Exercise – Behaviour Trees – Part 2

The goal of this tutorial is to further extend the capabilities of your Behaviour Trees. This tutorial builds off of the previous tutorial on Behaviour Trees. You will need to have completed the previous tutorial before continuing with this one.

Exercise 1:

In the last tutorial we implemented the basics for a Behaviour Tree including the base Behaviour class, Composite, Sequence, and Selector classes. We will now add Decorator behaviours, specifically one that inverts the result of its child behaviour, and one that logs a message when it executes its child behaviour.

Decorators are behaviours that wrap around an existing behaviour, forming a type of Composite behaviour that typically has a single child behaviour. Decorators have many uses and add extra functionality to an existing Behaviour Tree code-base without requiring modification to existing classes.

Simply implement the following two behaviours, based off the pseudo-code taken from the lecture and try and incorporate the Decorators into your previous Behaviour Tree designs:

```
//the interface class for all behavior nodes
class InverseDecorator : IBehaviour
    IBehaviour child

    func execute(agent)
        result = child.execute(agent)
        switch result
            case Success: return Failure
            case Failure: return Success
```

```
//base class for sequence and selector nodes
class LogDecorator : IBehaviour
    IBehaviour child
    string message

    func execute(agent)
        print message
        return child.execute(agent)
```

Exercise 2:

Not all behaviours can return a result instantly; some behaviours need a way to return that they are still on-going, or pending. The reason they need to return before completing is sometimes a task will take a certain amount of time and we don't want our entire program to freeze until the behaviour has returned its result.

For example, we may have an A* behaviour that returns success once the agent has reached the end of the path. In this example the behaviour would execute, move the agent along the path, and if the agent hasn't reached the end then the behaviour would return a pending result. Then the next time the tree is executed it will jump straight to the previously pending behaviour and continuing executing it until it returns either success or failure.

The lecture covered a way in which we can add the ability for behaviours to return pending. Using the pseudo-code from the lecture as a guide, attempt to modify your Composite classes, including Selectors and Sequences, to include the ability to return pending:

```
//enum for behavior result (bool would have worked before, but no longer)
enum BehaviourResult
    Success
    Failure
    Pending //add this
```

```
//updated composite node with an optional var for a pending child
class Composite : IBehaviour
    list childBehaviours
    IBehaviour pendingChild : null

    func execute(agent) = 0
```

```
//updated selector node to handle the case of a child returning PENDING
class Selector : Composite

    func execute(agent)
        child = pendingChild  //also make these changes to the Sequencer class
        pendingChild = null

        if child is null
            child = childBehaviours.first

        where child <= childBehaviours.last
            result = child.execute(agent)
            if result is Success
                return Success
            else if result is Failure
                child = next child
            else if result is Pending
                pendingChild = child
                return Pending
        return Failure
```

References:

- An excellent, in-depth explanation of behaviour trees can be found on gamasutra here.
- AltDevBlogADay also have a video tutorial on behaviour trees here.