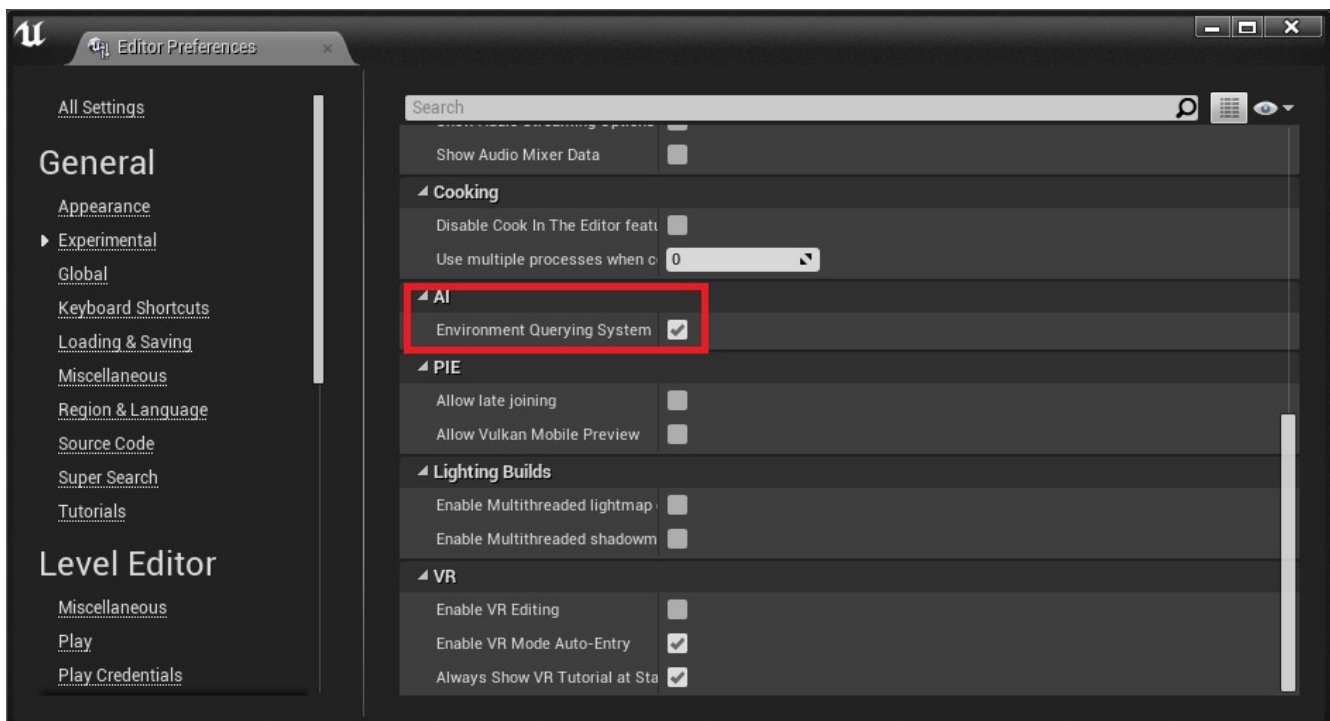
6. Bây giờ chuyển sang cây tiếp theo, chúng ta phải tạo một thiết lập tương tự, bắt đầu với một node tổng hợp **Selector**. Chúng ta sẽ nhấp chuột phải, thêm Decorator và tìm kiếm **Blackboard**. Điều này cần kiểm tra xem biến **State** có bằng *Barking* để thực hiện mục nhánh này hay không.
7. Tiếp theo, chúng ta sẽ tạo một node **Sequence Composite** sẽ chịu trách nhiệm yêu cầu AI của chúng ta dò tìm người đưa thư. Nó sẽ có **Blackboard Decorator** trên hỗn hợp để kiểm tra xem biến *Mailman* có được đặt hay không. Chúng tôi cũng muốn thêm một Decorator vòng lặp khác chịu trách nhiệm gọi điều này hai lần nếu điều kiện được đáp ứng.

8. Với thiết lập này, ba nút tiếp theo trong nhánh Sequence này trước tiên sẽ quét khu vực, di chuyển chú chó của chúng ta và sau đó đợi ở vị trí đánh hơi. Vì vậy, để quét khu vực của chúng ta, tôi muốn sử dụng EQS (Hệ thống truy vấn môi trường).

Hệ thống truy vấn môi trường (EQS)

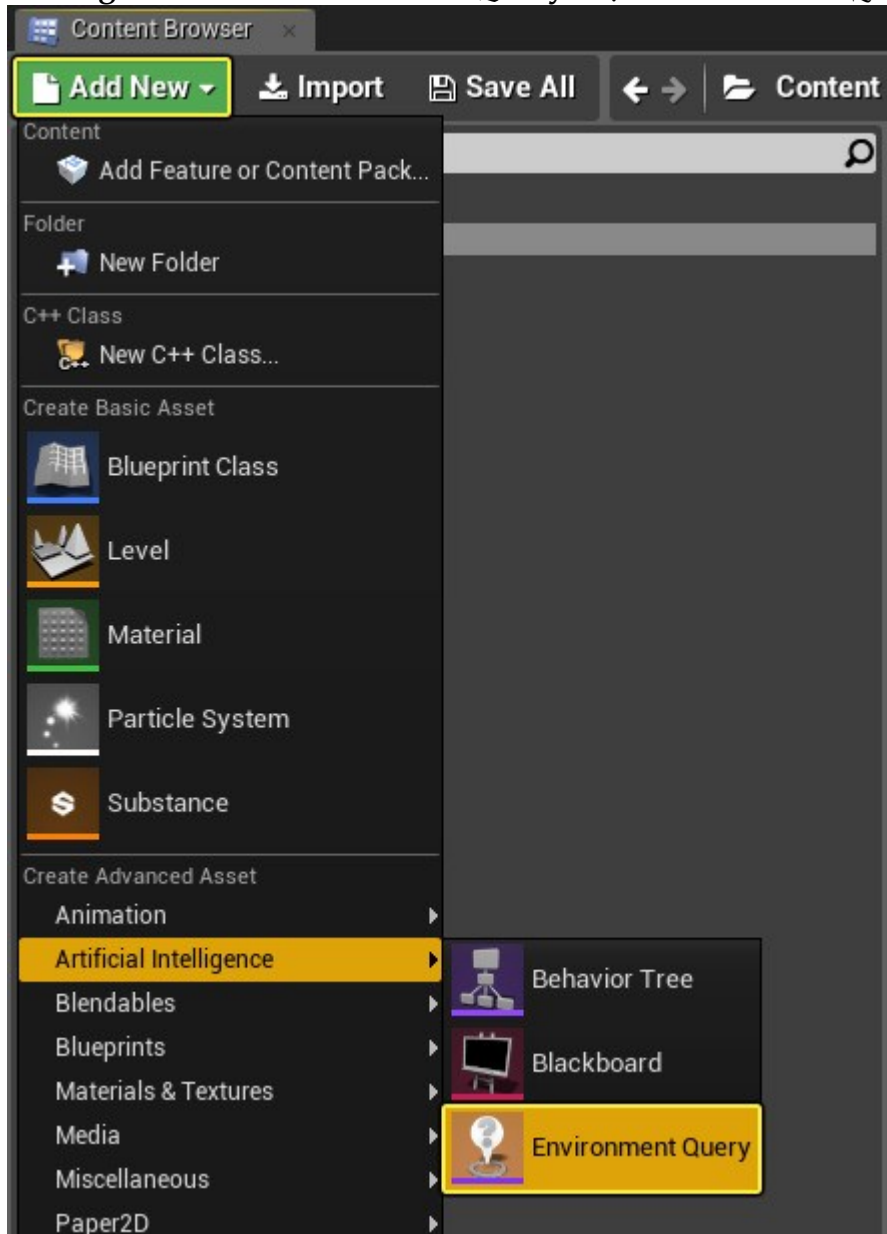
Tôi muốn đề cập đến vấn đề này nhiều hơn trong chương tiếp theo, chương này sẽ đề cập đến cách AI của chúng ta có thể cảm nhận được môi trường, nhưng ít nhất tôi sẽ giới thiệu ý tưởng và cách chúng ta sẽ sử dụng nó ngày hôm nay. Hệ thống truy vấn môi trường chịu trách nhiệm cho phép đối tượng bối cảnh gọi một yêu cầu truy vấn để tạo thông tin dựa trên các bộ lọc và thử nghiệm được áp dụng cho yêu cầu. Một truy vấn sẽ chứa một mẫu hướng dẫn để chạy EQS. Vì vậy, ví dụ: nếu bạn muốn quét một khu vực để tìm những nơi có thể ẩn nấp, bạn có thể quét những tác nhân đại diện cho những nơi ẩn nấp đó. Sau đó, bạn có thể áp dụng các bộ lọc cho truy vấn của mình để loại bỏ các kết quả mà bạn không muốn cho điểm và chấm điểm kết quả của bạn dựa trên hướng từ kẻ thù để có vị trí ẩn nấp tối ưu nhất.

Trước khi làm việc với Hệ thống truy vấn môi trường EQS, bạn phải cần kích hoạt nó từ **Editor Preferences**.

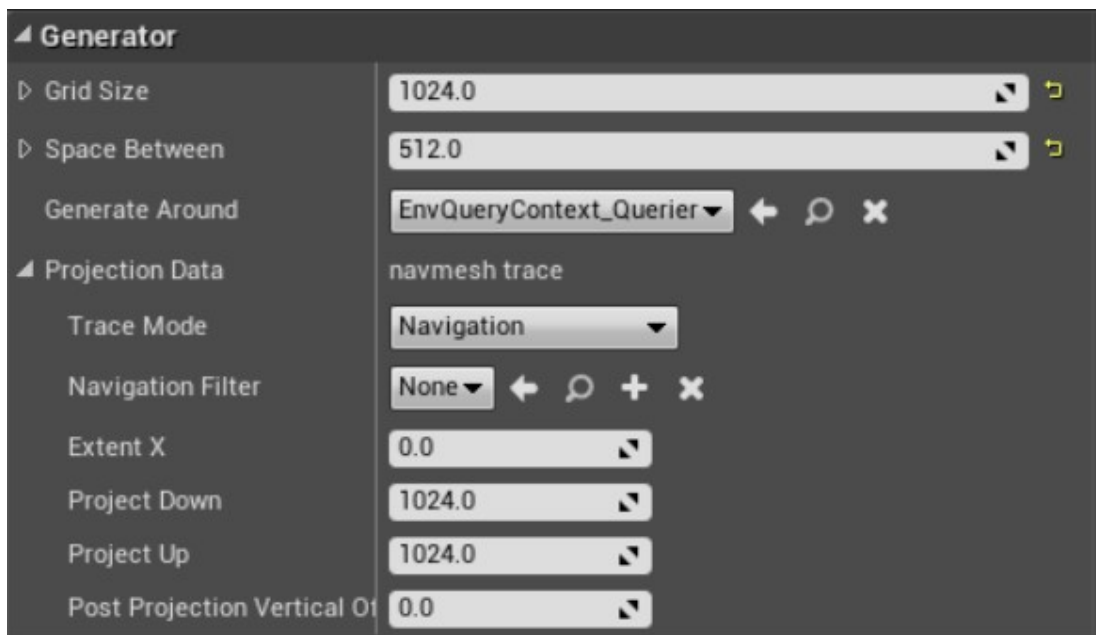


Trong trường hợp của chúng tôi, chúng tôi muốn chọn một vị trí chiến lược để đánh hơi *Mailman* của chúng tôi. Chúng tôi sẽ làm điều này bằng cách tạo Truy vấn môi trường (Environment Query). Hãy tạo cái này như sau và trong chương tiếp theo, chúng ta sẽ đề cập đến các cách khác để sử dụng EQS:

1. Vì vậy, hãy ghi nhớ điều này, hãy chuyển đến **Content Browser** và chọn thư mục *Blueprint* của chúng ta. Nhấp chuột phải vào thư mục và nhấp vào mục **Artificial Intelligent** rồi tìm **Environment Query**. Đặt tên cho nó là *EQS_PointsAround*.



2. Hãy mở cái này lên và vào root. Tại đây, bạn sẽ nhận thấy một giao diện trực quan tương tự như giao diện của **Behavior Tree**. Kéo từ thư mục root và tìm **SimpleGrid** (Point:Grid). Nó sẽ tạo ra một cái lưới phủ các mục xung quanh actor của chúng ta mà sẽ được ghi và trả lại cho người yêu cầu.
3. Chúng ta muốn thay đổi giá trị **Grid Size** thành **2048** và của **Space Between** thành **512**. Phần còn lại của cài đặt sẽ là mặc định; bây giờ hãy thêm một bài kiểm tra như trong ảnh chụp màn hình sau:



4. Bài kiểm tra này sẽ lọc và sau đó chấm điểm kết quả của chúng tôi dựa trên hướng từ chú chó của chúng ta. Để xác định bài kiểm tra **DOT** sẽ cho điểm, chúng ta phải cho bài kiểm tra biết những gì chúng ta muốn kiểm tra. Vì vậy, **Line A** sẽ là **Rotation** của phần tử nào đó của chúng ta. **Line B** sẽ là hướng giữa chúng ta và phần tử.
5. Cuối cùng, vì chú chó của chúng ta sẽ cần tìm kiếm toàn bộ khu vực, nên chúng ta cần chú chó của mình có cơ hội khảo sát toàn bộ khu vực. Bằng cách bật **Absolute Value**, chúng ta có thể nhận được kết quả để tính điểm theo cả hai hướng.
6. Chúng ta muốn điều chỉnh bộ lọc của mình để loại bỏ các mục ngay trước mặt chúng ta hoặc bên phải của chúng ta bằng cách chọn một phạm vi trong khoảng từ 0 đến 1. Hãy thay đổi giá trị **Filter Type** của chúng ta thành **Range, Float Value Min** thành **0,4** và **Float Value Max** giá trị đến **0,85**.
7. Điều cuối cùng ở đây là **Score** và chúng ta sẽ để điều này làm mặc định. Hãy lưu cái này và quay lại Behavior Tree của chúng ta:

▲ Dot

▲ Line A

ModeRotation

RotationEnvQueryContext_Querier

▲ Line B

ModeTwo Points

Line FromEnvQueryContext_Querier

Line ToEnvQueryContext_Item

Test ModeDot (3D)

Absolute Value☒

▲ Test

Test PurposeFilter and Score

▲ Filter

Filter TypeRange

▷ Float Value Min0.4

▷ Float Value Max0.85

▲ Score

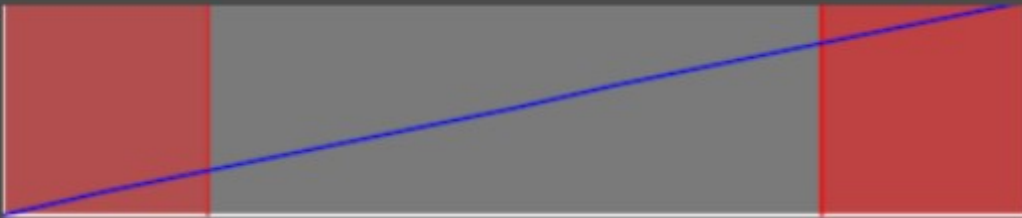
▷ Clamping

Scoring EquationLinear

Final score = ScoringFactor * NormalizedItemValue

▷ Scoring Factor1.0

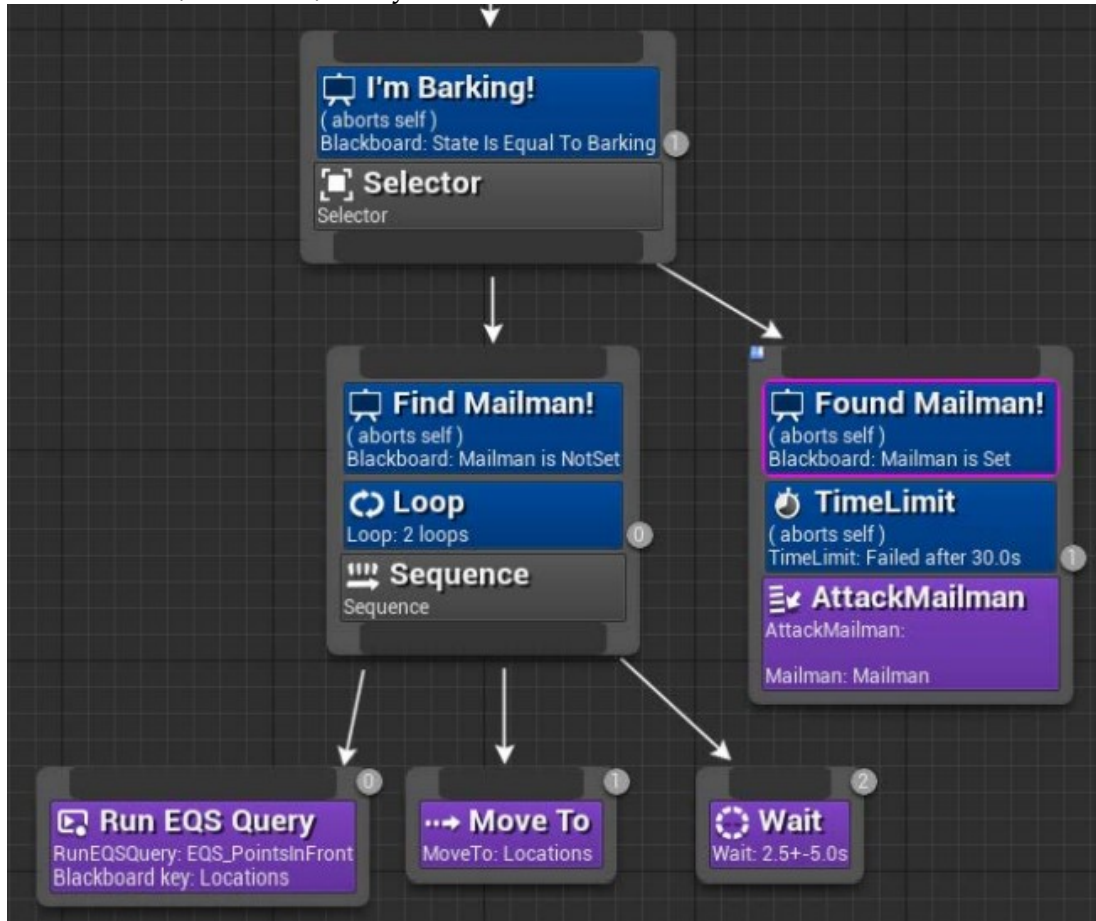
▲ Preview



8. Từ nút **Sequence** mà chúng ta đã tạo bằng **Mailman is NotSet** Decorator, bây giờ chúng ta muốn chạy truy vấn EQS của mình và lưu trữ kết quả trong khóa Blackboard Key *Location* của chúng ta. Chúng ta có thể kéo từ node và tìm **Run EQS Query**.
9. Sau đó, chúng ta cần chọn tùy chọn **EQS_PointsAround** và cài đặt Blackboard Key *Location* của chúng ta.
10. Node tiếp theo sẽ di chuyển chú chó của chúng ta đến vị trí được trả về bởi EQS. Hãy kéo và tìm **Move To**. Blackboard Key này cần phải là những vị trí *locations* của chúng

ta.

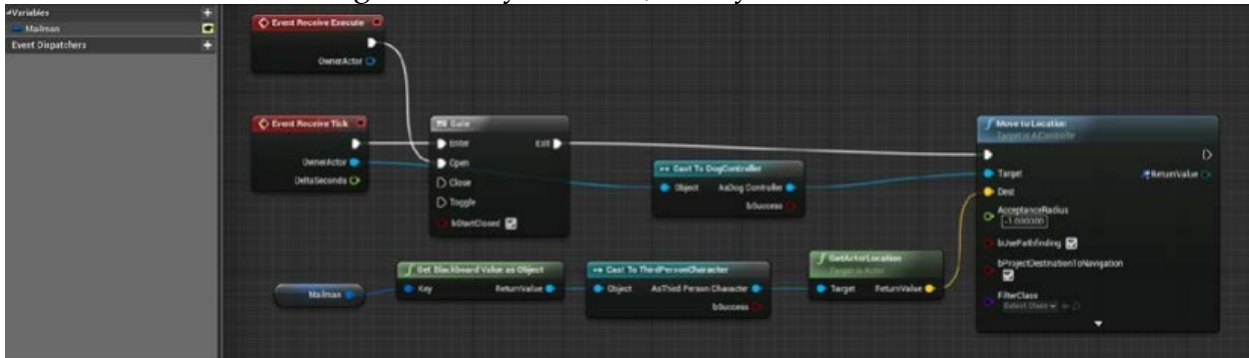
11. Node cuối cùng ở đây sẽ khiến chú chó của chúng ta đợi ở vị trí mới được tìm thấy trong một khoảng thời gian ngẫu nhiên. Vì vậy, hãy tìm node **Wait** và đặt giá trị **Wait Time** thành **2,5** và **Random Deviation** thành **5,0**. Cấu trúc hoàn chỉnh cho các nút **Selector** được hiển thị ở đây:



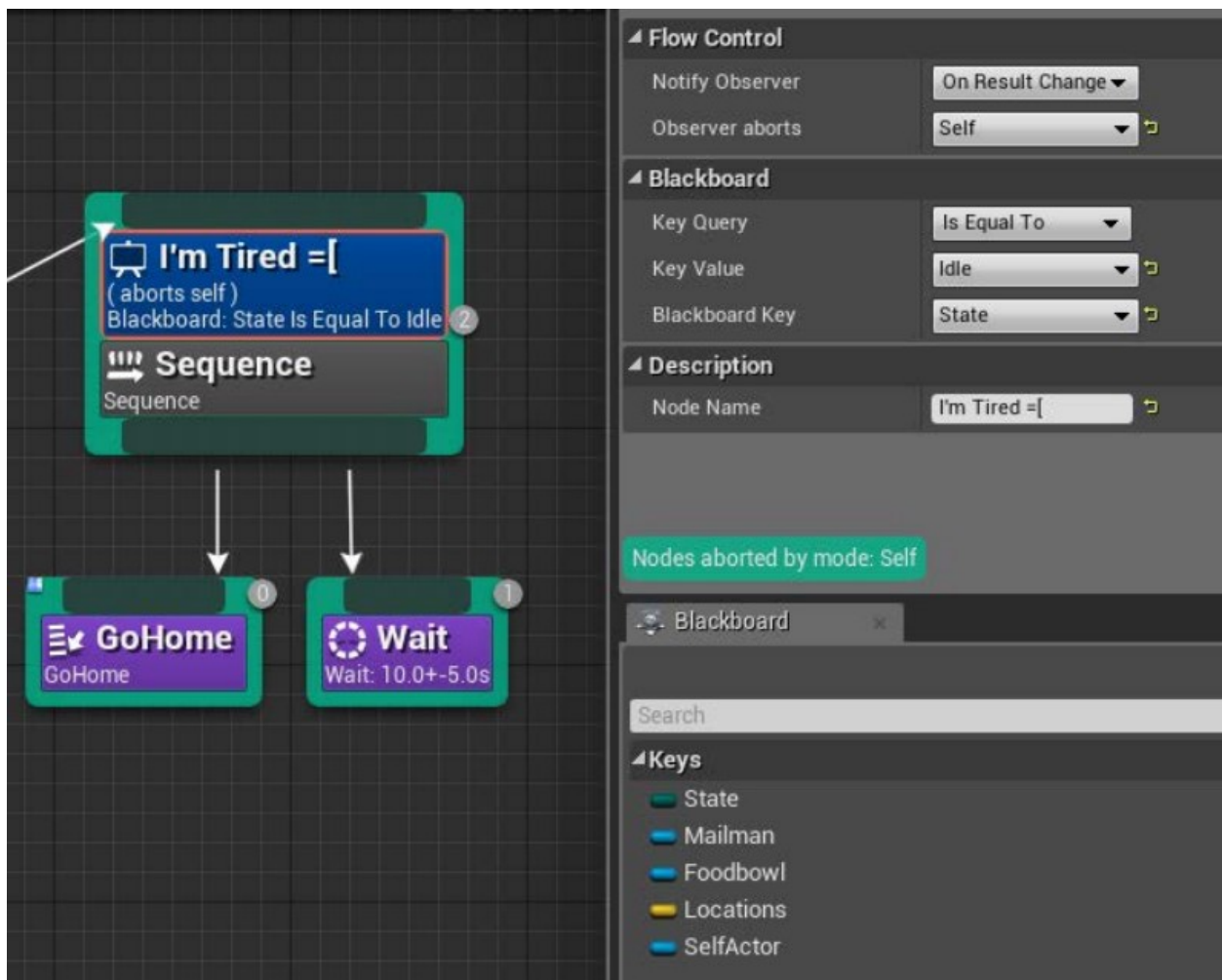
12. Quay lại nút **I'm Barking**, chúng ta sẽ tạo một nút mới sẽ khiến chú chó của chúng ta đuổi theo người đưa thư sau khi biến được đặt. Trước tiên chúng ta phải tạo nút của riêng mình, nút này sẽ không bao giờ kết thúc quá trình thực thi. Vì vậy, hãy chuyển sang **Content Browser**.
13. Chúng ta sẽ vào thư mục *Blueprint* của mình, nhấp chuột phải và tạo một bản blueprint mới. Trong **Custom Class**, chúng ta phải tìm **Tast Blueprint Base Behavior Tree**. Nó sẽ được đặt tên là *AttackMailman*.
14. Hãy mở cái này lên và đi đến sơ đồ **EventGraph**. Trước tiên, chúng ta cần lấy vị trí của mình, vì vậy trước tiên chúng ta phải tạo tham chiếu đến tài nguyên Blackboard của mình. Trong phần **Variables**, hãy tạo một phím Bảng đen Bộ chọn mới và đặt tên là *Mailman*. Chính cho biến này thành có thể chỉnh sửa được **Editable**.
15. Chúng ta phải tìm **Event Receive Execute** và nó sẽ mở **Gate** lên. Vì vậy, hãy tạo một node **Gate** và bơm event của chúng ta vào chân thực thi **Open** của node **Gate**.
16. Sau đó, chúng ta cần tạo **Event Receive Tick** và bơm nó vào chân thực thi **Enter** của node **Gate**. Sau đó, chúng ta cần kéo **OwnerActor** từ node **Event Receive Tick** và

truyền nó bằng cách sử dụng **Cast to DogController**. Từ đây, chúng ta có thể yêu cầu **DogController** của mình theo lệnh **MoveToLocation**.

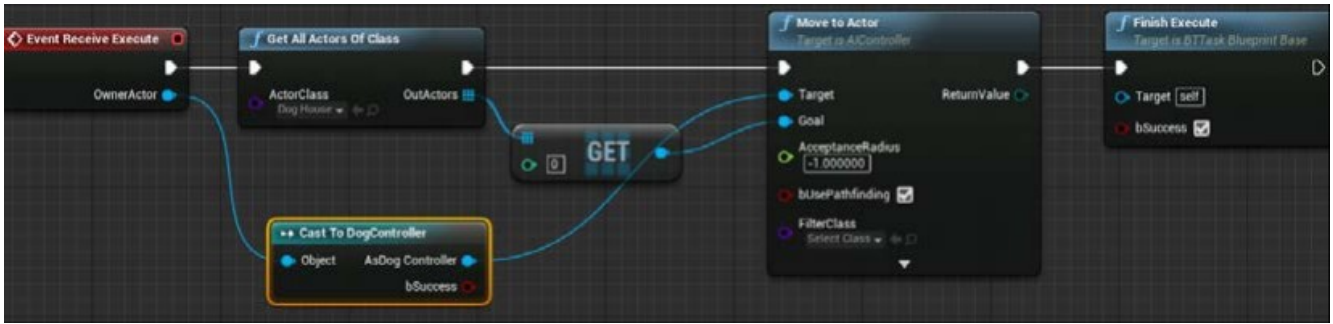
17. Bây giờ, chúng tôi phải cung cấp một địa điểm. Kéo biến *Mailman* của chúng ta, tiếp theo chúng ta phải kéo tiếp **Get Blackboard Value as Object**. Sau đó, nó sẽ được chuyển thành *ThirdPersonCharacter*. Từ đây, chúng ta có thể lấy vị trí của actor *Mailman* và bảo chú chó của chúng ta di chuyển đến vị trí này:



18. Trình tự Sequence **Idle** cuối cùng sẽ chỉ biểu thị khi con chó mệt mỏi và muốn nghỉ ngơi. Bắt đầu từ nhánh chính, chúng ta sẽ tạo một cây trạng thái mới bằng node **Sequence Composite**. Chúng ta phải nhấp chuột phải vào node **Sequence** của mình và thêm **Blackboard Decorator**. Nó sẽ kiểm tra xem giá trị **State** hiện tại có bằng giá trị *Idle* hay không.
19. Lấy từ nhánh chính của chúng ta, hãy tạo một node **Sequence**. Node này sẽ có Blackboard Decorator, sẽ có State được đặt làm giá trị Blackboard Key và **Key Query** sẽ được đặt thành **Is Equal To**. Giá trị Blackboard **Key Value** của chúng ta cho Decorator phải ở chế độ *Idle*.
20. Bên dưới **Flow Control**, chúng ta nên đảm bảo rằng giá trị **Notify Observer** là **On Result Change** và giá trị **Observer aborts** là **Self**.
21. Cập nhật trường **Description** của chúng ta thành trạng thái **Idle/Sleep** để làm cho cây của chúng ta dễ hiểu hơn:



22. Chúng tôi cần một cái cây mới để đưa chú chó của chúng ta về nhà và đợi nó ở đó.
23. Hãy đi đến **Content Browser** của chúng ta và điều hướng đến thư mục *Blueprint* của chúng ta. Nhấp chuột phải và tạo một class blueprint mới. Đây sẽ là một Custom Class **BTTask_BlueprintBase** khác. Đặt tên cho nó là *GoHome* này và mở nó lên sơ đồ EventGraph.
24. Event mà chúng tôi muốn là **Event Receive Execute** và event này sẽ gọi **Get All Actors of Class**.
25. Class Actor phải là **Dog House** và chúng ta sẽ chỉ có một trong một màn chơi. **Get All Actors of Class** sẽ trả về **OutActors**; lấy mục chỉ mục đầu tiên từ mảng này.
26. Hãy chuyển **OwnerActor** từ **Event Receive Execute** của chúng ta sang **DogController** và sau đó thực hiện **Move to Actor**.
27. Điều cuối cùng chúng ta muốn gọi là **Finish Execute** và trả về **bSuccess** là *true*:



Tóm tắt

Chương này chắc chắn nặng nề hơn với phần hướng dẫn, và các chương sau cũng vậy. Trong chương này, tôi đã trình bày cách chúng ta có thể làm cho AI của mình phản ứng hiệu quả với những con tốt khác và cả cách di chuyển đến các vị trí mục tiêu khác nhau. Chúng ta cũng đã sử dụng EQS một thời gian ngắn để giúp chú chó của mình chọn các địa điểm đánh hơi chiến lược. Trong chương tiếp theo, bạn có thể mong đợi sử dụng EQS hơn nữa. Mặc dù việc phát hiện dựa trên sự kiện, chuyển động vẫn được thực hiện bởi cây.

Trong chương tiếp theo, chúng ta sẽ xem xét các thành phần được xây dựng trong Unreal Engine 4 để cảm nhận AI khác và có thể giúp chúng ta đạt được AI linh hoạt và nhạy bén hơn.