# Worksheet – Behavior Trees

## Overview

### Objectives:

1. Create Behavior Tree

## Preliminary

Primarily this tutorial work will require the use of a C++ IDE. Visual Studio is the recommended software to use although any C++ development environment should be sufficient.

If you need help setting up the software, we have a tool called AppsAnywhere that allows you access to some of the software we use. You can use this to install some of the software we use. <https://myapps.abertay.ac.uk/>

Use the link in MyLearningSpace to download the “*Behaviour Trees - C++”* file. Unzip and open the Visual Studio Project.

## Building a Behavior Tree

The code for this project only requires us to determine the functions for the leaf nodes. Selector nodes have pre-defined behaviour which can be hard coded, meaning we don’t need to have custom functions for every node in our tree.

For example, a **FallbackSelector** is going to work the same no matter what child branches it has.

Each custom function for the leaf nodes has the same general format. It must return a type called **NodeStatus**  and must take as an argument a pointer to the node that owns (or is calling) the function in the first place.

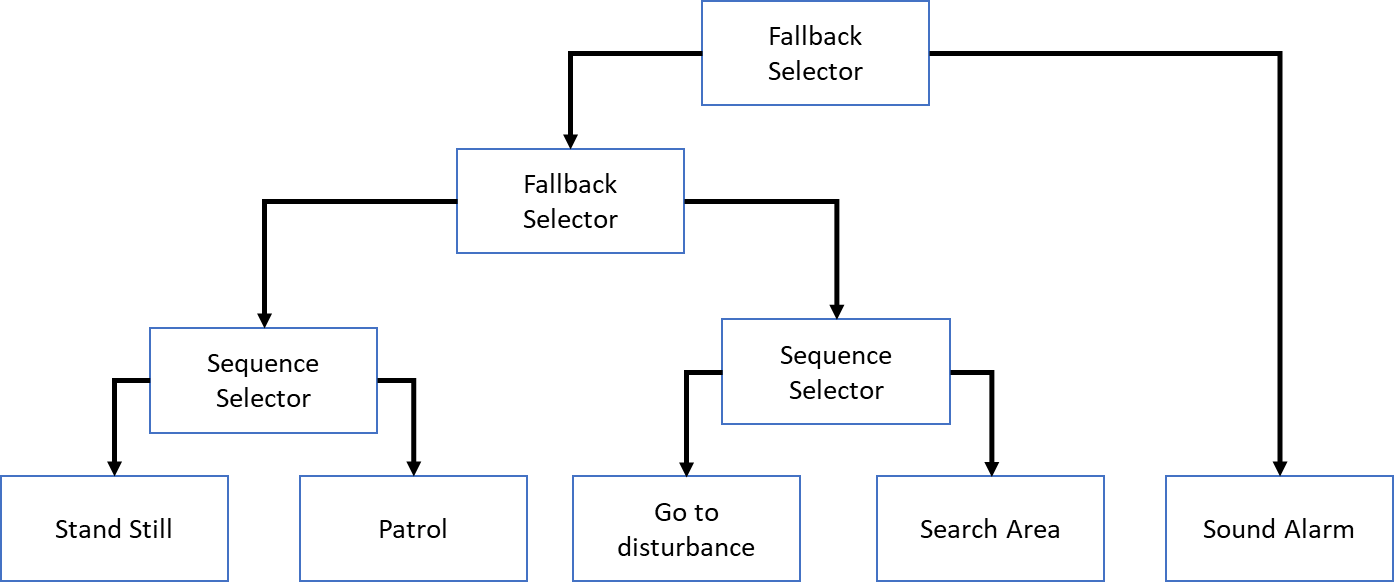
E.g.



### Model the Behavior Tree

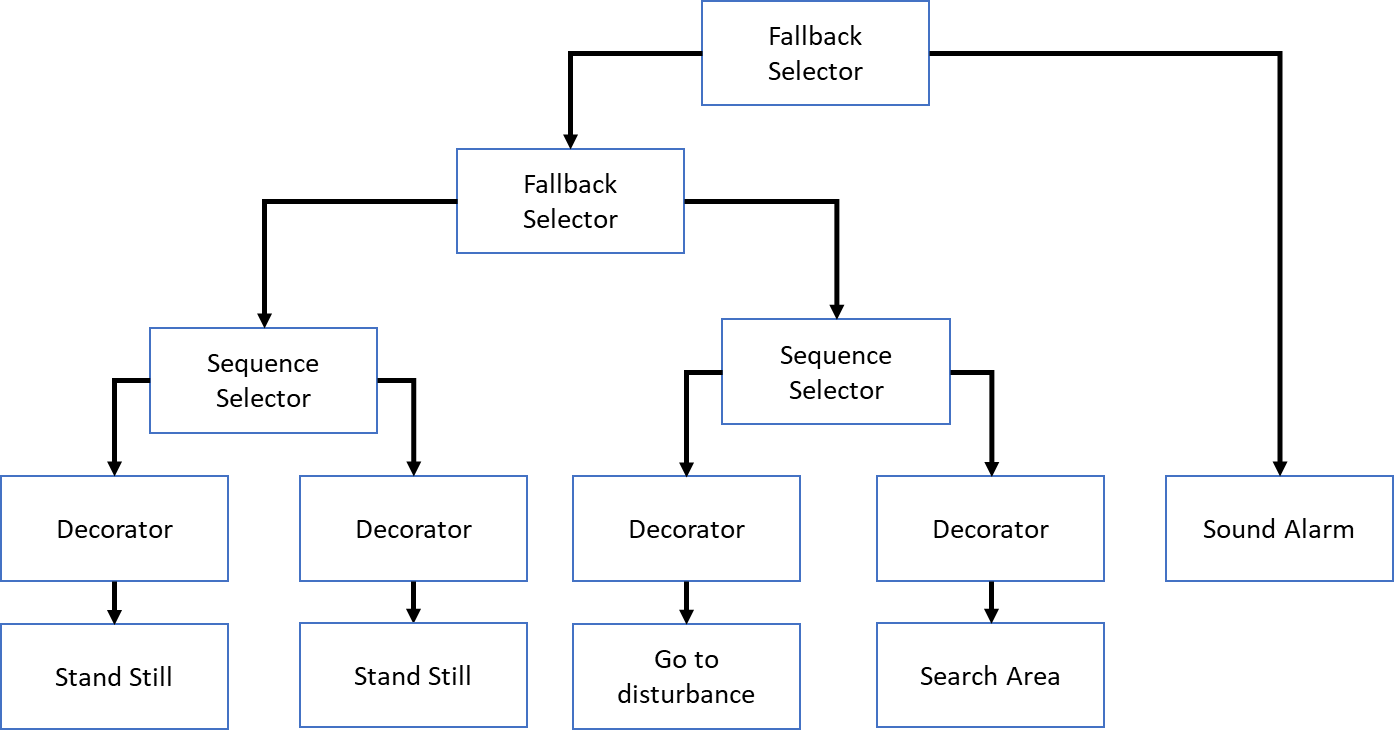
I have a stealth game where the player has to get past guards without being noticed (no combat). I want the guards to wander and patrol the area until they hear a disturbance. If they hear one, they must go and investigate. If they hear another disturbance whilst searching, they trigger the alarm.

We might model it like this:



Using fallback and sequence selectors, we could model this behaviour. The guard would alternate between standing still and patrolling. Those tasks will fail if they hear a disturbance at any point. The fallback selector moves onto the second sequence selector, which will have the guard path-find to the disturbance and search the area. If that is successful, we restart to first selector. If it enters a fail state, the root node will move onto its second main branch, which directs the guard to sound the alarm.

As our code is just going to be messages, we need to have timers to tell us when events occur. We will need to add **Decorator** nodes to leaf of the leaves, just for time management.



### Coding Using BTLib

#### Root node

Our first job is to create the root branch. Add the following code to the *main()* function in **main.cpp** in the BehaviorTrees project.



#### Layer Two

We can now create the next layer, which is another fallback selector, and a single leaf now. First we need to create the leaf nodes function which we place above our main:



This function makes the game print out a message saying the alarm is being raised and that the node has failed. Don’t worry if this seems strange, this is intentionally so that this forces the root node fails, which is what triggers the game to end.

We can now create the second layer nodes, so another fallback and our single leaf node:



Now its time to add them as child nodes to the root now. We can do this like so:



#### Layer Three

The third layer has to be the two **SequenceSelector** nodes. Like the **FallbackSelector**, we don’t need to add any custom functions so we can create and add them to the tree like so:



#### Layer Four

The fourth layer is made up of **Decorator** nodes. As we don’t have in-game events, we need to time each of our remaining leaves. We can do this like so, with each value being passed into the constructor being the number of ticks it will let the leaf run for:



#### Layer Five

Finally, we can create each of our leaves. As stated, each leaf needs its own function to run. Copy and paste the following code:



Each function starts with a check to see if it can find the ‘disturbance’ data on the Trees blackboard object. If it can, it deletes it and then fails, propagating this result up the tree. This is where the **FallbackSelector** nodes come in handy.

We can now create our leaf nodes and add them to the tree:



#### Main Game Loop

Our last task is to create a game loop to manage all of this. Ideally, the input to the Behaviour Tree would be handled by in-game events, but as we cannot do that, we will have to simulate it with keyboard presses.

Lets start with a while loop that represents the game loop and add some very simple keyboard detection. The input code were are using is VERY, VERY basic, so don’t do this in a proper application, but it will be fine for this experiment.



Here, we check if a keyboard key has been hit and we capture that key (this is done just to clear the input buffer. We could get the key value from \_getch() like so: char key = \_getch(); but we don’t care at this point. All we need to know is when any key is pressed). Once a keypress is detected we print a message for our game, and we tell the blackboard to add a new variable to its database. The blackboard now has a variable called ‘disturbance’ and it has a value of 1. Our leaf nodes from earlier can now read this from the database and fail if they need to.

As the printed a message we will also call the sleep function just to keep the timing the same.

We can now run the BehaviorTree root node. We do this like so:



This runs the root node, which will start one of its branches, which will propagate down the tree until it hits a lead node. Each time we call *.tick()* on our root node, its should return one of three results to our gameloop.

1. NODE\_RUNNING -> The tree is running. Nothing to do here.
2. NODE\_SUCCESS -> The tree has run through a loop of behaviour, no errors have managed to backpropagate back to the root. We can reset the tree and run it again.
3. NIDE\_FAILURE -> As our alarm node returns a fail result when it runs, this will back-propagate to the root. If the root returns this result, it means the player is captured and the game is over.

So we need some basic logic to manage this:



If the root returns NODE\_SUCCESS, reset. If it returns NODE\_FAILURE, break out of the game loop.

We should also add some delay timings as well, just to make sure it doesn’t print out too many messages.



#### Run the Example

The example is very simple. The tree should print out messages of what the AI is doing. To simulate an in-game event, press any key on the keyboard. Whilst the AI is searching for the player, press the key again to have the AI find the player and end the game.