Actions

An action is a **Unity Component** of an Agent, enabling it to have behavior and effects on the game. You can think of it as a typical **MonoBehaviour** such as **GoToBehaviour.**

What's special about an **Action** is that it does not do any state planning and all of the work is left to **Agent Goap Planner.** When there is no action in progress, the planner will use the **Agent's states and World States** to find the best possible **set of actions** and the Agent will carry them out.

Making an Action

All SGoap classes are from the SGoap namespace. Create a class that Inherits either Action or BasicAction.

```
public class FooAction : Action{}
public class FooAction : BasicAction{}
```

Plan Execution Order

An action will only run if the plan includes it. A plan can be `GoTo -> Attack`.

It is important to know that before the Action is even considered for the planning sequence, the planner will filter out actions that don't have the AlwaysIncludeInPlan or fails the Action.IsUsable() check.

Note, you can click the arrow to see more details

- ▼ Action.DynamicallyEvaluateCost();

  This is called once for each actions the agent has. It is generally much cheaper to do any dynamic cost calculations in here than the later.

- ▼ Action.IsPreconditionsMet(currentState);

  The preconditions here are what you add in the Action Component in the inspector. If the action fails any precondition, it will not be include in the plan.

- Action.IsProceduralPreconditionMet(currentState);

  This is the method you want to override to add precondition via code based on the current planning state. Nothing you add will effect the world state, it is simply for the planner.

- Action.DynamicallyEvaluateCost(currentState);

  Unlike the first DynamicallyEvaluateCost call, this call contains the current state the planner is at, meaning you can calculate future costs. For example, the current state says you'll be at the `Temple` after executing `X` action.

- Action.ProceduralEffect(currentState);

  Having the current state, you can add dynamic effects and can't normally be done

Action Execution Order

Once your agent has receive the Queue of Actions to run. The following is the cycle of an action.

- Action.PrePerform();
- Action.ShouldAbort();

  In realtime actions games, you most likely want an action to be abortable. What this means is, in the middle of walking to a tree, if you are shot, you want to be able to exit this task and plan again.

- Action.Perform();

  Returning Success will end the action and call PostPerform. Returning Running will continue the perform method until success or failed is returned.

- Action.PostPerform();

  Called only after Action.Success is returned.

- Action.OnFailed();

  Called when an action fails/is aborted and the agent will make new plans.

- Action.CanAbort() && PlannerSettings.CanAbort

Preconditions

A precondition can be added in the inspector on the Action component.

If the preconditions of an action is not satisfied by the Agent's states or World's states which is actually the result of a bunch of after effects (see below), the action will not chain with another.

A **CutTree** action will have a precondition of `HasAxe`.

After Effects

The after effect can be added in the inspector on the Action component. The after effect of one action is what satisfies the precondition of another.

A **BuyAxe** will have the effect of `HasAxe`.

> 💡 When an action completes, successfully or not, no after effects will be applied to any of the Agent's states or the World States. It is only used in the planner.

Action.States

The Action.States points to the states of the Agent. This is a way to modify the agent's state.

For example, when the **BuyAxe** action completes, you can add the `HasAxe` state to the Agent.

```
States.Add("HasAxe", 1);
```

Action.PrePerform()

This method is called before Performing an Action, this is the perfect place for any kind of setup and re-initialization.

```
public override bool PrePerform()
{
  //Setup the action
  return true;
}
```

Action.Perform()

Called when an Agent first run.

Returning Success: will end the action and call Action.PostPerform();

Returning Running: will keep calling Perform until either Success or Failure is returned.

Returning Failure: will not call Action.PostPerform() and instead call Action.OnFailed(). The Agent will also start the plan again from scratch as this plan is no longer possible.

```
public override EActionTask Perform()
{
    return EActionTask.Success;
}
```

Action.PostPerform()

Called after an Agent successfully perform the action. This is the result of return EActionTask.Success on Action.Perform();.

This the perfect place for applying meaningful world state or agent state changes. By default, Action.AfterEffects don't get apply to the Agent's states or the World's states.

```
public override bool PrePerform()
{
  //Setup the action
  return true;
}
```

IsUsable()

Any actions that fails the IsUsable check during the beginning of the plan will not be included. This is for optimization purposes. This can be bypass by the boolean below as you may want an action to still be included while it is cooling down because by the time you reach, it'll be ready.

Example, this action can be use when Hp is > 5;

```
public override bool IsUsable() => Hp > 5;
```

AlwaysIncludeInPlan

Override this method and set to true will bypass the IsUsable check.

```
public override bool AlwaysIncludeInPlan => true;
```

IsProceduralPreconditionMet()

By default, every action has a list of preconditions in the inspector. Sometimes that isn't enough and you want to do more check. For example if you want an action to always pass.

```
public bool AlwaysPass;

public override bool IsProceduralPreconditionMet(Dictionary<string, float> currentState)
{
     if(AlwaysPass)
      return true;

    return false;
}
```

ProceduralEffect()

By default, every action has a list of after effects in the inspector. Sometimes that isn't enough because you want a certain effect to only occur in a chain of actions. Such as, you want this action to give 100hp if you are doing X action.

This allows you to procedurally chain actions.

```
public virtual void ProceduralEffect(Dictionary<string, float> currentState)
{
  if(currentState.Contains("X")
    currentState.Add("Hp", 100);
}
```

DynamicallyEvaluateCost()

An Agent may have multiple set of actions that can satisfy a goal, the cheapest set is the selector.

The cost can be set in the inspector and there is an evaluator script to dynamically do this. However you can override this and do it all in the action if you would like. A dynamic cost is useful for fuizziness.

```
public override DynamicallyEvaluateCost()
{
   Cost = DistanceToPlayer;
}
```

DynamicallyEvaluateCost(currentState)

Similar to ProceduralEffect in the sense that you get access to the currentState and can calculate costs in the future.

```
public override float DynamicallyEvaluateCost(Dictionary<string, float> currentState)
{
   if(currentState.Contains("Sad"))
      Cost = SadCost;
}
```

CanAbort()

When the Planner Settings of an Agent has the CanAbort Flag set to true and can forcefully re-plan every X seconds, this check allows an action to be abortable or not. In general, this is not needed as most agents don't need to re-plan.

OnFailed()

This is called when an Action fails.