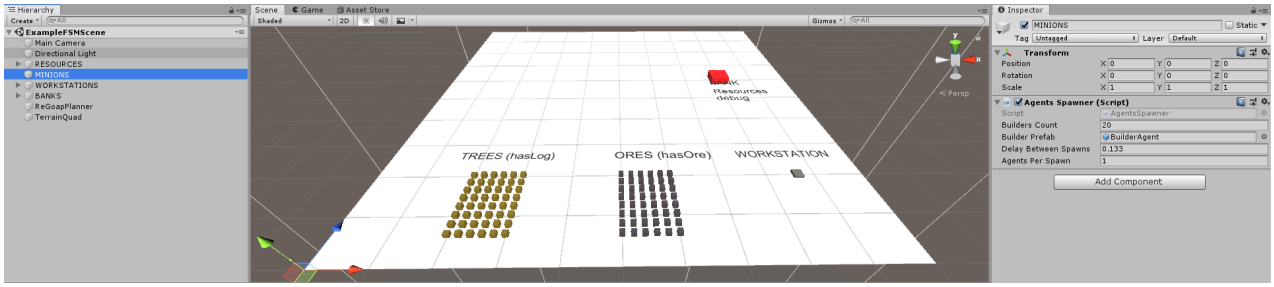
Step 1: Create a Game Object for the Agent. In this case the Game Object Object is Minions, then add a ReGoapAgent component to the Game Object, and this agent class serves as the central junction for all necessary data and methods to calculate a goal and subsequent plan. In the FSM example, the Minions Game Object has the script Agents Spawner as a component:



This script simply spawns multiple GameObjects instantiated with some Prefab (this case a script) into the game scene, until the maximum number of agents is met. There is a delay between spawns.
(AgentSpawner is inside the OtherScripts directory).

The Prefab is the defined class Builder Agent and that inherits the ReGoapAgentAdvanced class which inherits the ReGoapAgent class that inherits IReGoapAgent.
(Interfaces are helpful, as we can pass to methods parameters of some interface type, thus class objects that inherit these interfaces directly or indirectly can also be passed.)

Agent (ReGoapAgent & Advanced):

Agent has the following important data: a list of goals, list of actions, and its memory or representation on how it views the conditions of the world. Agent also keeps track of the current goal it has, the current action state it is in, what goals that have been blacklisted (goal can be blacklisted upon failure of a defined action, depending on how action is defined on failure), a list of possible goals, the plan or queue of actions.

```
public class ReGoapAgent<T, W> : MonoBehaviour, IReGoapAgent<T, W>, IReGoapAgentHelper
{
    public string Name;
    public float CalculationDelay = 0.5f;
    public bool BlackListGoalOnFailure;

    public bool CalculateNewGoalOnStart = true;

    protected float lastCalculationTime;

    protected List<IReGoapGoal<T, W>> goals;
    protected List<IReGoapAction<T, W>> actions;
    protected IReGoapMemory<T, W> memory;
    protected IReGoapGoal<T, W> currentGoal;

    protected ReGoapActionState<T, W> currentActionState;

    protected Dictionary<IReGoapGoal<T, W>, float> goalBlacklist;
    protected List<IReGoapGoal<T, W>> possibleGoals;
    protected bool possibleGoalsDirty;
    protected List<ReGoapActionState<T, W>> startingPlan;
    protected Dictionary<T, W> planValues;
    protected bool interruptOnNextTransition;

    protected bool startedPlanning;
    protected ReGoapPlanWork<T, W> currentReGoapPlanWorker;
    public bool IsPlanning
    {
        get { return startedPlanning && currentReGoapPlanWorker.NewGoal == null; }
    }
```

Agent uses the following above to keep track of itself, including what state it's in when executing a plan, whether or not it should interrupt an action it's currently performing, as well as, what goals are even possible prior calculating a new goal, hence the different list of blacklisted and possible goals.

When awaking, the agent refreshes its list of goals and actions, and its memory, then proceeds to calculate a new goal on start.

```
#region UnityFunctions
protected virtual void Awake()
{
    lastCalculationTime = -100;
    goalBlacklist = new Dictionary<IReGoapGoal<T, W>, float>();

    RefreshGoalsSet();
    RefreshActionsSet();
    RefreshMemory();
}

protected virtual void Start()
{
    if (CalculateNewGoalOnStart)
    {
        CalculateNewGoal(true);
    }
}
```

Here we calculate a New Goal:

```
protected virtual bool CalculateNewGoal(bool forceStart = false)
{
    if (IsPlanning)
        return false;
    if (!forceStart && (Time.time - lastCalculationTime <= CalculationDelay))
        return false;
    lastCalculationTime = Time.time;

    interruptOnNextTransition = false;
    UpdatePossibleGoals();
    //var watch = System.Diagnostics.Stopwatch.StartNew();
    startedPlanning = true;
    currentReGoapPlanWorker = ReGoapPlannerManager<T, W>.Instance.Plan(this, BlackListGoalOnFailure ? currentGoal : null,
        currentGoal != null ? currentGoal.GetPlan() : null, OnDonePlanning);

    return true;
}
```

Notice that the agent won't plan if already planning and if the calculation delay is not exceeded. But if it a goal can be calculated, then we set interruptOnNextTransition to false, then call UpdatePossibleGoals() and this filters out all blacklisted goals from the set of goals the Agent possesses:

```
protected virtual void UpdatePossibleGoals()
{
    possibleGoalsDirty = false;
    if (goalBlacklist.Count > 0)
    {
        possibleGoals = new List<IReGoapGoal<T, W>>(goals.Count);
        foreach (var goal in goals)
            if (!goalBlacklist.ContainsKey(goal))
            {
                possibleGoals.Add(goal);
            }
            else if (goalBlacklist[goal] < Time.time)
            {
                goalBlacklist.Remove(goal);
                possibleGoals.Add(goal);
            }
    }
    else
    {
        possibleGoals = goals;
    }
}
```

Back to CalculateNewGoal(), the last statement is a job request for the Planner Manager to calculate a new goal and its subsequent plan:

```
startedPlanning = true;
currentReGoapPlanWorker = ReGoapPlannerManager<T, W>.Instance.Plan(this, BlackListGoalOnFailure ? currentGoal : null,
    currentGoal != null ? currentGoal.GetPlan() : null, OnDonePlanning);

return true;
```

currentReGoapPlanWorker is of type ReGoapPlanWork which is a struct that mimics a job for the Planner Manager to perform.

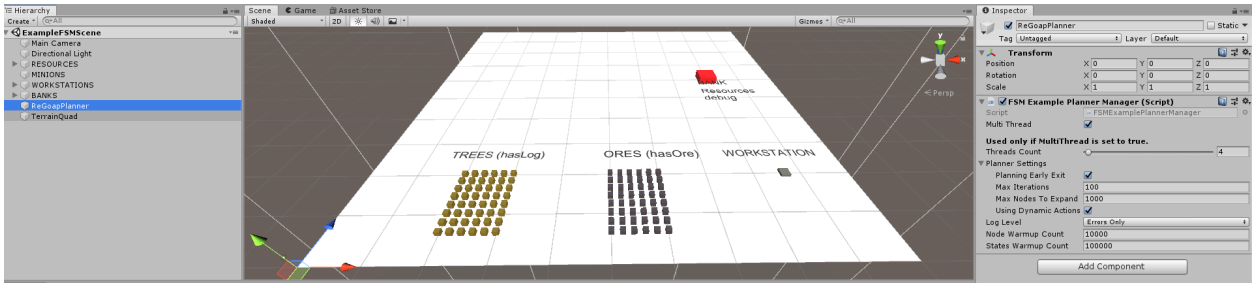
What's passed to the Planner Manager to run is the agent, the current goal if it's been blacklisted, and the goal's associated plan, and lastly a method defined inside ReGoapAgent that executes the subsequent plan after planning.

Planner Manager (ReGoapPlannerManager):

The Manager's job is to assign planners work and station them in separate Threads to perform their queue of works, thus each Planner has their own Thread, so processing power is less demanding on a single thread which would slow down.

```
// behaviour that should be added once (and only once) to a gameobject in your unity's scene
    public class ReGoapPlannerManager<T, W> : MonoBehaviour
    {
        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;

        #UnityFunctions
```

The Manager will keep track whether Multi Thread has been turned on or off, keeps track of its planners, the works that have been completed, the thread, and the settings that the planner is working with. Here's how the Planner Manager looks in the inspector as a component:



The class FSM Example Planner Manager simply inherits the ReGoapPlannerManager class, and that's it.

```
namespace ReGoap.Unity.FSMExample.Planners
{
    public class FSMExamplePlannerManager : ReGoapPlannerManager<string, object>
    {
    }
}
```

When the manager awakes, the doneWorks, threads, and planners lists are initialized, and if MultiThread is turned on, then all planners are initialized, and the threads are given each planner's main loop and are started, then added to the threads list:

```
#region UnityFunctions
protected virtual void Awake()
{
    ReGoapNode<T, W>.Warmup(NodeWarmupCount);
    ReGoapState<T, W>.Warmup(StatesWarmupCount);

    ReGoapLogger.Level = LogLevel;
    if (Instance != null)
    {
        Destroy(this);
        var errorString =
            "[GoapPlannerManager] Trying to instantiate a new manager but there can be only one per scene.";
        ReGoapLogger.LogError(errorString);
        throw new UnityException(errorString);
    }
    Instance = this;

    doneWorks = new List<ReGoapPlanWork<T, W>>();
    ReGoapPlannerThread<T, W>.WorksQueue = new ConcurrentQueue<ReGoapPlanWork<T, W>>();
    planners = new ReGoapPlannerThread<T, W>[ThreadsCount];
    threads = new Thread[ThreadsCount];

    if (MultiThread)
    {
        ReGoapLogger.Log(String.Format("[GoapPlannerManager] Running in multi-thread mode ({0} threads).", ThreadsCount));
        for (int i = 0; i < ThreadsCount; i++)
        {
            planners[i] = new ReGoapPlannerThread<T, W>(PlannerSettings, OnDonePlan);
            var thread = new Thread(planners[i].MainLoop);
            thread.Start();
            threads[i] = thread;
        }
    } // no threads run
    else
    {
        ReGoapLogger.Log("[GoapPlannerManager] Running in single-thread mode.");
        planners[0] = new ReGoapPlannerThread<T, W>(PlannerSettings, OnDonePlan);
    }
```

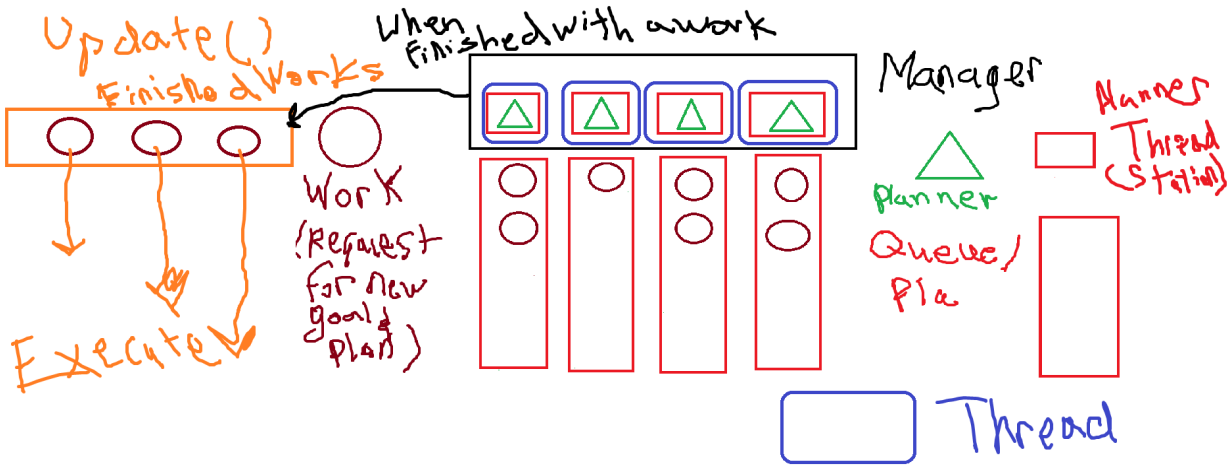
Before further discussing this, let's take a look at the Run method that was called in Agent to the Planner Manager. Note that Instantiate just lets the Planner Manager reference itself. With the passed information, the Planner Manager creates a Work/Job of type ReGoapPlanWork:

```
    public ReGoapPlanWork<T, W> Plan(IReGoapAgent<T, W> agent, IReGoapGoal<T, W> blacklistGoal, Queue<ReGoapActionState<T, W>> currentPlan, Action<IReGoapGoal<T, W>> callback)
    {
        var work = new ReGoapPlanWork<T, W>(agent, blacklistGoal, currentPlan, callback);
        lock (ReGoapPlannerThread<T, W>.WorksQueue)
        {
            ReGoapPlannerThread<T, W>.WorksQueue.Enqueue(work);
        }
        return work;
    }
}

public struct ReGoapPlanWork<T, W>
{
    public readonly IReGoapAgent<T, W> Agent;
    public readonly IReGoapGoal<T, W> BlacklistGoal;
    public readonly Queue<ReGoapActionState<T, W>> Actions;
    public readonly Action<IReGoapGoal<T, W>> Callback;

    public IReGoapGoal<T, W> NewGoal;

    public ReGoapPlanWork(IReGoapAgent<T, W> agent, IReGoapGoal<T, W> blacklistGoal, Queue<ReGoapActionState<T, W>> actions, Action<IReGoapGoal<T, W>> callback) : this()
    {
        Agent = agent;
        BlacklistGoal = blacklistGoal;
        Actions = actions;
        Callback = callback;
    }
}
```

In summary the Manager when called to run, creates the jobs for its planners to perform. Planners are called inside another class of type ReGoapPlannerThread:

```
{ // every thread runs on one of these classes
    public class ReGoapPlannerThread<T, W>
    {
        private readonly ReGoapPlanner<T, W> planner;
        public static ConcurrentQueue<ReGoapPlanWork<T, W>> WorksQueue;
        private bool isRunning = true;
        private readonly Action<ReGoapPlannerThread<T, W>, ReGoapPlanWork<T, W>, IReGoapGoal<T, W>> onDonePlan;

        public ReGoapPlannerThread(ReGoapPlannerSettings plannerSettings, Action<ReGoapPlannerThread<T, W>, ReGoapPlanWork<T, W>, IReGoapGoal<T, W>> onDonePlan)
        {
            planner = new ReGoapPlanner<T, W>(plannerSettings);
            this.onDonePlan = onDonePlan;
        }

        public void Stop()
        {
            isRunning = false;
        }

        public void MainLoop()
        {
            while (isRunning)
            {
                CheckWorkers();
                Thread.Sleep(0);
            }
        }

        public void CheckWorkers()
        {
            if (WorksQueue.TryDequeue(out ReGoapPlanWork<T, W> checkWork)) {
                var work = checkWork;
                planner.Plan(work.Agent, work.BlacklistGoal, work.Actions,
                             (newGoal) => onDonePlan(this, work, newGoal));
            }
        }
    }
```

Works are added to the queue of the Planner Thread, and the onDonePlan in the planner thread is just a formality method that adds the completed work to the doneWorks list and outputs to ReGoapLogger that the planner has completed the work. MainLoop will continuously run on the given Thread, until the planner stops upon the Manager disabling (Game Scene completed or Game ended). Every loop, check if the Planner Thread's queue is not empty, dequeue the Work and tell the Planner to run it. The following below is a diagram that captures the process:



Planners (ReGoapPlanner, AStar, and ReGoapNode):

What the Planner does is calculate a filtered list of goals, even though the agent updated possible goals not blacklisted, there can still be goals that cannot be valid, further even if

planning were completed, the goal may not have a valid plan (not enough actions to complete the goal). The planner then filters out these kinds of goals:

- Blacklisted goals (including the passed current goal of the agent if blacklisted)
- Goals not valid
- Goals that cannot be validated, due to insufficient actions for planning

The filtered goals are arranged in a list, with the lowest priority goals to the left and the highest to the right, and get a copy of the current state of the agent from its memory. There's a section of the code that can be ignored, because it just brute forces whether a goal can be reached from the current state by amassing all actions together. Next clone the goal state.

Essentially, we cloned the agent's current state and the goal state, because we don't want to edit how the agent actually perceives the game world, since this is just planning. When you brainstorm strategies, the brainstorming phase doesn't actually change the environment, unless you're an XMan like Charles Xavier. The planning can be compared to the Virtual world vs the Real World:

Real World	Virtual World

• Agent memory state
  ○ Record of what the agent perceives in real time
• Agent sensors
  ○ The only thing that should change the agent's memory ideally. What the agent perceives through its senses are real.
	• Planning
	  ○ Brainstorming
	• Action States
	  ○ Performing actions don't necessarily get the expected outcome which is why running actions don't change the agent's memory state
	• Goal State
	  ○ What we hope to accomplish

After getting the filtered list of valid possible goals, proceed to iterate through the list, then the following is ran:

```
// 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
    );
```

More on AStar and Nodes later. The leaf node that is returned contains the first action that is to be run by the agent. The leaf node is the first node in the plan, and its parent node

contains the next action, and all nodes are linked by reference to the parent, and the root is the goal state node. Get the current path from the leaf node which is the subsequent plan of the iterated goal, then perform the following checks:

```
goalState = goalState.Clone();
var leaf = (ReGoapNode<T, W>)astar.Run(
    ReGoapNode<T, W>.Instantiate(this, goalState, null, null, null), goalState, settings.MaxIterations, settings.PlanningEarlyExit, debugPlan : settings.DebugPlan);
if (leaf == null)
{
    currentGoal = null;
    continue;
}

var result = leaf.CalculatePath();
if (currentPlan != null && currentPlan == result)
{
    currentGoal = null;
    break;
}
if (result.Count == 0)
{
    currentGoal = null;
    continue;
}
currentGoal.SetPlan(result);
break;
}
```

If the leaf is null, then a plan couldn't be formulated, so check the next goal of highest priority, otherwise, check if the resulting plan matches the agent's current plan. If the resulting plan matches, then there is no new goal, so proceed to exit and continue the agent's current plan. If the resulting plan has no nodes inside it, then continue to the next goal because this one can't be validated with the provided actions. Passing all these checks means the current iterated goal will be the new goal, and it references its subsequent plan. Perform the call back or the Manager method onDonePlan(currentGoal), thus formally finalizing the work.

Planner Manager and Agent:

For every update, the planner Manager will check its list of finished works to perform the OnDonePlanning() method defined in agent, to proceed to execute the given 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();
    }
}
```

Here's the OnDonePlanning() method inside ReGoapAgent:

```
protected virtual void OnDonePlanning(IReGoapGoal<T, W> newGoal)
{
    startedPlanning = false;
    currentReGoapPlanWorker = default(ReGoapPlanWork<T, W>);
    if (newGoal == null) {
        if (currentGoal == null)
        {
            ReGoapLogger.LogWarning("GoapAgent " + this + " could not find a plan.");
        }
        return;
    }

    if (currentActionState != null)
        currentActionState.Action.Exit(null);
    currentActionState = null;
    currentGoal = newGoal;
    if (startingPlan != null)
    {
        for (int i = 0; i < startingPlan.Count; i++)
        {
            startingPlan[i].Action.PlanExit(i > 0 ? startingPlan[i - 1].Action : null, i + 1 < startingPlan.Count ? startingPlan[i + 1].Action : null, startingPlan[i].Settings, currentGoal.GetGoalState());
        }
    }
    startingPlan = currentGoal.GetPlan().ToList();
    ClearPlanValues();
    for (int i = 0; i < startingPlan.Count; i++)
    {
        startingPlan[i].Action.PlanEnter(i > 0 ? startingPlan[i - 1].Action : null, i + 1 < startingPlan.Count ? startingPlan[i + 1].Action : null, startingPlan[i].Settings, currentGoal.GetGoalState());
    }
    currentGoal.Run(WarnGoalEnd);
    PushAction();
}
```

The goal is checked on whether it's completed or about to finish, and if so, a new goal is calculated, otherwise, an action is pushed:

```
protected virtual void PushAction()
{
    if (interruptOnNextTransition)
    {
        CalculateNewGoal();
        return;
    }
    var plan = currentGoal.GetPlan();
    if (plan.Count == 0)
    {
        if (currentActionState != null)
        {
            currentActionState.Action.Exit(currentActionState.Action);
            currentActionState = null;
        }
        CalculateNewGoal();
    }
    else
    {
        var previous = currentActionState;
        currentActionState = plan.Dequeue();
        IReGoapAction<T, W> next = null;
        if (plan.Count > 0)
            next = plan.Peek().Action;
        if (previous != null)
            previous.Action.Exit(currentActionState.Action);
        currentActionState.Action.Run(previous != null ? previous.Action : null, next, currentActionState.Settings, currentGoal.GetGoalState(), WarnActionEnd, WarnActionFailure);
    }
}
```

Current action run method is called:

```
// 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;
}
```

Done is the WarnAction End method:

```
// 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();
}
```

Fail is the Warn Action Failure method:

```
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);
}
```

By default the run method in ReGoapAction just initializes local variable instances, thus the class that inherits ReGoapAction must override this virtual method to run an action. More on this later. Upon, an action finishing, the PushAction() method is called to run the next action in the plan, and if the action were to fail, then a new goal is calculated.

The call to calculate a new goal isn't always done once the game starts, or an action fails or an action is completed. A new goal is also calculated when the agent is not in any action state, that way the agent is ensured to be always planning, and goals can become un-blacklisted after some delay in time. Goals even compete while the agent executes its plan, individual goals will call to warn the agent to check their goal's priority in comparison to the current goal. This is done within ReGoapGoalAdvanced:

```
public class ReGoapGoalAdvanced<T, W> : ReGoapGoal<T, W>
{
    public float WarnDelay = 2f;
    private float warnCooldown;

    #region UnityFunctions
    protected virtual void Update()
    {
        // if there is a planner and we're currently not planning, and we passed the warnCooldown, then
        //
        if (planner != null && !planner.IsPlanning() && Time.time > warnCooldown)
        {
            warnCooldown = Time.time + WarnDelay;
            // current goal of the planner, can be separate from this goal, hence a different plan
            var currentGoal = planner.GetCurrentGoal();
            // Get the current plan (sequence of node action states) from the planner
            var plannerPlan = currentGoal == null ? null : currentGoal.GetPlan();
            // check if the planner plan is equal to the plan set for the goal
            var equalsPlan = ReferenceEquals(plannerPlan, plan);
            // bool flag to check if this goal is possible (bool WarnPossibleGoal)
            var isGoalPossible = IsGoalPossible();
            // check if this goal is not active but CAN be activated (Author's note)
            //  or                                                   (Author's note)
            // if this goal is active but isn't anymore possible     (Author's note)

            // if the plans are not equal, then the goal is not active, but it can be activated,
            // because it's set as possible
            // or
            // But if the goal is active, the plans match, and the goal is not possible
            if ((!equalsPlan && isGoalPossible) || (equalsPlan && !isGoalPossible))
                planner.GetCurrentAgent().WarnPossibleGoal(this);
        }
    }
    #endregion
}
```

```
public virtual void WarnPossibleGoal(IReGoapGoal<T, W> goal)
{
    if ((currentGoal != null) && (goal.GetPriority() <= currentGoal.GetPriority()))
        return;
    // new passed goal has a higher priority, so the currentActionState must be interrupted, as a new
    // seqence of actions will be made as consequence of the new goal.
    if (currentActionState != null && !currentActionState.Action.IsInterruptable())
    {
        interruptOnNextTransition = true;
        currentActionState.Action.AskForInterruption();
    }
    else
        CalculateNewGoal();
}
```

Goals (ReGoapGoal):

```
public class CollectResourceGoal : ReGoapGoal<string, object>
{
    // set the name of the resource to be collected
    public string ResourceName;

    protected override void Awake()
    {
        base.Awake();
        goal.Set("collectedResource" + ResourceName, true);
    }

    public override string ToString()
    {
        return string.Format("GoapGoal('{0}', '{1}')", Name, ResourceName);
    }
}
```

In the FSM example, there is only a single goal, collect Resource goal, and it instantiates an empty state, and it sets the key-value pair:
<"collectedResource" + ResourceName, true> and the ResourceName in the Builder Prefab is Axe, so the goal state is simply the dictionary with the key string "collectedResourceAxe" with the bool value true. The priority of the goal is 1, and it's set to true to Warn the agent that this goal is possible. Since this is the only goal in the example, it will be the only goal calculated for highest priority, unless blacklisted, or invalidated due to insufficient actions.

Memory (ReGoapMemory & ReGoapMemoryAdvanced):

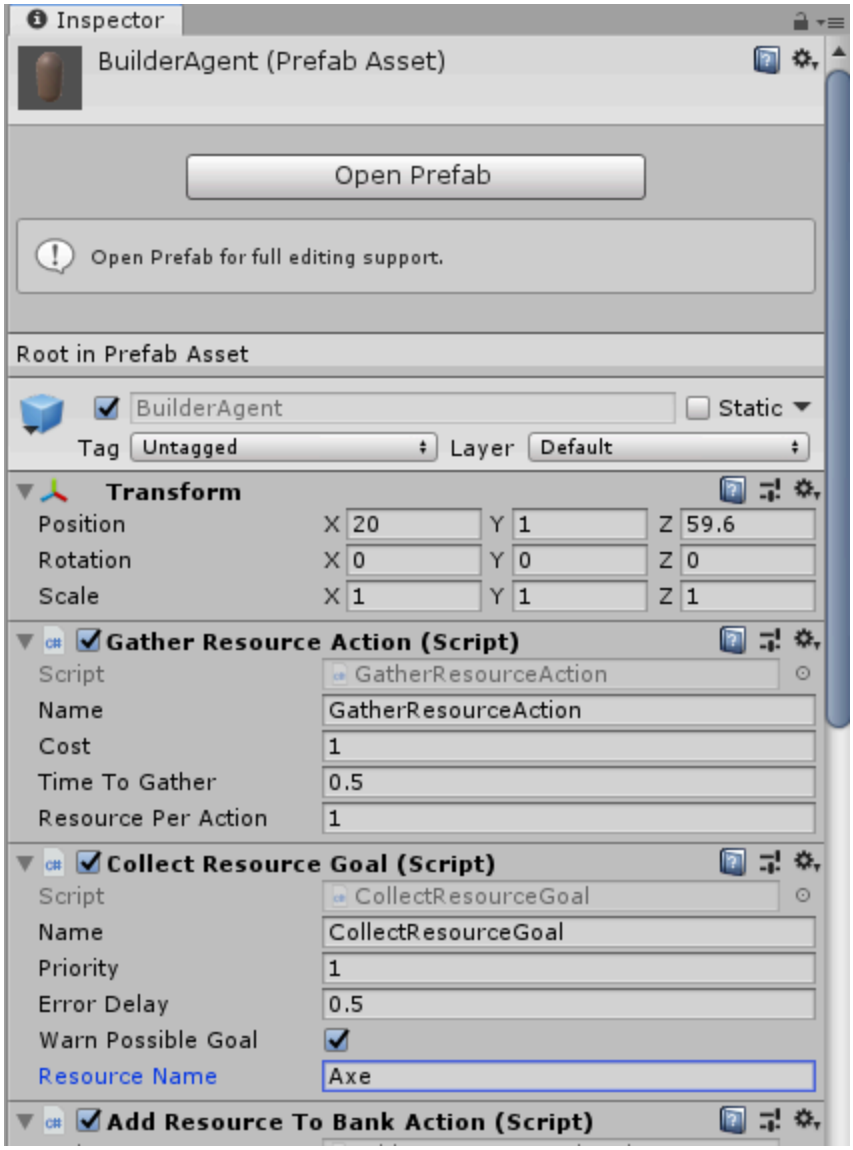
```
// Step 4: Add a ReGoapMemory Componenet, choose a name (you must create your own class that
//         inherit ReGoapMemory or implements IReGoapMemory)
public class BuilderMemory : ReGoapMemoryAdvanced<string, object>
{
}
```

Builder memory is the memory component of the Builder Prefab and memory is just a dictionary (state) of key-value pairs similar to how goals are defined.

```
public class ReGoapMemory<T, W> : MonoBehaviour, IReGoapMemory<T, W>
{
    // This is essentially the memory of an NPC, the world state as the agent knows it!
    protected ReGoapState<T, W> state; // Three dictionaries , Dictionary<T, W> values; bufferA; bufferB

    #region UnityFunctions
    protected virtual void Awake()
    {
        // get an empty dictionary "values" that is cleared and assign to state.
        // We want an empty world state when awaking any ne NPC to the game world!
        state = ReGoapState<T, W>.Instantiate();
    }

    // push the worldstate/state into a cachedStates Stack<ReGoapState<T, W>> which means all
    // dictionaries currently and size of dictionaries.
    protected virtual void OnDestroy()
    {
        state.Recycle();
    }

    protected virtual void Start()
    {
    }
    #endregion

    // Just return ReGoapState<T, W> state, as this will have all the necessary dictionaries to
    // define what the NPC agent knows about the world.
    public virtual ReGoapState<T, W> GetWorldState()
    {
        return state;
    }
}
```

The important part of memory is that it grounds the agent to the game world (reality), and it defines how the agent perceives the world through a dictionary of conditions and values. The ReGoapMemoryAdvanced class has the purpose of updating the memory of how the agent senses the world. There are sensor defined classes added as components to the agent which the memory advanced class will retrieve and through intervals update collected information from the sensors to the memory.

```
public class ReGoapMemoryAdvanced<T, W> : ReGoapMemory<T, W>
{
    // An array of IReGoapSensor<T, W>
    private IReGoapSensor<T, W>[] sensors;

    public float SensorsUpdateDelay = 0.3f;
    private float sensorsUpdateCooldown;

    // Get the different sensors and put them within the
    // IReGoapSensor<T, W>[] sensrs array
    #region UnityFunctions
    protected override void Awake()
    {
        // Call the awkae method from the base class ReGoapMemory,
        // so we instantiate a ReGoapState, or cleared dictionaries of state.values; state.bufferA; state.bufferB
        base.Awake();
        // Get all related IReGoapSensor<T, W> components aded to this GameObject the script is added onto.
        // Meaning the sensors added to the AI character which is the GameObject.
        sensors = GetComponents<IReGoapSensor<T, W>>();
        foreach (var sensor in sensors)
        {
            // Initialize IReGoapSensor with IReGoapMemory<T, W> memory object.
            // void Init(IReGoapMemory<T, W> memory) which memory is a IReGoapState,
            // specifically, the state that was initialized in base.Wake or ReGoapMemory

            // Note: Can use derived classes of the interface...
            sensor.Init(this);
        }
    }

    protected virtual void Update()
    {
        // Update Sensors after every update cooldown.
        if (Time.time > sensorsUpdateCooldown)
        {
            sensorsUpdateCooldown = Time.time + SensorsUpdateDelay;

            foreach (var sensor in sensors)
            {
                sensor.UpdateSensor();
            }
        }
    }
    #endregion
}
```



Sensors (ReGoapSensor):

The BuilderAgent (Prefab) has the following sensors: BankSensor, MultipleResources, ResourceBagSensor, Resource Sensor, WorkStation Sensor.

The ReGoapSensor class is defined as follows:

```
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;
}
// Get the memory of the sensor...
public virtual IReGoapMemory<T, W> GetMemory()
{
    return memory;
}

public virtual void UpdateSensor()
{

}
```

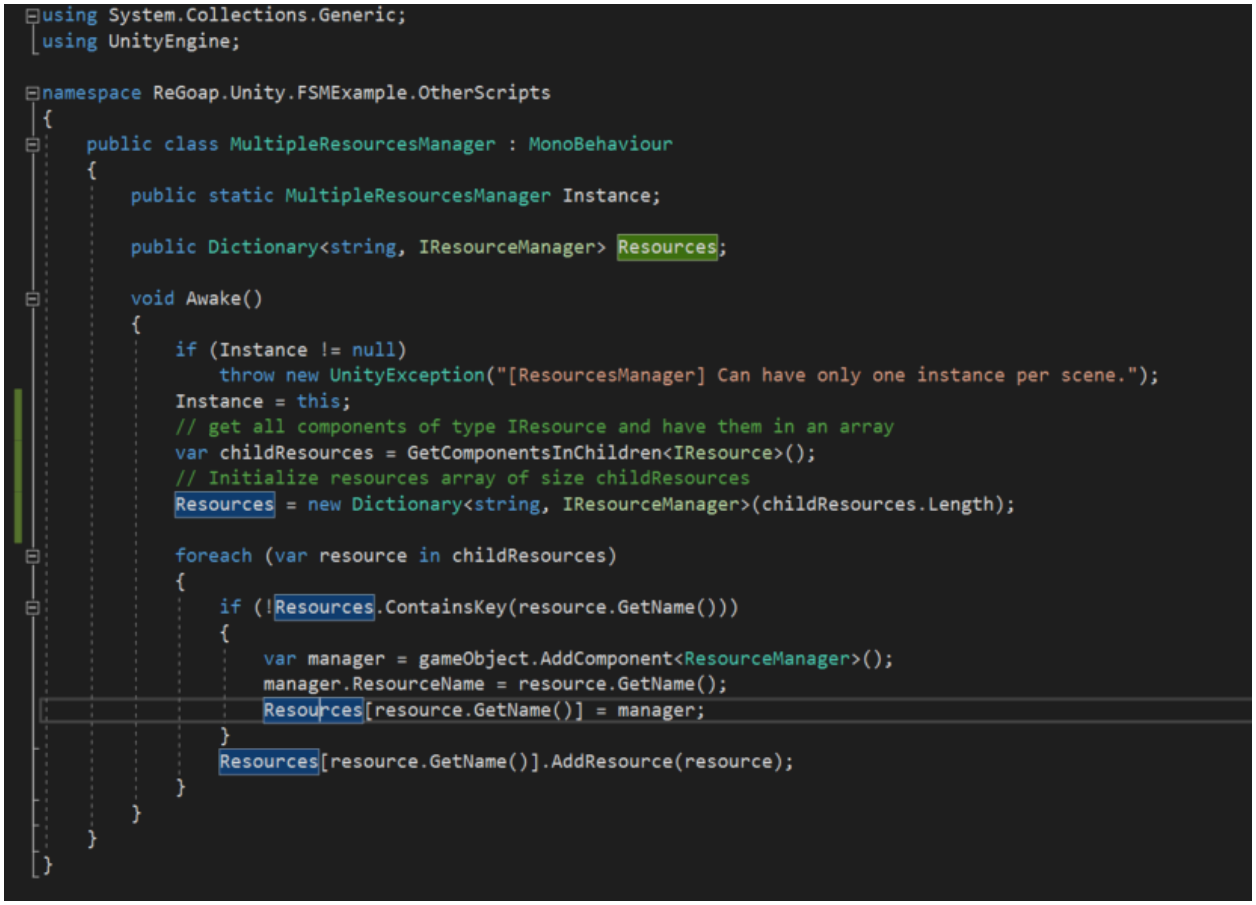
```
namespace ReGoap.Unity.FSMExample.Sensors
{
    public class BankSensor : ReGoapSensor<string, object>
    {
        private Dictionary<Bank, Vector3> banks;    // dictionary of banks and their respective locations

        public float MinPowDistanceToBeNear = 1f;   // minimum distance to be near bank

        void Start()
        {
            banks = new Dictionary<Bank, Vector3>(BankManager.Instance.Banks.Length); // initilize size of banks dictionary
            // get the location of each bank
            foreach (var bank in BankManager.Instance.Banks)
            {
                banks[bank] = bank.transform.position;
            }
        }

        // Update the memory state that the agent can see the bank "seeBank" and where is the nearest Bank to
        // the agent and its position.
        public override void UpdateSensor()
        {
            var worldState = memory.GetWorldState();
            worldState.Set("seeBank", BankManager.Instance != null && BankManager.Instance.Banks.Length > 0);

            // Get the location of the nearest bank from the banks array
            var nearestBank = OtherScripts.Utilities.GetNearest(transform.position, banks);

            worldState.Set("nearestBank", nearestBank);
            worldState.Set("nearestBankPosition",
                (Vector3?) (nearestBank != null ? nearestBank.transform.position : Vector3.zero));
        }
    }
```

Let’s start with the Bank Sensor, it gets a dictionary of type bank with a transform position, and a minimal set distance the agent can be within proximity of a bank. Upon start, the banks dictionary length is set to the BankManager length and every position of each bank in BankManager is set to the banks dictionary. Whenever the sensor updates, we update the three conditions to memory of whether a bank can be seen, what’s the nearest bank, and that nearest bank's Vector3 location.

The BankManager simply is an array of type Bank, and the Bank class is simply a Resource Bag which is an array of type resources, and resource class is simply a string name of the resource with some capacity.

Moving on, Multiple Resources Sensor gets the dictionary of type <string, IResourceManager> from a Multiple Resources Manager class. The Multiple Resources

Manager will initialize a dictionary of Resource Managers, so a resource will have a manager, and the manager will add that resource to its list of resources:



```
using System.Collections.Generic;
using UnityEngine;

namespace ReGoap.Unity.FSMExample.OtherScripts
{
    public class MultipleResourcesManager : MonoBehaviour
    {
        public static MultipleResourcesManager Instance;

        public Dictionary<string, IResourceManager> Resources;

        void Awake()
        {
            if (Instance != null)
                throw new UnityException("[ResourcesManager] Can have only one instance per scene.");
            Instance = this;
            // get all components of type IResource and have them in an array
            var childResources = GetComponentsInChildren<IResource>();
            // Initialize resources array of size childResources
            Resources = new Dictionary<string, IResourceManager>(childResources.Length);

            foreach (var resource in childResources)
            {
                if (!Resources.ContainsKey(resource.GetName()))
                {
                    var manager = gameObject.AddComponent<ResourceManager>();
                    manager.ResourceName = resource.GetName();
                    Resources[resource.GetName()] = manager;
                }
                Resources[resource.GetName()].AddResource(resource);
            }
        }
    }
}
```

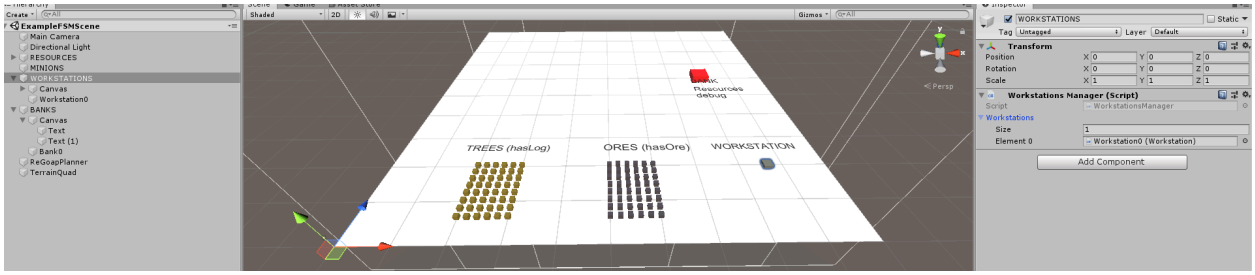
```
namespace ReGoap.Unity.FSMExample.OtherScripts
{ // one resourcemanager per type
    public class ResourceManager : MonoBehaviour, IResourceManager
    {
        private List<IResource> resources;
        private int currentIndex;
        public string ResourceName;

        UnityFunctions

        #region IResourceManager
        public virtual string GetResourceName()
        {
            return ResourceName;
        }

        public virtual int GetResourcesCount()
        {
            return resources.Count;
        }

        public virtual List<IResource> GetResources()
        {
            return resources;
        }

        public virtual IResource GetResource()
        {
            var result = resources[currentIndex];
            currentIndex = currentIndex++ % resources.Count;
            return result;
        }

        public void AddResource(IResource resource)
        {
            resources.Add(resource);
        }
```

```
// the agent in this example is a villager which knows the location of trees, so seeTree is always true if there is an available  tree
namespace ReGoap.Unity.FSMExample.Sensors
{
    public class MultipleResourcesSensor : ResourceSensor
    {
        public float MinResourceValue = 1f;
        public float MinPowDistanceToBeNear = 1f;

        public override void UpdateSensor()
        {
            var worldState = memory.GetWorldState();

            foreach (var pair in MultipleResourcesManager.Instance.Resources)
            {
                var resourceManager = pair.Value;

                worldState.Set("see" + resourceManager.GetResourceName(), resourceManager.GetResourcesCount() >= MinResourceValue);

                UpdateResources(resourceManager);

                var nearestResource = OtherScripts.Utilities.GetNearest(transform.position, resourcesPosition);
                worldState.Set("nearest" + resourceManager.GetResourceName(), nearestResource);
                worldState.Set("nearest" + resourceManager.GetResourceName() + "Position",
                    (Vector3?) (nearestResource != null ? nearestResource.GetTransform().position : Vector3.zero));
            }
        }
    }
}
```

MultipleResourcesSensor inherits the ResourceSensor class which handles updating a dictionary of positions of specific resources, based on a Resource manager passed to it through the MultipleResourcesSensor.

```
namespace ReGoap.Unity.FSMExample.Sensors
{
    public class ResourcesBagSensor : ReGoapSensor<string, object>
    {
        private ResourcesBag resourcesBag;

        void Awake()
        {
            resourcesBag = GetComponent<ResourcesBag>();
        }

        public override void UpdateSensor()
        {
            var state = memory.GetWorldState();
            foreach (var pair in resourcesBag.GetResources())
            {
                state.Set("hasResource" + pair.Key, pair.Value > 0);
            }
        }
    }
}
```

A workstation is a place to craft the resources of log and ore into an Ax. Workstation is defined by a single method, CraftResource, in which it takes a ResourcesBag, a parameter of type IRecipe, and a float value of 1 by default. IThe method gets the dictionary of needed resources, each key a resource with an associated quantity value. Checks if the resource bag has the needed resources and quantity, and if no, then returns false. If resource requirements are met, remove the resource quantities from the bag, and get the crafted resource from the recipe, and add it to the bag. The Workstation Manager is similar to the Bank Manager in which is just holds all workstations in a list.

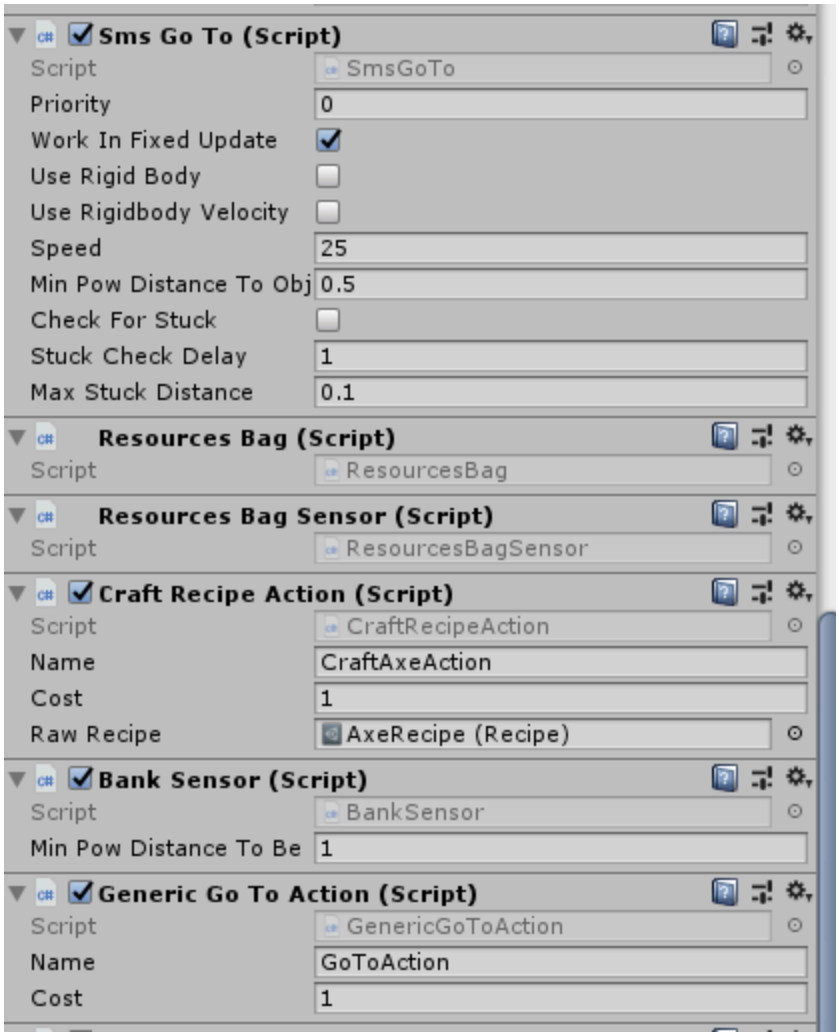


Actions (ReGoapAction):

The Builder Agent collects the trees and ores resources then builds an Ax at a Workstation to then deposit it in the bank. There are four actions: GenericGoToAction, GatherResourceAction, CraftRecipeAction, and AddResourceToBankAction.

The GenericGoto action handles the agent's movement and requires a state of type SmsGoTo. In the Awake() function, it sets its effect to <"isAtPosition", default(Vector3)>

The GatherResourceAction



Finite State Machine FSM:

There are two states defined in the FSM: SmsGoTo and SmsIdle. The SmsIdle state doesn't do anything in particular, but inherits the SmState which inherits ISMState.

The FSM in ReGoap is defined by classes StateMachine, SmState, and ISMState. The State Machine has a dictionary called states, a dictionary of values, a global dictionary of values, and a list of generic state transitions.