Contents

Site: https://github.com/luxkun/ReGoap

Instructions by Luciano Ferraro (ReGoap Author)

2. Create a GameObject for your Agent
3. Add a ReGoapAgent component, choose a name (you must create your own class that inherit ReGoapAgent, or implements IReGoapAgent)
4. Add a ReGoapMemory component, choose a name (you must create your own class that inherit ReGoapMemory, or implements IReGoapMemory)
5. [optional | repeat as needed] Add your own sensor class that inherit ReGoapSensor or implements IReGoapSensor
6. [repeat as needed] Add your own class that inherit ReGoapAction or implements clReGoapAction (choose wisely what preconditions and effects should this action have) and implement the action logic by overriding the Run function, this function will be called by the ReGoapAgent.
7. [repeat as needed] Add your own class that inherit ReGoapGoal or implements IReGoapGoal (choose wisely what goal state the goal has)
8. Add ONE ReGoapPlannerManager (you must create your own class that inherit ReGoapPlannerManager) to any GameObject (not the agent!), this will handle all the planning.

ReGoap Unity Example Explained:
Example Overview:

In the provided example there are four areas of interest: TREES (hasLog); ORES (hasOre); WORKSTATION; and BANK. 20 agents are spawned into the game scene upon play, each named BuilderAgent, and they accomplish a single goal of depositing an ax into the bank. Since there's only one goal, the PlannerManager will keep prioritizing that goal every time.

Consequently, the sequence of actions is to collect logs, then ores, then go to the workstation to produce an ax, then finally deposit an ax to the bank; rinse and repeat until the goal becomes invalidated.

Step 2: Create a Game Object for your Agent

In the provided example, the GameObject, is the BuilderAgent (Prefab Asset) under the FSMExample directory in the Unity directory of ReGoap. This prefab asset is the template of the AI character with the following components:

- Transform
- Builder Agent (Script)
- Builder Memory (Script)
- Multiple Resources Sensor (Script)
- Resource Bag Sensor (Script)
- Resource Bag (Script)
- Workstation Sensor (Script)
- Bank Sensor (Script)
- Gather Resource Action (Script)
- Add Resource to Bank Action (Script)
- Craft Recipe Action (Script)
- Generic Go To Action (Script)
- Collect Resource Goal (Script)
- State Machine (Script)
- Sms Idle (Script)
- Sms Go To (Script)

We will go over these components later in order of the instruction steps. After the prefab asset is defined, then in the game scene (ExampleFSMScene) in the GameObject Minions there is a component with the script Agents Spawner, and in this file are code to spawn in the AgentBuilder prefab by default 100 times with 0.1f delays in between spawns at the designated position described in the AgentBuilder prefab's transform. In the example, the max number is specified as 20 for the Builders Count field of the component and delay between spawns is 0.133 by the user. Agents per spawn can also be set by the user too.

With that, the GameObject character AI is created with all necessary components of sensors, memory, goals, actions, and finite state machine states. In the FSM example, there are only two states: idle and goTo. We will go over these components in the next sections. Lastly, planning is done by a class script that inherits the ReGoapPlannerManager class, and is added as a component to a separate GameObject excluded from the agents AI GameObjects.

FSM Example Overview & ReGoap Structure Summary:

NOTE: First we'll go over a brief overview of the ReGoap architecture and its important components that we have to consider when utilizing the architecture. Each component is usually composed of one or more classes with many helper methods, but we'll only go over what is relevant in this section concerning development of GOAP AI. If you like further analysis of specific components and methods not mentioned, then check out the core components respective header sections in this document. The table of contents section above has links for these headers for quick access, and other terms in this section will have similar links too. Afterwards, we'll dive into the provided FSM example and how it utilizes the core mechanics explained. Lastly, throughout this section, some terminologies or descriptions will be in blue bold or red bold in which red identifies concerns about the architecture relevant to future projects, and blue to highlight possible adaptations and improvements to be made.

Firstly, as the instructions declares, create a GameObject for an agent, and add a ReGoapAgent component to the GameObject. As discussed earlier, in the FSM example, a C# script named BuilderAgent inherits ReGoapAgentAdvanced which inherits ReGoapAgent and that inherits the interference class IReGoapAgent. IReGoapAgent takes a value pair of unknown type <T, W>, and in the FSM example the types specified are string and object - <string, object>. These generic types must be consistent across all classes of the agent as these are the types that will be used for planning processes. Additionally, in the FSM example, the builder agents are spawned into the scene by a separate script AgentSpawner which is added as a component to a separate GameObject called Minions in the scene.

The ReGoapAgent is essentially the central junction where all calls to other classes are made. Upon Awake(), or when the game starts, the component will refresh all necessary information for planning in which it retrieves lists of goals and actions, and the memory by getting all components added to the same GameObject that inherit their respective interface types.

Similar to ReGoapAgent, we define classes that inherit ReGoap core classes and then add them as components to the same GamObject as ReGoapAgent. For the memory, the FSM example has the BuilderMemory script inherit ReGoapMemoryAdvanced and that inherits the ReGoapMemory which inherits its respective IReGoapMemory. Notice a pattern yet? Only one memory component is required, and this helps the agent keep track of the game world in their perspective. I say perspective, but how does the agent observe and collect data about the world? This is done with sensors which are essentially helper methods to update the memory storage on how the agent perceives/senses the game world. Any number of sensor components can be implemented and they must inherit IReGoapSensor either directly or indirectly.

So how is the world defined and how is memory defined. Simply the world is defined as states and all states are handled as dictionaries of undefined key-value pairs. The keys serve as names/identifiers of conditions and values are the condition-values, and because they are undefined types is what makes ReGoap a generic architecture. Generally, string and object key-value pairs are sufficient for most defined conditions, for example, *'hasWeapon': true*. The world state is defined by these conditions, and memory is defined in the same way too. Memory of an agent doesn't need to record everything about the world, just what it perceives through its sensors. States are defined by IReGoapState related classes with three dictionaries, a main dictionary to record a state and two temporary dictionaries to hold copies, followed by helper methods that operate this state with other states. ReGoapMemory is just a variable of type ReGoapState, and it retrieves all IReGoapSensor related components upon awakening, and updates its memory state every interval it calls updates on all its sensors.

Goals are defined the same way as states, and rather than sensors, users will manually set the condition (key) and condition-value (value) pairs by calling the method Set() on the ReGoapGoal instance variable "goal" of type ReGoapState - again very similar to memory. This is done in the single goal in the FSM example, CollectResourceGoal when it overrides the Awake() function. Very important to consider, unlike the other core class so far, the goal component doesn't inherit ReGoapGoal through an Advanced class which handles the unity Update () function. For more complex projects, this must be inherited, so we automatically warn the agent if a goal is or not available, rather than leaving it up to the planner later.

Goals have a priority value for later calculations to determine agent's current goal, also placeholders for both a plan (queue of actions) and the planner. Further, we want to notify whether a goal is possible and by default that bool is set to true. The priority is of type float and is public, so it can be changed through serialization manually or with a method from a separate script, including the class that inherits ReGoapState. Two key methods to note are SetPlan() and Precalculations() in ReGoapGoal: Precalculations() set the planner to equal a IReGoapPlanner component; SetPlan() allows the goal to reference its own resulting sequence of actions from planning. Precalculations are done more so to allow a class to reference relevant classes before calculating a goal priority and plan, like for actions too as we will discuss later.

Earlier ReGoapGoalAdvanced was mentioned, in the Update() function, checks if a planner is referenced and not currently planning and that we pass a cool down threshold to warn about a goal's availability. Passing this check means we get the current goal of the planner and its plan if there's a goal. With these new values we compare them to the plan the goal references and the check whether the bool var WarnPossibleGoal is set to true or false. Either the goal is possible, but not active or the goal is active, but not possible anymore, then we pass this goal to the agent which will compare its current goals priority with this goal. If the priorities equal, then we are dealing with the same goal perhaps, so ignore the warning, same if the current goal is a

higher priority still, otherwise, interrupt the current action and calculate a new goal. This is done within planner.GetCurrentAgent().WarnPossible(this).

Moving on from goals, how can goal states be chained from the agent's memory state? Note that the goal state doesn't necessarily consider all conditions, only those that we desire the agent to change within its memory state. There are two major things to consider: the real world and the virtual world. The real world involves the goal states and the memory states of agents, while the virtual world is involved with planning where actions are being chained together by their preconditions and effects. Planning is generally associated with the virtual world, since copies of the goal states and agent memory are made and converted to nodes. Actions are converted to nodes and are chained together referencing their parent nodes (basically a linked list). These action nodes possess their states about the world which they copy from their respective parent's states that were then updated with the effects and preconditions of the action in that node. The root of this virtual world is a copy of the goal state, so an action node is discovered and references the goal node as its parent, then it references that parent's state as its own to then apply its effects and preconditions onto that copy. The effects would nullify certain preconditions of the goal state, but in order to apply an action's effects, then the preconditions need to be met, hence why they are also added. This chaining finally stops if: 1. The memory state is reached; 2. There is no viable sequence of actions to reach the memory state, so the goal cannot be validated. (IMPORTANT: This is what I noticed in ReGoapPlanner to which I'm confused, don't we already check if a goal is valid during goal calculation, unless this is a debugger for not having enough viable actions!) This virtual world establishes the warning made by the Author of ReGoap that action effects aren't written in memory!

Speaking of actions, defined action classes inherit the ReGoapAction class that inherits IReGoapAction. Important instances are the preconditions and effects, and since these are going to be applied to different states of type ReGoapState, then these instances are also defined as type ReGoapState. Another important instance is the cost of the action "Cost" and similar to "Priority" in ReGoapGoal, this can be edited manually or through some method that returns a float. Note that doing so we will have to override Awake() to calculate the cost and run an Update() to recalculate the cost if necessary, but there is NO ReGoapActionAdvanced, thus either create the class, or update within the action classes that inherit ReGoapAction.

Additionally, in the ReGoapAction class there are instances to reference a previous action and the next action if this action were to enter a chain of actions (queue of action nodes including goal node and agent memory node). There are also instances for the ReGoapAction class to reference an agent and another var of type ReGoapState referred to as settings which is **presumably** the current agent state the action will run on. More importantly, there are action delegates that will be passed methods and both take a parameter of type IReGoapAction<T, W> and these methods will be passed by the agent as call back functions if the action were to

complete or fail. Lastly, there is a bool var, interruptWhenPossible, that will call for an interruption of an action when set to true, and this is especially important when the planner calculates a new goal and subsequent plan, or the action were to become invalidated.

There is a lot concerning action, so please look into the header section and C# file. There is also a run method that will be later called by the ReGoapAgent class, and helper methods IsInterruptable() and AskForInterruption() which deal with interrupting an action. By default all actions according to ReGoapAction class are interruptible and more interesting, there's a function that checks the procedural condition. Procedural condition checks are just ways to see if an action is valid to perform, in this case, however, just returns true. This could be related to the warning that this architecture will not deal with false preconditions. Depending on the future development of the project, this could be an issue!

Returning to the ReGoapAgent class, we are now knowledgeable of the agent possessing memory, set of actions, and a set of goals, including what these components offer in terms of functionalities. Remember that the ReGoapAgent is simply a junction that houses the necessary information for the planner to execute. After awakening (Awake()), the agent will calculate a new goal upon Start(). There are two instances that reject calculating a new goal: 1. The planner is currently planning already; 2. Did not exceed the calculator delay threshold. The purpose of the calculation delay is to check whether a goal is still of a higher priority or blacklisted, but there needs to be a delay whenever we update in ReGoapAgentAdvanced, otherwise, processing power will be more expensive to calculate a new goal every time.

If a new goal can be calculated, then to save processing power further, we only calculate goals that are not blacklisted/failed prior, and this is done through UpdatePossibleGoals(). Note that goals can become un-blacklisted after a set delay.

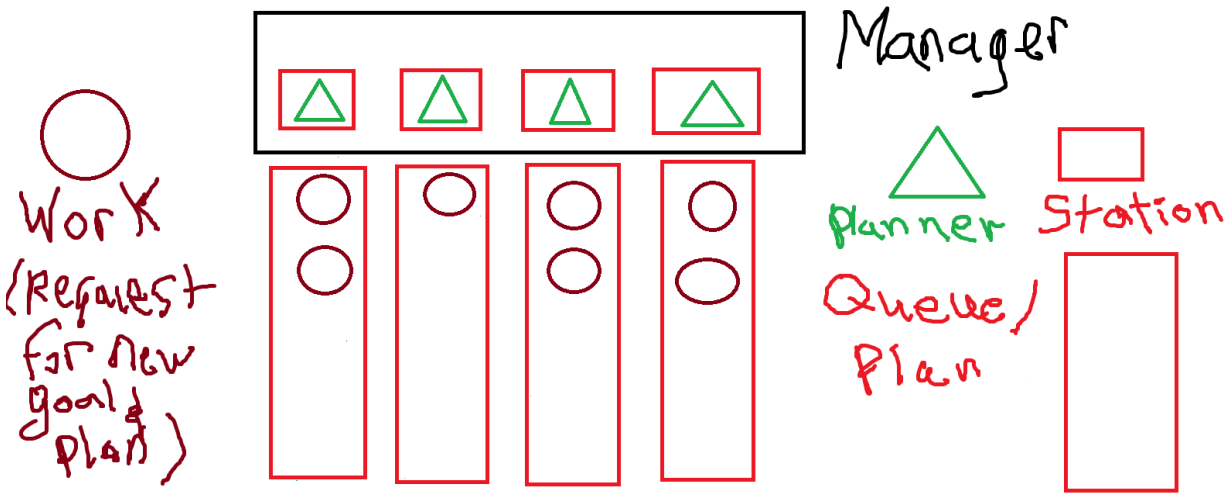
Before moving further on, there is another important instance var called "currentReGoapPlanWorker" of type ReGoapPlanWork<T, W>. This type is further defined in the ReGoapPlannerManager C# script and header section. Later in calculate goal after acquiring our list of possible goals, the "currentReGoapPlanWorker" equals the method call ReGoapPlannerManager<T, W>.Instance.Plan() with the passed parameters:

```
// !! We need to update the currentReGoapPlanWorker as it will plan wit
currentReGoapPlanWorker = ReGoapPlannerManager<T, W>.Instance.Plan(
    this,
    BlackListGoalOnFailure ? currentGoal : null,            // If we bl
    currentGoal != null ? currentGoal.GetPlan() : null,     // if the c
    OnDonePlanning                                          // Method t
    );
```

This method call performs the calculation of the goal and the subsequent planning to accomplish that goal. It takes the agent, the current goal if it failed, and the current plan associated with the goal if the current goal exists. Lastly, the fourth parameter is the method that will execute the resulting action plan after planning is complete - OnDonePlanning().

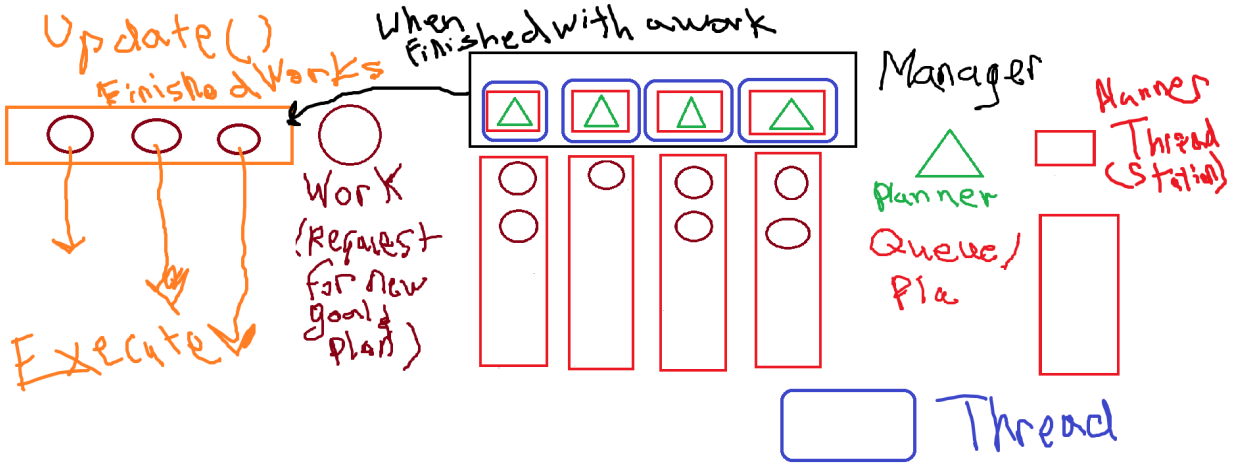
The type ReGoapPlanWork<T, W> is a struct within the ReGoapPlannerManager script and the script essentially handles planning for multiple agents via multithreading in which different planners of different agents are given their own work threads by the planning manager.

Let's break things down! It's established that there will be one single Planner Manager per Game Scene, and the Manager is being sent multiple jobs/works from different agents at a time, each agent requesting for a new goal and plan to be made. Let's say the Manager is a booth with a single employee called Planner, and all the agents are requesting their meals (the request for a goal and a plan - Work). A single employee Planner cannot handle hundreds of agents without burn out (exhausting processing power), so the Manager hires other Planner employees to handle the Work requests. And to better utilize the employees, the manager appoints each their own stations and they get their own lines of Works.



Inside ReGoapPlannerManager, there is a class for Works (ReGoapPlanWork); a class for the the Station (ReGoapPlannerThread); a class for the Manager (ReGoapPlannerManager); and a class for the planner (ReGoapPlanner). The Manager needs to manage the following, their employees (Planners) and the Stations (Planner Threads) they will work in. Additionally, the Manager needs to keep track of what works are finished and ready for take out by the customers (Agents). Another important thing to note is that the ReGoapPlannerThread is not the actual thread the planner will work in, rather it's a class that gives it the orders, since the queue/line is an instance variable of the class. The real thread or station is separate from the class and provided by the Manager, simply the real Thread executes the MainLoop() of the

ReGoapPlannerThread and the planner referenced by the ReGoapPlannerThread begins working. Here is an updated diagram of the considerations by the Manager:



All right, starting from the beginning, the agent calls for work in its own class:

```
// !! We need to update the currentReGoapPlanWorker as it will plan wi
currentReGoapPlanWorker = ReGoapPlannerManager<T, W>.Instance.Plan(
    this,
    BlackListGoalOnFailure ? currentGoal : null,            // If we b
    currentGoal != null ? currentGoal.GetPlan() : null,     // if the
    OnDonePlanning                                          // Method
    );
```

Then in the Manager class, its Plan method simply constructs and initializes a ReGoapPlanWork<T, W> object with the passed parameters, then appends it to the queue/line of a ReGoapPlannerThread (Station). Earlier in the Awake() function of Manager it set all the PlannerThreads, each with their own planner, inside different Threads, then it started those threads, so that the PlannerThreads MainLoop continuously check their respective queues for Work to give to their planner (employee) - PlannerThreads acts as a smaller version of manager to planner, so it makes more sense that the Thread is the station.

Finally the planner will finish the work, and append that finished work to the list of completed works that the Manager possesses. And for every update within the Manager class, if there is any finished work in the completed work list, then the manager will execute the resulting plan of that work with the callback function from the agent OnDonePlanning(IReGoapGoal).

Before discussing the execution of the plan, how does the Planner operate? Within the ReGoapPLannerThread, the planner is called to run on the properties of the given Work:

```
var work = checkWork;
planner.Plan(work.Agent, work.BlacklistGoal, work.Actions,
    (newGoal) => onDonePlan(this, work, newGoal));
```

The planner is given the agent that requested the work, a blacklisted goal or null, that agent's current plan or null, and the method onDonePlan() from the PlannerManager class (NOT the same as OnDonePlanning() from ReGoapAgent)! The PlannerManager's onDonePlan() is simply a formality that tells the debugging logger that the planning finished and the list of subsequent actions, but more importantly it will add the work to the Manager's list of completed works when called. onDonePlan inside planner will only take one parameter of type IReGoapGoal, because it's passed as (newGoal) => onDonePlan(this, work, newGoal).

The planner works with the given PlannerSettings which is simply the maximum iterations of nodes to inspect and the maximum nodes that can be expanded too, and whether dynamic actions are utilized and if planning can be exited early.

Planner goes through the list of possible goals from the agent, and these lists of possible goals are further filtered and only possible goals are considered. By default in the ReGoapGoal class, all goals are made possible by the public bool WarnPossibleGoal = true statement. This can be edited based on the goal component that inherits ReGoapGoal with a IsValid() method. Also the passed blacklisted goal is filtered out too if any. These possible goals are then sorted out from least priority to the highest priority. Afterwards the current state of the agent is retrieved from its memory state.

If not dealing with dynamic actions (actions with dynamically changing preconditions and effects), then brute force to check if all actions can reach the current memory state from the goal state by accumulating all action effects onto the goal state, then checking if the resulting state equals the current memory state. But since we're more than likely dealing with dynamic actions, then we do something similar but more complex.

Clone the goal state, then run the Astar algorithm as follows:

```
// Get the resulting returned child node that is the first action node step in the plan to reach the goal state, and
// assign it to variable leaf
var leaf = (ReGoapNode<T, W>)astar.Run(
    ReGoapNode<T, W>.Instantiate(this, goalState, null, null, null), // Get the root node to expand from
    goalState,                                                       // The goal state
    settings.MaxIterations,                                          // Maximum number of actions to iterate over
    settings.PlanningEarlyExit                                       // Whether the plan is to be exited earlier
    );
```

The ReGoapNode<T, W>.Instantiate(this, goalState, null, null, null) statement simply creates a node with references to a planner, parent, action, state, and goal, etc. The purpose of nodes is to hold a state that results from applying the effects and preconditions of the action the node

possesses onto the virtual state inherited from its parent node if any. The node will also have an associated cost from the sum of its parent's cost and the cost of its action. The utility of nodes allows us to expand from different virtual states based on the appliance of actions. AStar will use this expansion to find the path of lowest cost to reach the current memory state from the goal state.

AStar begins from the node representing the goal state, and it will expand from this node by creating child nodes resulting from actions whose effects satisfy a condition of the parent state. The child, if satisfying a condition of the parent, will have an update state of the parent in which its preconditions and effects are added, and the cost of the child will update. These children in AStar.Run will be added to a Fast Priority Queue in which nodes of lower value are added to the front to be dequeued first. Additionally, nodes that have been iterated will be kept in a record dictionary with the state as the key and its respective node the value.

Each node in the Fast Priority Queue will be checked if it equals the current agent memory state via IsGoal(). If not, then the node will be added to an explored dictionary. Afterwards, the dequeued node is expanded upon as a parent to repeat the process of adding child nodes both to the record dictionary and Fast Priority Queue. There are instances in which child nodes have states that have already been explored, so we continue to iterate the next child node, as explored nodes have already been expanded upon. Otherwise, check if there is a node with a similar state in the record dictionary. If this child node were to have a lower cost than the similar node found in the record, then don't bother expanding upon that similar node if there is already a cheaper alternative path with the child node. The similar node is delimited from the Fast Priority Queue, and the child node is added to the record and the Fast Priority Queue.

But if the child node were to cost more, then that parent node that is of the latest lowest cost node from the Priority Queue is not worth expanding if its child nodes somehow cost more than earlier expansions! Remember that every node's cost is the sum of the path cost plus the heuristic, in other words, the sum of the total actions cost so far at that point plus the number of conditions in the node's state unfulfilled after applying its action's effects and preconditions.

Whenever we apply an action's effects and preconditions to a node's parent state, we aim to remove the parent's state's effects and add the precondition to get that child node's resulting state. Inevitably the goal is to acquire a state that has no differences with the agent memory state, and that could mean the latest leaf node has an empty state or at least the lowest heuristic value, because note that some actions may not need preconditions to be in effect. The node closet with the memory state will have a higher action cost in its chain from the goal state, since we are going backwards.

We are aiming for a lower heuristic and lower sum of action costs to get a node of low total cost. But if an expansion of a node results with a single child sharing a state of a similar earlier iterated node from a prior expansion, and it's a higher cost, then we CANNOT waste time iterating over the parent's other children. We assume that the parent node isn't worth further processing power as it attracts actions that either, or both increase the subsequent state's heuristic and path cost values. Even if the parent has child nodes with states yet iterated, WE DO NOT WANT TO END UP WITH CIRCLE PLANNING.

BACK TO PLANNER:

```
// Get the resulting returned child node that is the first action node step in the plan to reach the goal state, and
// assign it to variable leaf
var leaf = (ReGoapNode<T, W>)astar.Run(
    ReGoapNode<T, W>.Instantiate(this, goalState, null, null, null), // Get the root node to expand from
    goalState,                                                       // The goal state
    settings.MaxIterations,                                          // Maximum number of actions to iterate over
    settings.PlanningEarlyExit                                       // Whether the plan is to be exited earlier
    );
```

This run will return a leaf node that is equivalent state-wise with the current memory state, as in it has no underlying differences in terms of conditions with different values with memory state's dictionary.

This leaf node is special, because if it isn't null, then a path can be calculated from it, as it links to a parent node, and so does every subsequent node in its chain, until the root - the goal node. But if it's null, then a plan couldn't be made, so the next hierarchical goal will go through the Astar process. With a leaf node, the agent's current plan can be compared with the leaf's plan, and if they're the same, then continue on with the agent's current plan and exit the planner, so that the agent can presume its plan. Otherwise, set the plan to the calculated current goal then perform a callback(currentGoal) (onDonePlan(currentGoal)) which will confirm the Work is complete.

Planner Manager during its Update() will execute all Works with their new goals and plans with the Callback(work.NewGoal) which is OnDonePlanning(work.NewGoal).

```
// The update function is where the completed works are then executed in which the Callback
// action (OnDonePlanning()) will begin to execute the resulting plan of the work!
protected virtual void Update()
{
    ReGoapLogger.Level = LogLevel;
    if (doneWorks.Count > 0)
    {
        lock (doneWorks)
        {
            foreach (var work in doneWorks)
            {
                work.Callback(work.NewGoal);
            }
            doneWorks.Clear();
        }
    }
    if (!MultiThread)
    {
        planners[0].CheckWorkers();
    }
}
#endregion
```

And so we return back to the agent with the to return type ReGoapPlanWork:

```
// !! We need to update the currentReGoapPlanWorker as it will plan wi
currentReGoapPlanWorker = ReGoapPlannerManager<T, W>.Instance.Plan(
    this,
    BlackListGoalOnFailure ? currentGoal : null,            // If we b
    currentGoal != null ? currentGoal.GetPlan() : null,     // if the
    OnDonePlanning                                          // Method
    );

return true;
```

In Agent, OnDonePlanning(IReGoapGoal<T, W> newGoal) will reset the currentReGoapPlanWorker. Next it will check if the newGoal is null, and null could mean that the same plan was founded or there has been an error in which no plan couldn't be calculated for any goal. The same plan means to continue with the plan. Otherwise, set the new goal as current goal and get the associated plan (A queue of ReGoapActionState) - action states posses the action and the resulting virtual state of that action - then have every action in the action state reference the agent - PostPlanCalcualtions().

Next do the following:

```
currentGoal.Run(WarnGoalEnd);   // che
PushAction();   // execute the next ac
```

currentGoal.Run(WarnGoalEnd) takes the WarnGoalEnd method as an action delegate, but the run method is currently empty, also, WarnGoalEnd simply takes parameter of type IReGoapGoal<T, W> and sends a warning if the passed goal is not current, then ask to calculate a new goal if done. This doesn't seem necessary, as new goals are calculated at delay intervals.

Lastly, we execute an action in the plan via PushAction(). Actions can be interrupted if a goal calculates itself to be possible, so it will ask the agent to ask for an interruption, and by default all actions are interruptible, but only when possible as the action can be mid-execution. Even if an action wasn't interruptible, then the next action will be interrupted and we still ask the current action to be interrupted; a new goal can't be calculated until then.

If the plan is completed then we calculate a new goal, hence why I say that currentGoal.Run(WarnGoalEnd) seems redundant. But if the plan isn't empty, dequeue the action state and run the action as follows:

```
// and grab the action from that ActionState class and have next hold it.
if (plan.Count > 0)
    next = plan.Peek().Action;
// The plan may not be empty after the first two checks, but if we still en

// Disable the action of the previous Action state
if (previous != null)
    previous.Action.Exit(currentActionState.Action);
// Pass the previous action state and the next action state (otherwise null
// pass the desired world state of the current goal, pass
currentActionState.Action.Run(previous != null ? previous.Action : null,
                              next,
                              currentActionState.Settings,
                              currentGoal.GetGoalState(),
                              WarnActionEnd,
                              WarnActionFailure);
```

Two methods of interest are WarnActionEnd() which resumes to push the next action once this method is called to end an action. WarnActionFailure() will check if the goal has been blacklisted, then add that to the goals blacklists, either way a new goal will always be calculated upon action failure. All action ends and failures will be handled by the Action class that inherits ReGoapAction. That can also be handed to the FSM by the action components when executing.

Again want to emphasize while the actions are running, the agent's Update() in ReGoapAgentAdvanced may cal to calculate a new goal if the current action state is null, so plan

is completed and the planner isn't running, thus the agent will forever be calculating while the game runs. Lastly, need to stress this out that every goal is competing to warn that they're possible and so agents need to check if they are of lower priority in order to warrant calculating a new goal. So two things are going off simultaneously after planning, goals are competing and actions are running sequentially.

Finally, the FSM will consist of a few states in which the actions will call methods from the FSM states and the FSM states will execute based on the resulting information and data from the action.

IMPORTANT NOTES:

- First, It is advisable to follow along with the code as you read, as this can be confusing without visual aid.
- For each section heading, in parentheses are the names of class and the >> symbol refers to the fact that the left-hand side class inherits the right-hand side class
- There will be mentions of action states or ReGoapActionState, but these are separate from actions, ReGoapAction, as action states are the states of the game world that are consequences of the associated action. For example, the action kill player will have the action state world with the condition that the player is dead is set to true. When defining action state worlds, we're only interested in defining the conditions that are affected by the action. Also, action states are referred to as nodes, when the A* search planner navigates through them to amass a sequence of action states that in total form the conditions of the desired world state, or goal state.

Agent (BuilderAgent >> ReGoapAgentAdvanced >> ReGoapAgent >> IReGoapAgent):

An agent is the ai character that is tasked to accomplish a goal through a sequence of plans. In this case, an agent, according to the **ReGoapAgent** script, has all the necessary information to calculate the next goal to plan a sequence of actions to accomplish said goal. Please note that calculating the next goal and planning the consequential action plan are done within the **PlannerManager** class, and the agent calls for that.

In the example: the **BuilderAgent** class is made and it inherits the **ReGoapAdvanced** class with the generic types <string, object> pair. What's important about this middleman class between ReGoapAgent and BuilderAgent class is that it holds the UnityFunction Update() in which it will calculate a new goal (CalculateNewGoal) if there is no current Action State and if there's no planning that's happening. This would mean that the previous goal failed (blacklisted), and so a new goal and consequential plan needs to be made.

When opening the ReGoapAgent class, you'll find that it inherits all abstract methods of the IReGoapAgent<T, W> interface and the IReGoapAgentHelper interface. Also, the first thing you'll notice are the public fields Name, CalculationDelay, BlackListGoalOnFailure, and CalculateNewGoalOnStart. Following this, we keep track of the time a goal was last calculated since the game started, then we have the important information:

- A list of goals (goals)
- A list of actions (actions)
- Memory (memory)
- The current goal (currentGoal)
- The current action state node (currentActionState)
- A dictionary of blacklisted goals, with IReGoapGoal as keys and associated value as float type - the value is the time when goal was blacklisted (goalBlackList)
- A list of possible goals (possibleGoals)
- A bool variable to flag if there are potential blacklisted goals in the goals set (possibleGoalsDirty)
- A queue of action state nodes to execute in sequence to complete a plan (startingPlan)
- A dictionary of different plan values (planValues)
- A bool variable to flag whether to interrupt the next transition from the current Action state node to the next action stated node in the queue (interruptOnNextTransition).
- A bool variable to flag if planning has started (startedPlanning)
- A ReGoapPLanWork<T, W> class (currentReGoapPlanWorker)
- A bool check if the planning process is being committed.

Additionally, there are the Unity Functions. The Awake function initializes the dictionary of black listed goals, sets the time a goal has been last calculated to -100, and then refreshes the set of goals, actions, and memory of the agent AI. With the calls on refresh, GetComponent(s) is called to get the respective components for each memory, goals, and actions. For example, RefreshGoalsSet(), gets every component of the AI agent, in this case scripts, that inherit the IReGoapGoal class interface. In The FSM example, the goal CollectResourcesGoal inherits ReGoapGoal which inherits IReGoapGoal, and this script as a component, is added to the BuilderAgent Prefab Asset Game Object which is the same Game Object Builder Agent script is added to. Consequently, by calling GetComponents<IReGoapGoal<T, W>(), we get CollectResourcesGoal and initialize that inside the list of goals. The same logic applies to memory and actions.

After the awake function, the start function follows and a new goal is calculated upon the method call CalculateNewGoal(true). What the method does is return true if a new goal is calculated, and false otherwise. If the planning process is already underway, or if we didn't force

a new calculation and are less than or equal to the delay to calculate a new goal, then return false. Otherwise, we update the latest time a goal is calculated. Set interruptOnNextTransition to false, and update all possible goals upon method call UpdatePossibleGoals. Remember that there is a list of possible goals and a dictionary of blacklisted goals, and an AI agent has a set of goals that needs to be filtered for failed/blacklisted goals to insert into possible goals list. Essentially, UpdatePossibleGoals cross references goals from the blacklist to the goals set and delimits blacklist goals from the possible goals set. The only way for a blacklist goal to be added to the list of possible goals, is if the associated value (time) is less than the instant moment update possible goals is being conducted.

Afterwards, set startedPlanning to true, and the currentReGoapPlanWorker will be initialized with a method call with the ReGoapPlannerManager class as follows:

```
// !! We need to update the currentReGoapPlanWorker as it will plan w
currentReGoapPlanWorker = ReGoapPlannerManager<T, W>.Instance.Plan(
    this,
    BlackListGoalOnFailure ? currentGoal : null,            // If we
    currentGoal != null ? currentGoal.GetPlan() : null,     // if the
    OnDonePlanning                                          //
    );

return true;
```

Further inspecting the parameters, we reference the ReGoapAgent with ***this***, then check if the currentGoal was blacklisted, if true then set the parameter to the currentGoal otherwise set to null. But upon starting the game the agent doesn't have a current goal yet, so in the third parameter, check if the currentGoal is null, and if so the parameter is null, else we pass the queue of ReGoapActionStates (sequence of actions state nodes) or the associated plan of the goal. Lasting, the fourth parameters passes a reference of the OnDonePlanning method call.

With OnDonePlanning, a class that inherits IReGoapGoal<T, W> needs to be passed; this is the new goal, or next goal. In this method the startedPlanning is set to false, as we would have finished planning at this point. Set the currentReGoapPlanWorker to default. Check if the new goal is null and if so, then check if the currentGoal is null too. These nested checks are done, because if a new goal couldn't be calculated, and no current goal exists, then NO plan can be made.

```
// Make sure that the current Action state variable doesn't point to any action node state as we
// are executing a new plan from the last.
if (currentActionState != null)
    // Disable the Action of the action state node first!
    currentActionState.Action.Exit(null); // disable action of the current action state node
currentActionState = null;                // we don't point to any action node state
```

Otherwise, set the current goal with the new goal, and do NOT point to any action state node, as a new plan needs to be made, disable the action of the action state node too. Get the plan of the updated current goal, then clear the dictionary of planValues. For every action state node in the plan, instantiate their actions to reference this ReGoapAgent.

```
// Run each action in the plan, and instantiate each action to reference this ReGoapAgent as their agent.
foreach (var actionState in startingPlan)
{
    actionState.Action.PostPlanCalculations(this);
}
```

Lastly, we run the current goal with the method WarnGoalEnd as a parameter:

```
// Run that we are about to finish the sequence to get to the goal.
// Use the Action<T> delegate to pass WarnGoalEnd method as a parameter.
// There will be a callback from the run method call if goal never equals the current goal.
currentGoal.Run(WarnGoalEnd);
PushAction();
```

What this does is that it will check whether the current goal equals the goal passed which will be an issue, since why continue executing the current goal if it's not the goal we see now! Otherwise, we call the CalculateNewGoal method, and remember that this method can return three things: false if we are planning; false if we have not yet exceeded the calculation delay for a new goal; and true if a new goal can be calculated. We still want to calculate new goals after every interval delay, as one goal in the possible goals set could end up having a higher priority than the goal we are currently executing actions for. Also, please know that calculating goals and plans are expensive, hence the delay intervals.

This is why for the planner manager, whenever calculating a new goal, we make sure to plan the current sequence of action state nodes remaining, in case if the node is not black listed, then we may presume with the original plan to complete the unchanged current goal.

```
// Check if the goal passed is the current goal, otherwise we may have a bug error. We warn the end of a goal so we can calculate a new goal.
public virtual void WarnGoalEnd(IReGoapGoal<T, W> goal)
{
    if (goal != currentGoal)
    {
        ReGoapLogger.LogWarning(string.Format("[GoapAgent] Goal {0} warned for end but is not current goal.", goal));
        return;
    }
    CalculateNewGoal();
}
```

If you were to inspect the run method for the ReGoapGoal class, it doesn't do anything, nor does it return anything. Simply the run method encapsulates it as an action delegate that must be passed a parameter of IReGoapGoal<T, W>, then this action is referenced by the callback variable. The referenced code, in this case, WarnGoalEnd, is expected to execute. Nothing else is done by the Run method.

```
//
public void Run(Action<IReGoapGoal<T, W>> callback)
{
}
```

Once that is done, we call the PushAction method which checks if an interruption is called to disrupt the transition to the next action state node in the plan from the currentActionState, and if so, a new goal is calculated (Note, we only interrupt a transition if a new higher priority goal is found). Otherwise, get the current goal's plan and check if it is zero, or complete, and of course calculate a new goal as a completed plan is a completed goal. But if the plan isn't completed, then we run the next action in the plan, hence the name PushAction(), dequeue it from the plan, and set that dequeue action state node as the currentActionState. Additionally, we reference the previous action state, and the next action state in the plan, as follows:

```
// If there are no interruptions and the plan has actions within it, then
else
{
    // have a pointer to the previous actionState, this will hold the currentActionState.
    var previous = currentActionState;
    // FIFO, Take out the ActionState from the front of the plan
    currentActionState = plan.Dequeue();
    // have a pointer to the next IReGoapAction<T, W> and have it hold null for now.
    IReGoapAction<T, W> next = null;
    // If there is still actions present in the plan, then peek the actionState, but not dequeue it
    // and grab the action from that ActionState class and have next hold it.
    if (plan.Count > 0)
        next = plan.Peek().Action;
    // The plan may not be empty after the first two checks, but if we still encounter the currentActionState to be null, then

    // Disable the action of the previous Action state
    if (previous != null)
        previous.Action.Exit(currentActionState.Action);
```

With this we run the action of the currentActionState as follows:

```
    currentActionState.Action.Run(previous != null ? previous.Action : null, next, currentActionState.Settings, currentGoal.GetGoalState(), WarnActionEnd, WarnActionFailure);
}
```

Here we pass the previous action and next action of their respective ActionStates, then the settings or consequent world state of the action, next the desired goal world state, and lastly the methods WarnActionEnd and WarnActionFailure.

```
// Function to run the action. This doesn't actually run the action but sets parameters
public virtual void Run(IReGoapAction<T, W> previous, IReGoapAction<T, W> next, ReGoapState<T, W> settings,
    ReGoapState<T, W> goalState, Action<IReGoapAction<T, W>> done, Action<IReGoapAction<T, W>> fail)
{
    interruptWhenPossible = false;
    enabled = true;
    doneCallback = done;
    failCallback = fail;
    this.settings = settings;

    previousAction = previous;
    nextAction = next;
}
```

What WarnActionFailure does is check if the action warned to fail is the currentAction, and if not, ignore the warning. Otherwise, check if we a goal has been blacklisted upon failure, then add that goal to the blacklist wit its value of the current time plus the error delay, then force start a new goal calculation:

```
public virtual void WarnActionFailure(IReGoapAction<T, W> thisAction)
{
    // we warn for failure, but the action is not the current action
    if (currentActionState != null && thisAction != currentActionState.Action)
    {
        ReGoapLogger.LogWarning(string.Format("[GoapAgent] Action {0} warned for failure but is not current action.", thisAction));
        return;
    }
    // The current action fails, so the Goal could be blacklisted if BlackListGoalOnFailure is flagged true, otherwise if the goal is
    // not blacklisted, then it can still be the new goal when calculating a new goal. If blacklisted, then it will not be a part of the
    // next calculation of the new Goal.
    if (BlackListGoalOnFailure)
        // Here when we blacklist a goal, we give it the time delay, so that it will be used later another time when calculating a new goal.
        // Note: remeber that goalBlackList is a dictionary, so we can get the value which is the time delay by just using currentGoal as a key.
        goalBlacklist[currentGoal] = Time.time + currentGoal.GetErrorDelay();
    CalculateNewGoal(true);
}
```

The WarnActionFailure is referenced as failCallBack in ReGoapAction class. Meanwhile, WarnActionEnd simply checks if the given action doesn't equal the currentActionState's action, and if it does, ignore the call (return). But if it does equal the currentActionState's action, then we have reached the end of the action, so push for the next action in the plan.

```
// Warn the end of an action, otherwise we dequeue the plan and push another action to run.
public virtual void WarnActionEnd(IReGoapAction<T, W> thisAction)
{
    if (thisAction != currentActionState.Action)
        return;
    PushAction();
}
```

Moving on from ReGoapAgent file, the ReGoapAgentAdvanced file, this file has the update function which checks whether the currentActionState is null, meaning that the agent is not executing a plan, and more than likely not even formulating a plan, so a new goal will be calculated.

Summary of ReGoapAgent:

Upon the awake function, get all the goals, actions, and memory components, then on start, proceed to force start a calculation for a new goal. Call the ReGoapPlannerManager to calculate the new goal along with its consequent plan of action state nodes. Also, prior to planning, always update the possible goals list to ensure goals that are blacklisted aren't considered when calculating for a new goal, but black list goal that exceed their error delays may be considered. The PlannerManager will consider whether the current goal has been blacklisted on failure, its current plan, and the ReGoapAgent function OnDonePlanning. Once the new goal is determined and the plan is made, we check if a plan was made first, and if yes, proceed to assign the current goal as the new goal and have all actions in the plan reference the agent. Lastly, run the current goal by having the currenGoal run the method WarnGoalEnd that calculates a new goal once a given goal is the same as the current goal. Next PushAction will execute the action from the next action node state to be dequeued in the current Goal's plan. This running of the action just sets parameters in the ReGoapAction class.

When we calculate a new goal in OnDonePlanning, a new goal won't be calculated as we aren't planning and the calculation delay has not been exceeded. So after currentGoal.Run(WarnGoalEnd) in OnDonePlanning is finished, we proceed to PushAction(). But if the calculation delay is exceeded, then we calculate a new goal and the PlannerManager again is given the current goal's plan, the agent, whether the goal is blacklisted, and the OnDOnePlanning method. Remember that we always calculate a new goal every delay interval, as calculating a goal and forming a new plan is expensive to process, but we need to make sure we prioritize the goal with the highest priority. Other goals can have dynamic priority, rather than static. If we get the same goal, then we continue on with the same plan.

Lastly, the update function in ReGoapAdvanced calculates a new goal if the currentActionState is null and the AI is not formulating a plan, meaning it's essentially not doing anything. Additionally, there are other useful methods in the ReGoapAgent folder, but they're self descriptive on their own, but comments are written too; please take a look on your own time.

World State (ReGoapState & ReGoapActionState):

- ***GoapState***:

  A ReGoapState consists of three dictionaries of key-value pairs of undefined types with a public default size of 20. When constructing a ReGoapState, initilize the dictionaries buffers A and B with the default size and set the dictionary values to equal bufferA. The Dictionary values will be the main dictionary to keep track of all world conditions.

  There are several methods relevant to these dictionaries, especially values, including the operator definition of + when adding ReGoapStates. Additionally, there are methods, such as HasAnyConflict which checks whether a given ReGoapState values

dictionary has key-value pairs that have matching keys with the dictionary of this ReGoapState, but different values. This method has another version that also cross references with a ReGoapState about changes.

Similar to HasAnyConflict, MissingDifference has two versions, the first simply counting the differences between the dictionary of this ReGoapState and the other, with differences defined as the same keys, but different values. Meanwhile, the other version adds another definition of difference based on a passed function delegate with parameters of types KeyValuePair<T, W> and bool. THe delegate function is referenced as predicate, and the predicate takes a key-value pair from the dictionary of ***this*** and the bool result, otherValue, of checking whether the other dictionary has the same key. The differences will be written to a reference to a ReGoapState called difference which will include the different key-value pairs in ***this*** dictionary. Lastly, ReplaceWithMissingDifference is very similar to the last discussed method, but instead the differences are written to the this.values dictionary and the previous this.values dictionary is kept to reference the previous buffer, as this.values is updated to the alternate buffer A or B.

There are other more self-described methods, but the remaining interesting are the Recycle() and Instantiate() method as they deal with a cache of states. The cache is a stack and it takes new states upon the call to Recycle(), while Instantiate can pop a state from the cache, then call the Init method on that pop state with a parameter either a ReGoapState or null. This will clear the states dictionary, and if a ReGoapState is passed it will copy that state's dictionary, but if null was passed, then the state will just have an empty dictionary.

- ***Action State***:
    - This is a much simpler class in which it references two object class types: IReGoapAction and ReGoapState, and refers to them as Action and setting respectfully. And its constructor initializes these public fields with the given action that inherits IReGoapAction and the given ReGoapState.
    - Remember that an ActionState is simply a node that has an action with an associated world state with conditions that are a consequence of said action, and such a world state is considered the setting.

Memory (BuilderMemory >> ReGoapMemoryAdvanced >> ReGoapMemory >> ReGoap Memory >> IReGoapMemory):

In Step 4, we're to add a ReGoapMemory component, BuilderMemory from the FSM example inherits ReGoapMemoryAdvanced which inherits ReGoapMemory, and lastly ReGoapMemory inherits its respective interface. Analyzing the ReGoapMemory, it's simply a ReGoapState, thus the memory is essentially a dictionary of conditions. Upon awaking the memory's ReGoapState, referred to as state, initializes to have a clear dictionary , and if we

every want to get the WorldState of memory, then simply return the state through the call GetWorldState().

This is very intuitive, because it again sees the world as a set of conditions, so the memory is just a more personalized ReGoapState to the character AI. But then, how does the character AI sense the game world? The answer lies with scripted sensors that helps update the character AI's personal memory, and the bridge between sensors and memory is conducted within ReGoapMemoryAdvanced. Within ReGoapMemoryAdvanced, the Awake() function is overridden to include the initialization of an array of IReGoapSensor objects and collect all IReGoapSensor related components to the array.

Next each sensor is called with method Init which just initializes every sensor component to reference this to the IReGoapMemory memory field.

```
public class ReGoapSensor<T, W> : MonoBehaviour, IReGoapSensor<T, W>
{
    protected IReGoapMemory<T, W> memory;
    // have the protected memory field equal the passed memory which again is
    // just a ReGoapState
    public virtual void Init(IReGoapMemory<T, W> memory)
    {
        this.memory = memory;
    }
```

In the Update function, we get an update about every sensor at specific time intervals, that way the Agent AI's memory is always adapting to the game world.

(Perhaps creatively, the senses do not have to be within the realms of realism…)

Sensors (BankSensor; MultipleResourcesSensor; ResourceBagSensor; ResourceSensor; and WorkstationSensor >> ReGoapSensor >> IReGoapSensor):

A sensor is simply a helper class that has access to an agent's memory, as whatever the character AI senses, the resulting data is recorded in memory. With this principle in mind, it makes sense (no pun intended) that the ReGoapSensor class has a protected IReGoapMemory variable; memory. Further, the class has the function Init that initializes the memory variable with a passed IReGoapMemory type, and the function GetMemory just simply fetches the memory instance.

The Update Sensor is undefined in ReGoapSensor class, rather the class that inherits ReGoapSensor must override the function, get the world state of the sensor's memory (the same shared memory with the agent), then with the ReGoapState defined helper methods, including set, specify how the sensor s to update the memory.

Let's examine BankSensor…

Actions (AddResourcetoBankAction; CraftRecipeAction; GatherResourceAction; & GenericGotoAction >> ReGoapActions):

An action should have the following, preconditions and effects on the world state, and these are defined as ReGoapState types. Additionally, actions have a cost, this can be either static

or dynamic, and this class has a public cost of type float, and this can be edited by outside scripts, or a method can return a type float to this public field.

Other instances that are protected include delegate Actions that take void methods that take parameters of type IReGoapAction which act as callback functions when the action were to finish or fail. Next are references to the previous and next actions in the plan, and then a reference to the agent, and next a flag called interuptWhenPossible which is set to true when we desire the action to be interrupted. Lastly, we reference the state of the action state node that action is associated with which is referenced as settings.

```
// Default name
public string Name = "GoapAction";

protected ReGoapState<T, W> preconditions;          // p
protected ReGoapState<T, W> effects;                // e
public float Cost = 1;                              // B

// An Action delegate is an encapsulated method that ret
// will fire off depending on what the type is passed in
protected Action<IReGoapAction<T, W>> doneCallback;
protected Action<IReGoapAction<T, W>> failCallback;
protected IReGoapAction<T, W> previousAction;
protected IReGoapAction<T, W> nextAction;

protected IReGoapAgent<T, W> agent;
protected bool interruptWhenPossible;

protected ReGoapState<T, W> settings = null;
```

An action should be able to be enabled and disabled, as not to affect the world state, and actions should only be enabled if they are queued within the planning process and are next to execute. Otherwise, an action should remain disabled. Within the Awake() function this appears to be the case:

```
protected virtual void Awake()
{
    enabled = false;
```

Peeking at the enabled definition, we get the following:

```
namespace UnityEngine
{
    //
    // Summary:
    //     Behaviours are Components that can be enabled or disabled.
    [NativeHeader("Runtime/Mono/MonoBehaviour.h")]
    [UsedByNativeCode]
    public class Behaviour : Component
    {
        public Behaviour();

        //
        // Summary:
        //     Enabled Behaviours are Updated, disabled Behaviours are not.
        [NativeProperty]
        [RequiredByNativeCode]
        public bool enabled { get; set; }
        //
        // Summary:
        //     Has the Behaviour had active and enabled called?
        [NativeProperty]
        public bool isActiveAndEnabled { get; }
    }
}
```

Now MonoBehavior inherits the Behavior class, and ReGoapAction inherits MonoBehavior, and by setting enabled to false, this ReGoapAction class can no longer be updated or any active code will no longer be active. Also, if we needed to check whether the action is active, just return this bool enabled flag.

```
// Check whether the action is active
public virtual bool IsActive()
{
    return enabled;
}
```

After this, simply instantiate the ReGoapState class objects of effects, preconditions, and setting which is just simply getting clear empty dictionaries of default size (see method definition again in ReGoapMemory file).

We'll go over the next methods briefly. PostPlanCalculations references a passed goapAgent, and PreCalculations has us referenced an agent field within the GoapActionStackData struct defined in IReGoapAgent class interface. Why reference different agent classes if an action is concerned for a single character AI using it? This is because agent's have different memory states and plans pre and post planning process… (I don't exactly know et)

There is a struct called GoapActionStackData that ReGoapAction inherits from its respective interface:

```
public struct GoapActionStackData<T, W>
{
    public ReGoapState<T, W> currentState;
    public ReGoapState<T, W> goalState;
    public IReGoapAgent<T, W> agent;
    public IReGoapAction<T, W> next;
    public ReGoapState<T, W> settings;
}
```

ReGoapAction has functions that return these fields. Besides the self descriptive methods, there are important methods worth mentioning. The method IsInterruptible() only returns true and the method CheckProceduralCondition returns true as well. These methods are better overridden by the class the user creates to inherit ReGoapAction, since an action can only be enacted if its preconditions are met. Some actions do not have preconditions, and some preconditions can be complex, including methods. The complexity of whether an action can be interrupted is also a factor for individual actions, hence why they are generic in this ReGoapAction class.

Examining the actions in the FSM examples

Goals (CollectResourceGoal >> ReGoapGoalAdvanced >> ReGoapGoal >> IReGoapGoal):

ReGoapGoal has public instances, including the name of the goal, its priority, the delay when the goal results in an error, and a warning for a possible goal.

```
// Name of the goal
public string Name = "GenericGoal";
// static default priority
// Need to optimize this to be dynamic...
public float Priority = 1;
//
public float ErrorDelay = 0.5f;
// Is there a goal to be calculated?
public bool WarnPossibleGoal = true;
// Remeber the two buffers and the values
protected ReGoapState<T, W> goal;
```

Following these are protected instances including a queue of action state nodes, or the plan, and the planner.

It's important to note that like memory, a goal is just a ReGoapState, so upon Awake(), it's instantiated to have an empty dictionary. Additionally the goal state can be recycled to a cache, again being a ReGoapState class, and this is encapsulated in the OnDestroy() method.

Another important function is the GetPriority() function which returns the priority float instance, and this can be static or dynamic through the use of a method or external script. IsGoalPossible is simply returning whether the goal is possible by WarnPossibleGoal. Also, the goal will have an associated plan given by the planner.

A goal is a desired world state with conditions an Agent needs to satisfy through actions. In the FSM example, the CollectResourceGoal simply overrides the Awake() and sets the string condition “collectedResource” paired with value true. This in turn sets the ReGoapState of goal to have a dictionary of <string, object> pair to possess <”collectedResource”, true>.
Planner (FSMExamplePlannerManager >> ReGoapPlannerManager | ReGoapPlanner >> IGoapPlanner | ReGoapPlannerSetting | FastPriorityQueue | ReGoapNode | AStar):

In the FSM example, the FSMExamplePlannerManager inherits the PlannerManager class to consider key-value pairs of types string and object respectfully.

- ReGoapPlanner >> IReGoapPlanner:
  This class handles the calculator of a new goal of higher priority and the planning of subsequent actions. It calls other helper classes and their methods, including ReGoapNode and AStar. ReGoapNode has methods to initialize an action as a node with a state and reference to a parent, as well as having an expansion method to get the resulting child nodes. AStar utilizes the expansion method to find and return a node that is equivalent to the current world state, while having a reference to a parent node.
  ReGoapPlanner uses the returned leaf from Astar.run to retrieve a queue of actions or the plan.
  - The planner class references a IReGoapAgent and IReGoapGoal, as we are trying to formulate a plan to execute a goal with the given relevant data stored with the Agent: memory, actions, goals, etc.
  - There is a bool flag to check if the planner has already calculated a plan for the desired goal.
  - There are two instances that we can only read, not manipulate: astar and settings. Settings is just a class in which we can set the maximum threshold of Actions nodes to expand from and the number of iterations to go through planning. Also, some actions may be dynamic in which they have preconditions and, or effects that can change during runtime or precalculations, so there is a flag that can be set to true if that’s the case. Another important note, a planning process can be exited early if the calculated new goal is the same as the current goal, hence a new plan doesn’t need to be made.
  - Astar is the astar * algorithm that searches through the action state nodes to form the sequence of plans. This does all the leg work.

- AStar:

  Astar begins with the goal state node and expands from it to queue children nodes to later expand from. Consequently, child nodes will be checked if they are similar to the agent's current state retrieved from memory, and if so, then the goal state has been fulfilled. AStar will keep track of states that have been expanded/explored, so we ignore child nodes that have already explored states. Additionally, AStar keeps track of states that have been iterated, so that any nodes with resulting expansions with child states that cost more than a previously iterated node (even of a different parent) of the same state will be detected. Upon detection, the parent node is no longer explored, and the next queued unexplored node is expanded upon. But if the cost is less than an already iterated node state, that state is reconsidered with a new node in the unexplored node queue (frontier). This will continue to repeat, until a resulting child node has a state that has no differences with the current state.

  A* search algorithm is of the best techniques for path-finding and graph traversals. At each step, it picks a node, on the map/graph it traverses, in accordance to the determined value ***F***. ***F*** is the sum of two parameters: ***G*** and ***G***. At each step it picks the node of the lowest ***F*** and processes the node. "***G*** is defined as the movement cost to move from a starting point to a given square on the grid, following the path generated to get there"; and "***H*** is defined as the estimated movement cost to move from that given square on the grid to the final destination," otherwise called the heuristic".
  https://www.geeksforgeeks.org/a-search-algorithm/#
  For the purposes of finding the low cost sequence of actions to achieve the desired world state from the current state, it would be too costly to go from the current state to the goal state. Instead, we begin from the goal state.
- ReGoapNode:

  The main importance of this class is that it allows the user to convert actions to nodes and chain them via preconditions and effects. Following this class are three important methods: instantiate and init to create a node; Expand() to expand a node to its children nodes; and CalculatePath() to get the queue of subsequent parent nodes from a leaf node.
  - A node has both a heuristic value and a movement value. Also, regarding this project, the nodes we're traversing are action state nodes along with settings or the consequent world state conditions altered based on their effects. Additionally, each node may reference a parent, in this case another node that has effects that link to their preconditions.
  - Additionally, the node should know the goal and the resulting current world merged with the goal, the planner, ad a total cost to traverse the action node state.

- FastPriorityQueue: Not going to explain this, just presume that it's a priority queue with helper methods that make searching faster and saves computing time. https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp/wiki/Getting-Started
- ReGoapPlannerSettings:
    - This class has four public serialized fields: PlanningEarlyExit (bool); MaxIterations (int); MaxNodesToExpand (int); and UsingDynamicActions (bool).
    - PlanningEarlyExit - Can exit the planning process if the calculated new goal is the same as the passed current goal, thus the consequent plan is already calculated
    - MaxIterations - Maximum number of nodes we may iterate through resulting from expansions of parent nodes.
    - MaxNodesToExpand - The maximum number of nodes that can be expanded, as different combinations of sequential nodes result with varying child nodes.
    - UsingDynamicActions - Some actions are dynamic, as in they have preconditions and effects that can change either during runtime or pre-calculations.

- ReGoapLogger (ReGoap.Utilities.ReGoapLogger) : This is simply a tool for debugging that outputs messages to the console, for example, warning if an action failed but the given action is not part of the current action state, etc (Don't worry too much about it).

- **FSMExamplePlannerManager >> ReGoapPlannerManager (Also discusses ReGoapPlannerThread<T, W> and ReGoapPlanWork<T, W>)**:
  Handles multithreading for agents and their planners. Make sure to only set this behavior to only one game object.

  In the agent, an instance var of type ReGoapPlanWork is made, simply this struct holds an agent instance, a blacklisted goal, a queue of actions (plan), an action delegate, and a new goal. The struct's constructor is very similar to the following method call at the end of CalculateNewGoal in ReGoapAgent:

```
// !! We need to update the currentReGoapPlanWorker as it will plan with a new goal
currentReGoapPlanWorker = ReGoapPlannerManager<T, W>.Instance.Plan(
    this,
    BlackListGoalOnFailure ? currentGoal : null,            // If we blacklisted a
    currentGoal != null ? currentGoal.GetPlan() : null,     // if the current goal
    OnDonePlanning                                          // Method that performs
    );

return true;
```

Let's inspect the ReGoapPlannerManager class, firstly it has a public static instance of type ReGoapPlannerManager and a public boo MultiThread that can be changed to true or false in Serialization. You can check the inspector on the ReGoapPlanner GameObject in the FSM example in the component FSM Example Planner Manager component that inherits ReGoapPlannerManager.

The public bool MultiThread, if set to true, will permit multithreading. Following these instances are an int var instance of the number of threads, an array of type ReGoapPlannerThread<T, W>, a list of type ReGoapPlanWork, an array of type Thread (More on threads in this website: https://www.tutorialspoint.com/csharp/csharp_multithreading.htm#:~:text=In%20C%23%2C%20the%20System.,threads%20in%20a%20multithreaded%20application), an instance to reference Planner settings from the ReGoapPlannerSetting class, and finally a debugging ReGoapLogger tool. The WarmUpCounts aren't entirely important, they just hold the maximum iterations of empty nodes and empty states to add to the caches upon WarmUp() of ReGoapNode and ReGoapState classes; those are later recycled.

```
public static ReGoapPlannerManager<T, W> Instance;

public bool MultiThread;
[Header("Used only if MultiThread is set to true.")]
[Range(1, 128)]
public int ThreadsCount = 1;
private ReGoapPlannerThread<T, W>[] planners;

private List<ReGoapPlanWork<T, W>> doneWorks;
private Thread[] threads;

public ReGoapPlannerSettings PlannerSettings;

public ReGoapLogger.DebugLevel LogLevel = ReGoapLogger.DebugLevel.Full;

public int NodeWarmupCount = 1000;
public int StatesWarmupCount = 10000;
```

Next within the Awake() function, WarmUps are performed on the ReGoapNode and ReGoapState classes. Next, there is a check if Instance is null which it should as there should only be one GoapPlannerManager. Instance will reference "this", and a new empty list of type ReGoapPlanWork is initialized for doneWorks.

Before moving on, let's discuss the ReGoapPlannerThread class:

- **ReGoapPlannerThread<T, W>** :

fff