Squad Behavior System

The Squad Behavior System (SBS) class takes a pair of undefined types <T, W>.

The **AddToSquad(IReGoapAgent<T, W> agent)** is an important method that Agents in the GameScene will call upon awakening, in this case, inside Awake() of classes that inherit ReGoapAgent. Calling this method allows agents to send references of themselves to the SBS instance in the GameScene, and the method will assign these agents to squads and the Dictionary of living agents. If the agent is not already inside the *aliveAgents* dictionary, then the *squads* list will be iterated to see if any squad hasn't reached maxed capacity and if there are any members within proximity, otherwise, the agent will be assigned its own squad of itself.

Inside of AddToSquad(), a new ISquad<T, W> is created with the *squadType* that holds the retrieved Get Component of type ISquad, is called to Instantiate itself with the agent added to the members list of the new instantiated ISquad<T, W>.

[***How To***] Define a class that inherits the Squad Behavior System <T, W> class, for example, SquadCoordinator : SquadBehaviorSystem<string, object>. Next override the following functions:

Squad

[Squad] Behaviors

Behaviors are handlers to give agents, from some squad, orders to fulfill in order to accomplish the behavior. Behaviors have a name, a priority, and methods to check if the behavior is warranted, **IsWarranted()**, and methods to check if the behavior is finished or failed: **IsFinished()**; and **IsFailed()**. To use defined Behaviors, they must be added as components to the GameObject that also possess the only Squad Behavior System component in the Game Scene. These Behaviors are collected by the SBS into a list that is later sorted based on the priority of each Behavior. Behaviors of higher priority are near the end of the list, and are the first to be checked if the Behavior is warranted to a squad.

This hierarchy of Behaviors based on priority dictates what behaviors are first checked to be warranted for each squad. Certain Behaviors are a higher priority than others, for instance, GetToCover Behavior would likely need to be best fit Behavior for a gunfight rather than Patrol Behavior. Even if the squad is not in a gunfight, rather be prepared than not. If the Behavior is warranted, then a copy of the behavior is made and assigned as the *Current Behavior* for the squad.

Inside the Squad Behavior System class upon Start(), all squads are iterated and a new <u>Valid High Priority</u> Behavior is assigned to each of them. Next,, within Update(), the squad list is iterated and we check if the squad has a behavior, or if either the behavior failed or finished with **IsFailed()** and **IsFinished()** respectively. Upon finishing or failure, the Behavior will Remove all Orders via **RemoveOrders()** methods, as there are instances where some orders aren't removed such as *FollowOrder*. Afterwards, the process of determining a new behavior is repeated.

If the Behavior did not failed or finished and the squad is assigned a behavior (instances that a squad doesn't have a behavior even after Start(), includes new squads resulting from merging or new agents that spawn into the GameScene), then **GiveOrders()** is called, and orders are given based on defined conditions to certain agents. It really depends on how GiveOrders() is redefined by a derived class of the **Behavior** class that inherits MonoBehavior and the **IBehavior** interface. For example, the **GiveOrders()** inside the Patrol Behavior : Behavior<string, object> is a very long extensive code that considers many different conditions before assigning orders to specific agents. What works for **GiveOrders()** inside Patrol Behavior may not work for another derived Behavior class, hence why **GiveOrders()** is open to developers' implementation.

[***How To***] Defined a class that inherits Behavior, for example, Patrol Behavior : Behavior<string, object>. Create some private constructor for the class that can be called by public helper methods in order to create clones of the class. Look inside *Patrol Behavior* class as a reference example. Create whatever necessary instances and methods needed to check if a behavior is warranted, finished, complete, and how to give orders. Most importantly, override the IsWarranted(), IsFinished(), IsFailed(), and GiveOrders() methods, as they will be used by the SBS<T, W> class.

Orders

Orders act as data banks of necessary information for agent's to consider in order to appropriately respond to complete the order. Defined *behaviors* are what set the Order object into the agent's memory state as <"order", IOrder someOrder>. The OrderSensor component of the Agent will detect in the agent's memory state for any orders by searching if the memory state contains key "order", hence one order at a time for each agent. Upon sensing an order, it will get the associated IOrder object, and call its method every Update() on whether the order is done, **OrderDone()**. Upon true, the Order Sensor will proceed to remove the "order" condition from the Agent's memory.

Every Order has a string *Name* and a private Dictionary<T, W> *orderData* which acts as a bank of all necessary information to complete an order. Associated helpful methods include **GetName()**, **GetData()**, and **SetData()**. Necessary information is added into the *orderData* bank inside some related Behavior class after constructing the Order.

Goals get the name of the order from memory and set themselves with a higher priority. This responsive order Goal remains valid as long as OrderDone is not true and it's not removed from agent memory (either due to the behavior failed or finished, or order finished before already, but hadn't been removed yet). Also independent threat level of the Agent is considered too, as certain orders are ignored if they don't help the agent in certain predicaments.

Examples of Orders are the **FollowOrder** and **PatrolOrder**, these defined orders inherit Order<string, object> making all T and W types in Order class of type string and object respectively, meanwhile Order inherits Monobehavior and IOrder interface; though Order doesn't need to be added as a component to a GameObject to be used. In PatrolOrder, OrderDone() is overridden, and it returns true if its conditions are satisfied, which it utilizes the information set in *orderData* by some Behavior to do the checks.

FollowOrder and PatrolOrder are ignored if the agent's threat level is too high, so their responsive goals won't be valid and have a lower priority!

[***How To***] To make your own Orders, inherit the Order class, and override the OrderDone() method to define what conditions need to be met in order for an order to be finished and be removed from agent memory.

Order Goals

Note that a ReGoapGoal has a name, an associated plan set by the planner, but no plan prior planning or after if not the higher priority valid goal. Inside the planner, we call **IsGoalPossible()** and if not then it's not considered for planning. After this filtering process in the planner of non-valid goals and blacklisted goals, the goals are sorted based on priority by calling **GetPriority()** for every goal. Goals are represented as a ReGoapState in which the defined goal will set conditions within its state representation that need to be fulfilled. This goal ReGoapState will be converted to a node that is the root for expansion and actions will need to fulfill the goal state conditions and chain themselves to a state in which there are no conflictions (matching key conditions, but different values) with the current state of the retrieved prior planning.

There is an intermediary class, ReGoapGoalAdvanced that upon every Update will try to get the agent to consider the goal if it is a valid goal of higher priority active or inactive. If there's planner (planner is referenced to all goals during planning process, so goals of specific agents reference the same planner), and the planner isn't processing (planning) and the warnCooldown threshold is surpassed, then do the following:

1. Get the current goal in the planner.
2. Get the plan of the current goal if there is a current goal
3. Determine if this goal's plan equals the current plan. Note that this goal's plan is null if it isn't the current plan of the planner, as the planner only determines and appoints a plan to a goal if it is validated (Can only be one)

4. And check if this goal is possible
5. If the goal is not active (doesn't have the same plan as the current goal), but it is possible, then warn the agent that this goal is possible which the agent will compare the priorities of this goal and current goal agent works on.
6. If the goal is active, but no longer possible, then definitely warn the agent, but this goal could still pass as the current goal as long as the priority is the same, unless the priority is edited based on IsGoalPossible().

Defining goals that respond to orders, define how the goal can be valid. Remember that every Order has a specific name, so have its respective Goal check within the agent's memory to try and see if it has a **HasKey("order")**. If so, call **GetName()** on the *IOrder* value and see if it matches the required name for the goal to respond. Additionally, the responding goal will consider the agent's Threat Level meter, and if it's too high than its threshold, then the goal is not valid. The **GetPriority()** of Order Goals are by default 1, but if IsValid() returns true (we call **IsValid()** inside **GetPriority()**, then the priority returned is a high number.

Actions