
([HTTP://WEB.ARCHIVE.ORG/WEB/20160325140827/HTTP://APEXGAMETOOLS.COM/](http://web.archive.org/web/20160325140827/http://apexgametools.com/)). DOCUMENTATION
([HTTP://WEB.ARCHIVE.ORG/WEB/20160325140827/HTTP://APEXGAMETOOLS.COM/DOCUMENTATION/](http://web.archive.org/web/20160325140827/http://apexgametools.com/documentation/)). APEX UTILITY AI
DOCUMENTATION ([HTTP://WEB.ARCHIVE.ORG/WEB/20160325140827/HTTP://APEXGAMETOOLS.COM/DOCUMENTATION/APEX-UTILITY-](http://web.archive.org/web/20160325140827/http://apexgametools.com/documentation/apex-utility-ai-documentation/)
[AI-DOCUMENTATION/](http://web.archive.org/web/20160325140827/http://apexgametools.com/documentation/apex-utility-ai-documentation/)). APEX UTILITY AI SCRIPTING GUIDE

Apex Utility AI Scripting Guide

What is a Utility AI?

Apex Utility AI is an advanced hierarchical scoring-based artificial intelligence (AI) framework for computer games. The AI is capable of advanced abstract reasoning using simple scoring functions to evaluate complex situations.

Quick Guide

To create a new Utility AI you do the following:

1. There are two ways for a create an AI:
 1. Tools => Apex => AI Editor
 2. Navigate to the Project View, right click => Create => Apex => Utility AI
2. Press “New”, enter a name and press OK to the save the AI with the new name.
3. Right-click within the editor window, and select one of the selectors, such as “First Score Wins” to create a new selector
4. Right click on the “First Score Wins” Selector, and select a Qualifier, such as “Sum of Children” to create a new qualifier
5. Right-click on the newly created Qualifier and select an Action, such as “Wander” to create a new action
6. Press “Save”. The AI is now saved.

Tip

You can set the AI to auto saving in the AI Settings. Tools => Apex => AI Settings.

Note

The AI automatically saves when Unity recompiles.

How does the Utility AI work?

The Utility AI works by selecting one option from a range of options based on the usefulness or “utility” of each option. The Utility AI then executes the selected option. This allows the Utility AI to select between options that can be very different, in a manner that is simple to implement.

The following table outlines the major concepts used in the Utility AI

Concept	Explanation
Selector	Selects the best Qualifier from the Qualifiers attached to the Selector
Qualifier	Calculates a score that represents the utility/usefulness of its associated action.

Scorer	A method for calculating scores that can be reused across Qualifiers.
Action	The action that the AI executes when a specific Qualifier is selected.
Context	The information available to the AI when calculating the scores.

Note

There are two types of Scorers. ContextualScorers are used to score Qualifiers. OptionScorers, are used to score different options available to Actions, such as scoring different potential destinations for the AI.

In the Apex Utility AI, Qualifiers are used to evaluate where any given action by the AI should be executed. The Qualifiers are grouped within a Selector. The Qualifiers each return a score to the Selector that the Selector uses to rank the Qualifiers and select the best. The Qualifiers can either contain a method for calculating and returning a score, or they can have a list of Scorers that each calculate and return a score.

Selectors typically select the highest score, but it can use other decision-algorithms, such as selecting the first Qualifier that scores above zero. Each Qualifier implements an Action or it invokes another Selector.

Using the Apex Utility AI in a Unity Project

AIs can either be added to a GameObject as a component using the UtilityAIComponent or a UtilityAIClient can be instantiated directly in code.

Usage:

```
var utilityAIClient = new LoadBalancedUtilityAIClient(AINameMap.PlayerBuildBots, player);
```

Where AINameMap is a list of the AIs currently implemented (see AINameMap) and player is an IContextProvider (see IContextProvider) that creates a context and makes this accessible to the AI in every AI update.

Note

The LoadBalancedUtilityAIClient automatically adds the AI to the Apex AI LoadBalancer.

Note

UtilityAIClient is an abstract class, so you cannot instantiate it directly. You can create your own implementation of the UtilityAIClient class by inheriting from it.

How does the Utility AI Make Decisions?

The Utility AI's ability to think is based on creating Qualifiers and Scorers and adjusting their scores to make the AI make reasonable choices.

Example

Problem: Should I work or have lunch?

My utility for work: Lines of code I need to debug

My utility for lunch: Hours since last meal

In the situation where I need to debug two (2) lines of code and its three (3) hours since my last meal, I will opt for lunch since 3 (lunch) > 2 (work).

The Utility AI consists of an arbitrary tree of Selectors. Each Selector consists of a list of Qualifiers. Each Qualifier leads to an Action, such as moving or attacking. The Qualifier is chosen by the Selector based on the score each Qualifier returns. The Selectors can use several ways of choosing the qualifier, such as the highest scoring Qualifier or the first scoring Qualifier.

The example below shows how decisions can be chained together in a tree structure. In this example, we need to choose what to have for lunch, after we have decided to have lunch.

Example

Now that I have decided I will have lunch, I need to determine what to eat.

Problem: What should I eat?

My utility for hamburger: Day of the month, since I have less money the further I am away from payday

My utility for steak: Minutes left of my lunch break

The day of the month and minutes of my lunch have nothing to do with each other. Comparing them is like comparing apples and oranges. However, using the Utility AI we can actually compare them anyway. This ability to compare abstract concepts is one of the core strength of the Utility AI.

In the situation where I have 20 minutes left of my lunch break, and it is the 9th in the month, I will have steak since $20 > 9$. Hence, even though minutes and day of the month are two totally unrelated concepts, we can still decide to compare them, and select whichever number is higher and implement the choice associated with the highest number. The ability to compare unrelated abstract concepts is similar to human reasoning, and our ability to make up our own rules for comparing, calculating utility, and make a rational choice, gives us an exceptional ability to take decisions.

As the Qualifiers and Scorers can act on any data, very advanced AI can be created.

The example below shows how the Apex Utility AI can be used for evaluating any type of decision-making, even complex investment decisions. The sample principle can be applied to other complex decision-making scenarios such as target selection and tactical positioning during combat.

Example of an advanced decision-making AI

Problem: What stocks should I invest in?

Potential Scorers:

What is the price-earning of the stock?

What is the current rate of revenue growth?

What is the current growth in customers?

What is the company's market share?

What is the strength of competitors?

Each Scorer can assign a value of 0 to 100. The highest scoring stock is then a candidate for investment.

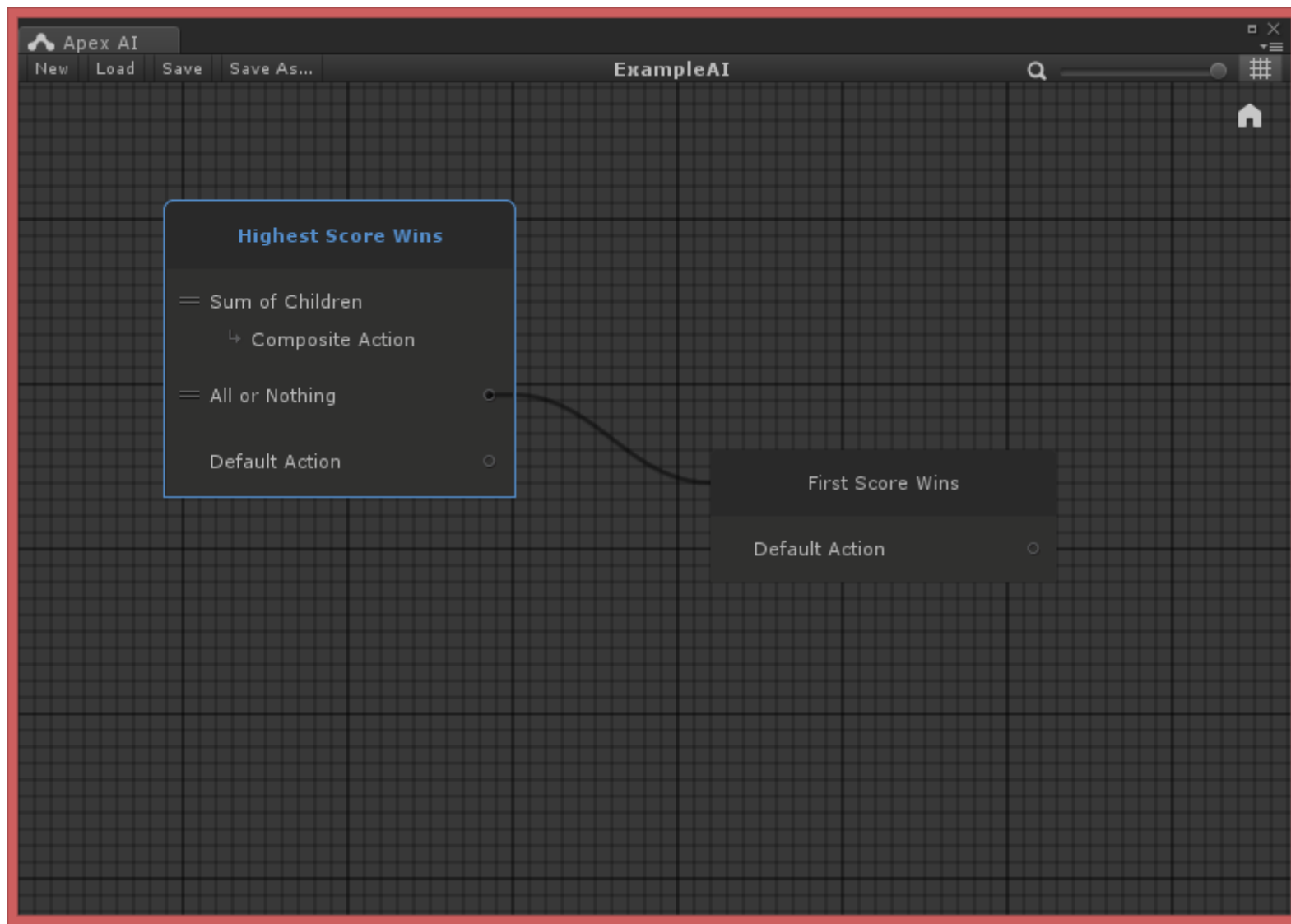
The score returned by Qualifiers and Scorers don't have to be fixed numbers. They can be calculated via any mathematical function, such as exponential or logistic functions. This is covered in separate tutorials.

The Apex Utility AI Editor

The Apex Utility AI is integrated in the Unity editor for quick design and configuration of AIs.

The figure below shows the basic structure of the Apex Utility AI.

The Apex Utility AI consists of Selectors, which are the large boxes. The Selectors have a list of Qualifiers, which are the left indented lines inside the Selector. Qualifiers have one of the following attached to them: An Action, a Composite Action, or a link to another Selector or SubAI. In this way, multiple Selectors and AIs can be chained together.

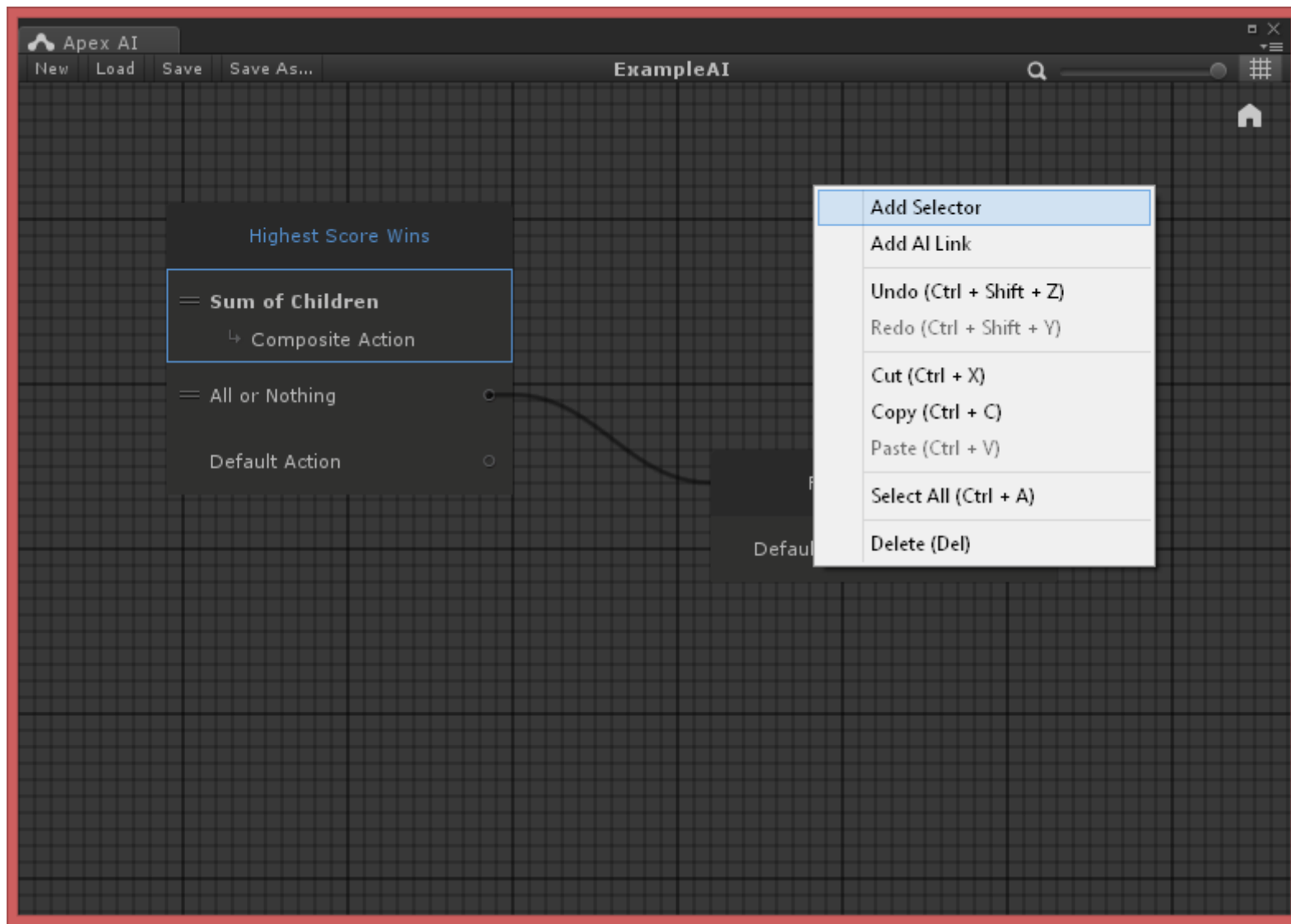


The “Highest Score Wins” Selector is the default or Root Selector. This can be seen by the blue text of the name of the Selector. It has three Qualifiers; “Sum of Children”, “All or Nothing”, and “Default Action”. The “Sum of Children” Qualifier executes a Composite Action. The Composite Action can consist of multiple Actions, which are executed sequentially. The “All or Nothing” Qualifier is linked to the “First Score Wins” Selector. Hence, if this Qualifier is selected, it will execute the “First Score Wins” Selector. Note that each Selector has a Default Action that will be executed if no Qualifier is executed. Per default, the Default Action will do nothing. However, a custom Action can be added to the Default Action Qualifier.

Tip

You can change the names of all Selectors, Qualifiers and Scorers in the Utility AI Editor window. This is done through the “Name” property in the Inspector.

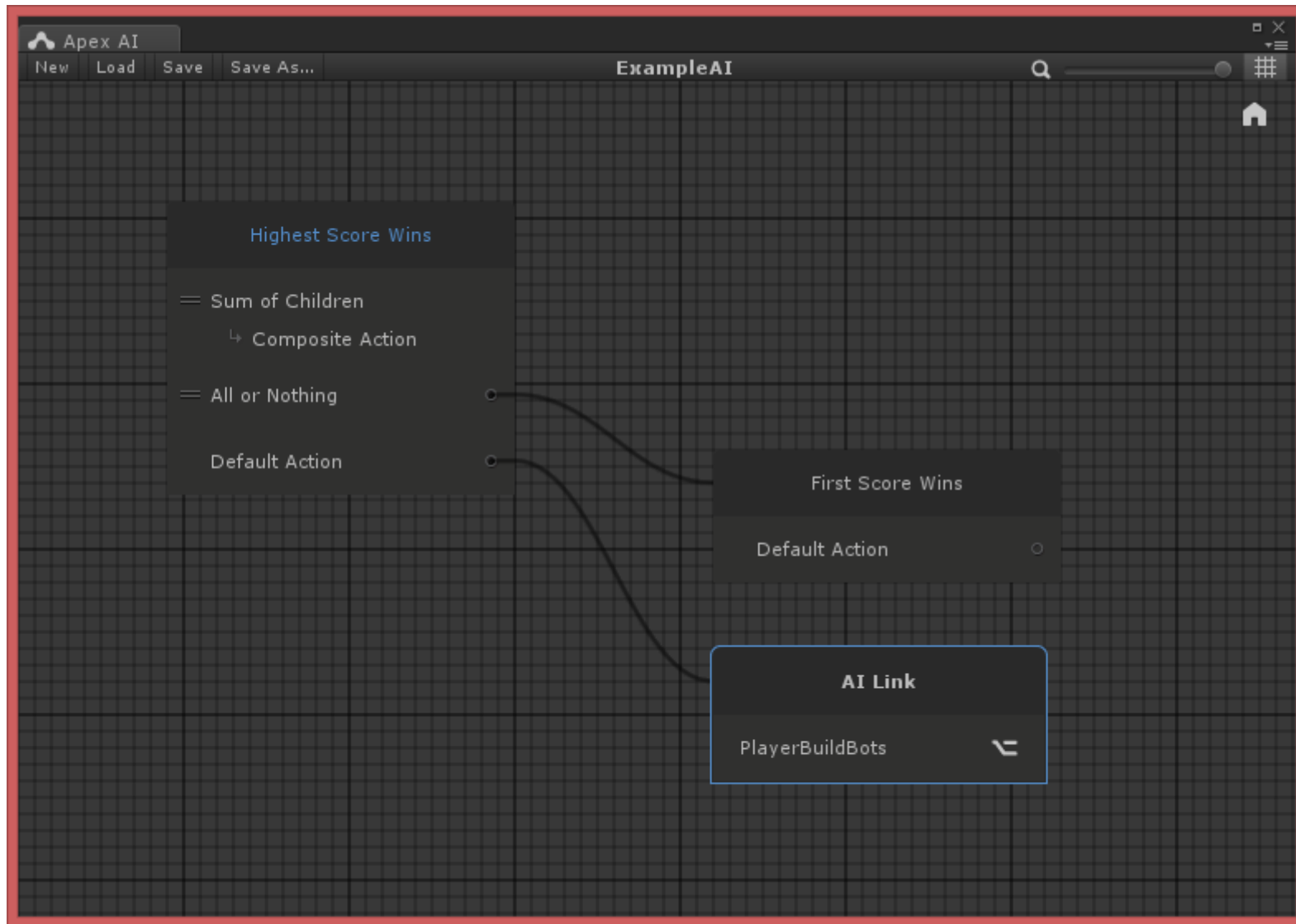
New Selectors can be added by right-clicking an empty space inside the Utility AI Editor window. The figure below shows how a new Selector can be added to the Utility AI.



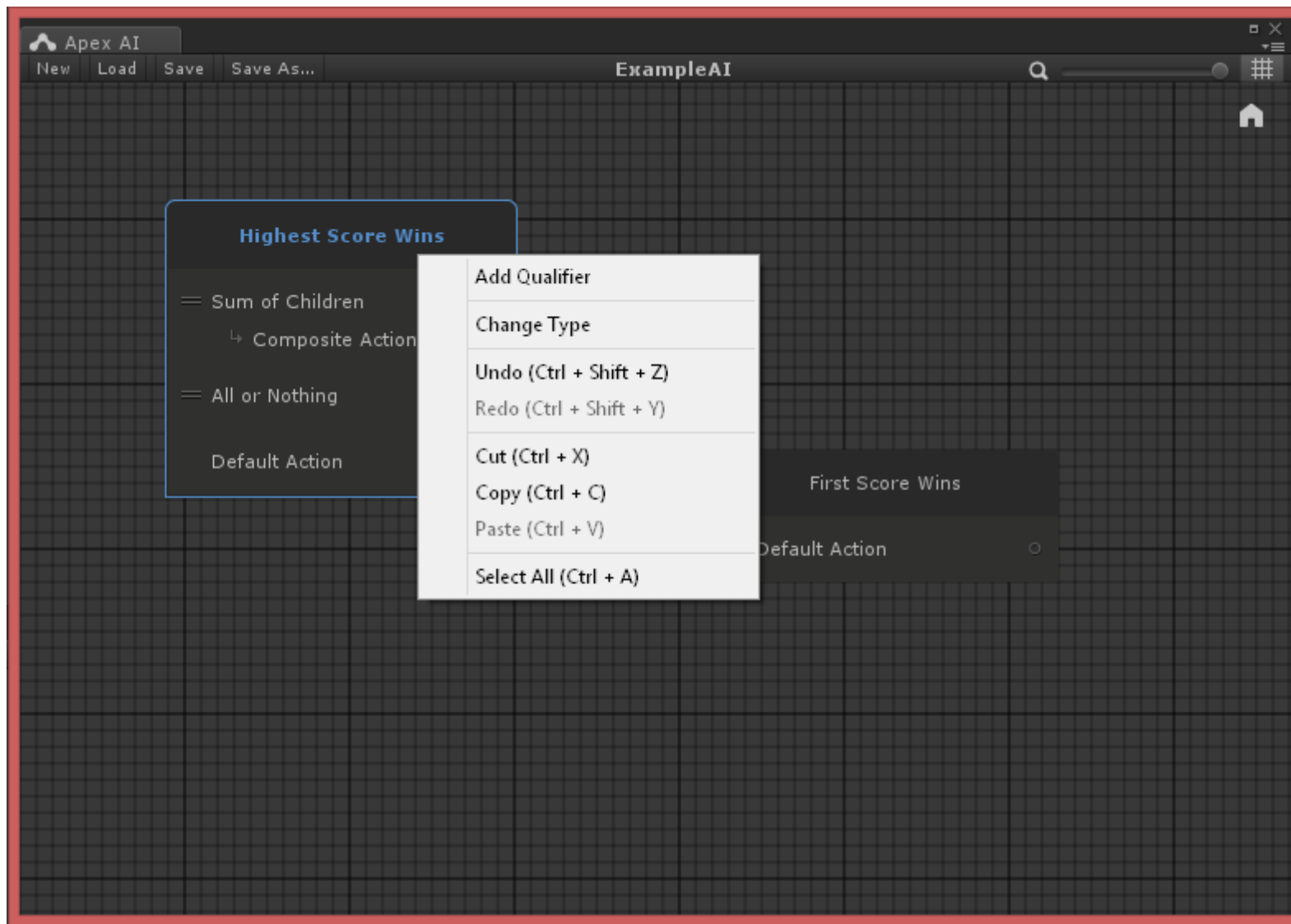
Selectors are connected by holding down the left-mouse button over a Qualifier that does not have an action associated and dragging the mouse to the Selector that should be connected. A black line indicating a link appears and links itself to the Selector it is dragged to. Links are deleted by left-clicking the small round icon in the Qualifier where the link starts, and then holding down the mouse button and dragging the link to an empty space in the Utility AI Editor.

Note that by right-clicking an empty space inside the Utility AI Editor a link to another AI can also be added to the Utility AI Editor window.

The figure below shows how another AI is linked to the Selector via the Default Action.



Right-clicking on the title of the Selector brings up the Selector Context Menu.

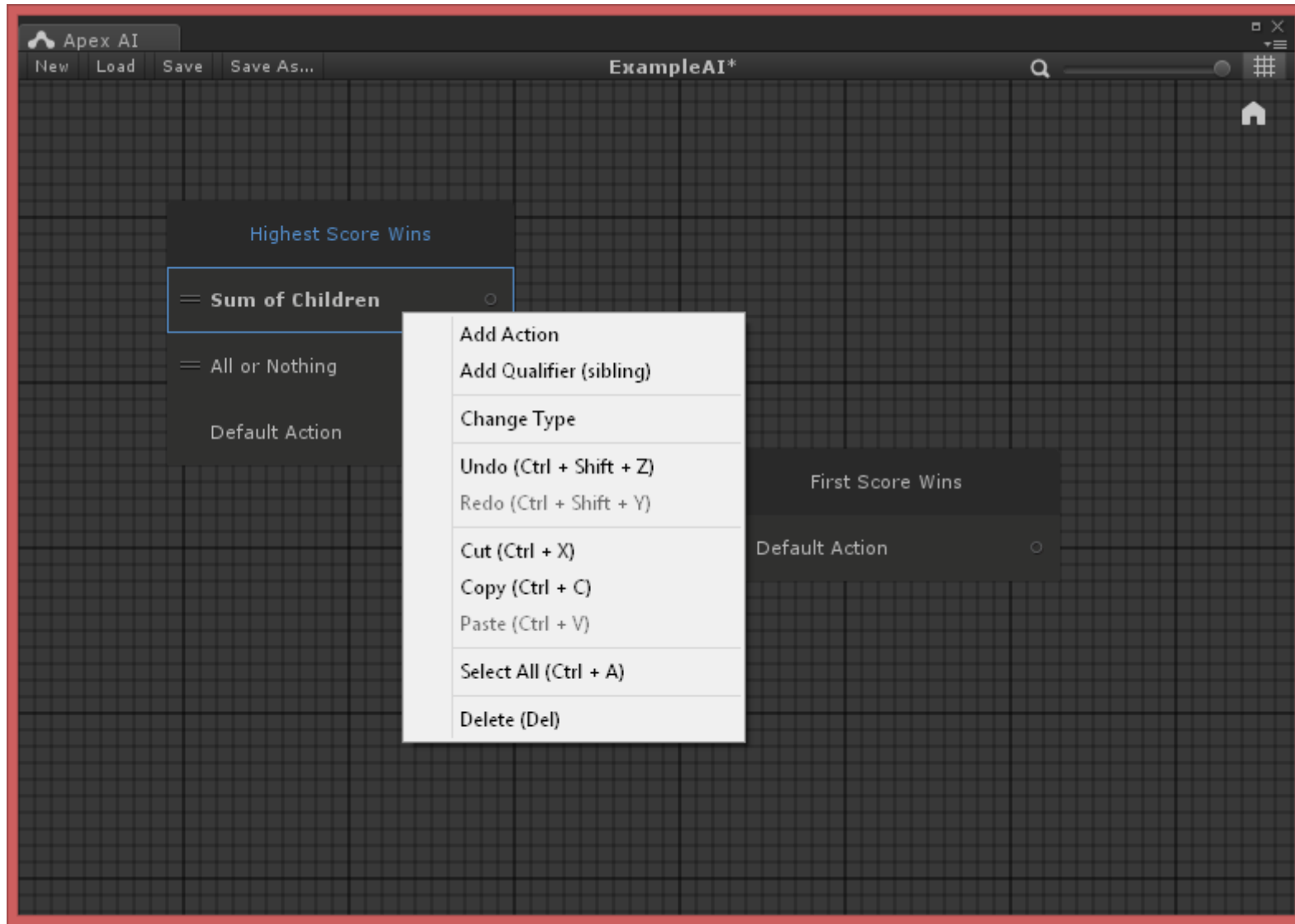


The “Add Qualifier” menu item adds a new Qualifier to the Selector. The order of the Qualifiers in the Selector can subsequently be changed by left-clicking on the two horizontal lines to the left of the title of the qualifier, holding down the mouse-button and dragging the Qualifiers to the desired position in the list of Qualifiers. The order is especially relevant for “First Score Wins” Qualifiers.

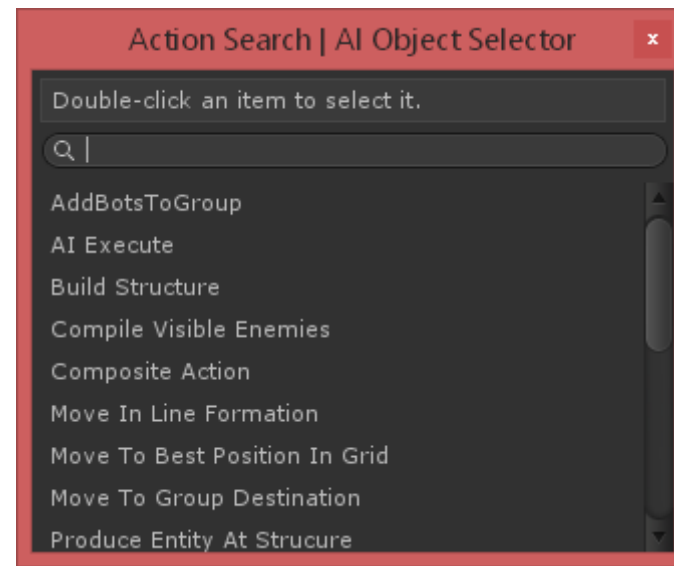
The “Change Type” menu item allows the type of the Selector to be changed. (See the Selector section for types of Selectors and their usage.)

Right-clicking inside a Qualifier brings up the Qualifier Context Menu.

The figure below shows how Qualifiers can be added or their type changed, and how Actions can be added or replaced.

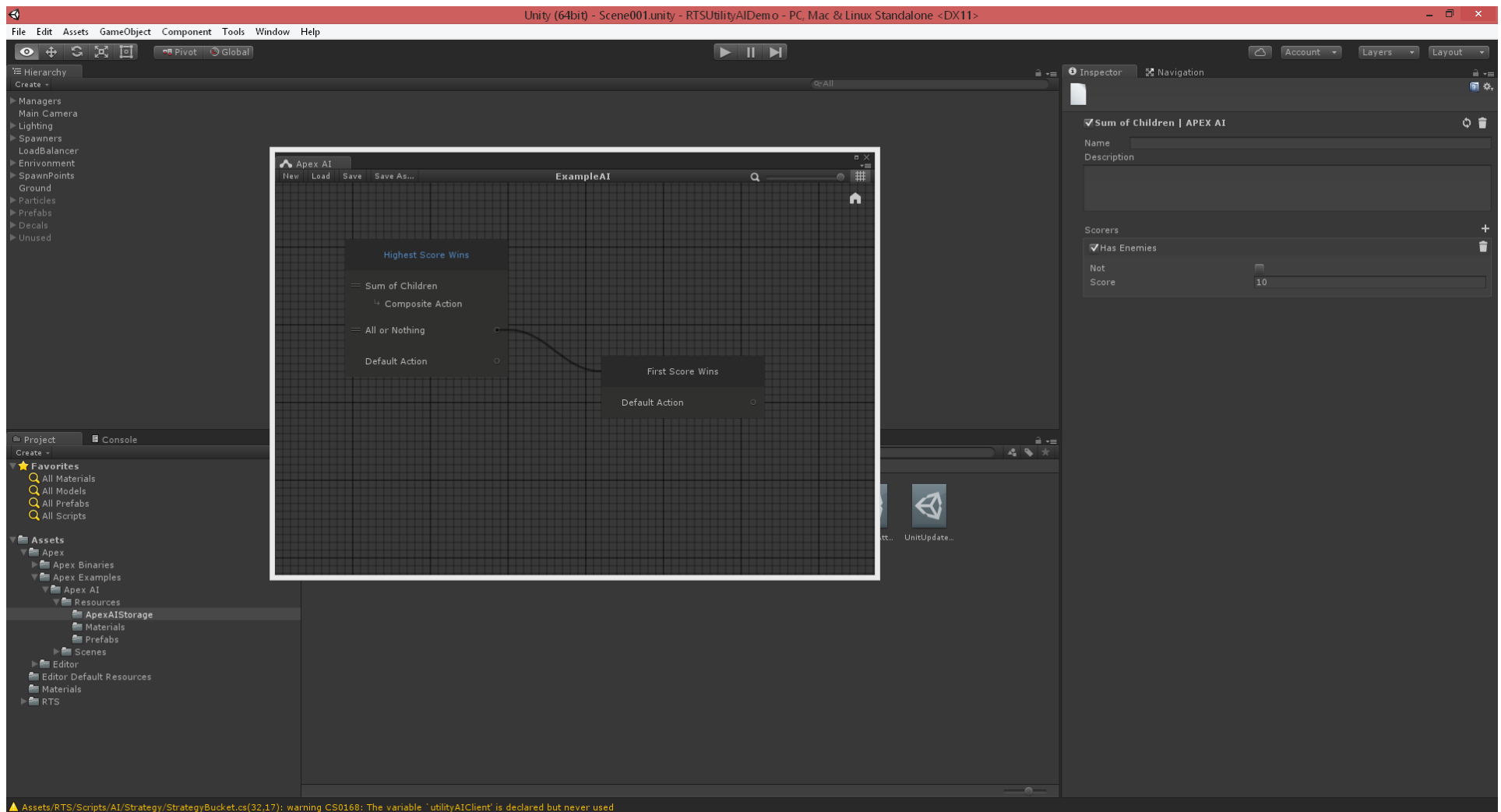


The “Add Action” menu item opens the Action selector and allows adding an Action from the list of available action objects. If there is already an Action attached to the Qualifier, the menu Item “Replace Action” will appear instead.



The Qualifiers and Scorers can be configured in the Inspector.

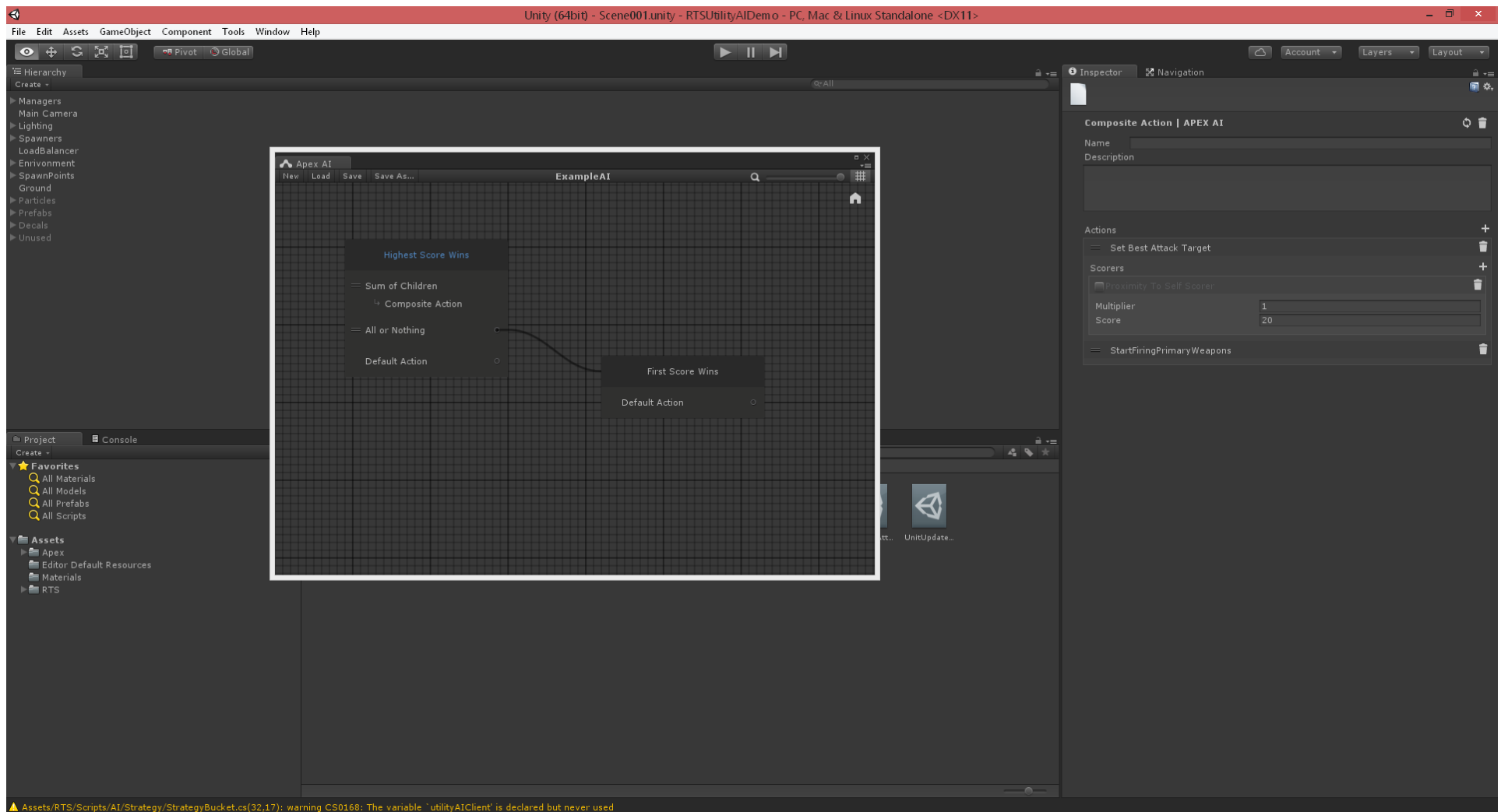
The figure below shows how Scorers can be added to the “Sum of Children” Qualifier.



The “Name” field in the inspector changes the name that the Qualifier shows in the Utility AI Editor window.

The “Description” field allows notes or description of the usage of the Qualifier to be added.

Clicking the “+” sign allows adding Scorers to the Qualifiers. Parameters on the Scorers can subsequently be set under each Scorer.



The Apex Utility AI Editor Context Menu

The Apex Utility AI Editor context menu provides a number of useful functions for the Utility AI.

“Undo” and “Redo” allows full undoing and redo of actions inside the editor.

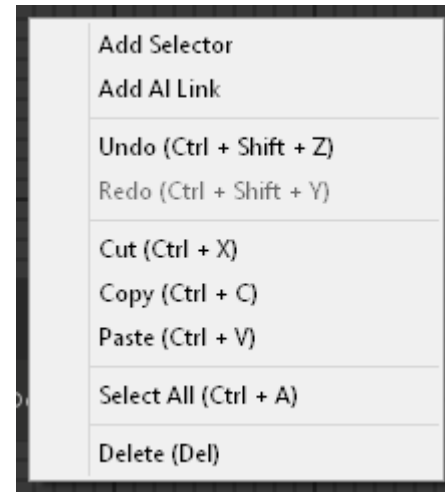
“Cut” allows moving and (through Paste) inserting Selectors, Qualifiers and Actions other places in the AI.

“Copy” allows copying Selectors, Qualifiers and Actions, including their parameters to other places in the AI.

“Paste” allows pasting of cut or copied Selectors, Qualifiers and Actions to other places in the AI.

“Select All” selects all Selectors, Qualifiers and Actions in the Utility AI Editor window.

“Delete” provides a way to remove Selectors, Qualifiers, Actions and Link AIs from the Editor window.



Tip

You can cut, copy and paste across Utility AI Editor windows. This means that you can copy part or all of an AI into other AIs.

Apex Utility AI Core Classes

In the following sections, we explain the core classes of the Apex Utility AI

Context

The Context is the central aspect of the Utility AI. The Context is where the AI stores all information about itself and its surroundings.

The Context is implemented via the `IAIContext` interface. However, the actual implementation is game specific and can be implemented in any way needed by the actual game. The Context can contain information about observations, events, weapons, locations and other information collected or needed by the AI.

The Context object is passed around the Utility AI as an `IAIContext` interface, and must be cast to the concrete type.

```
1 public class ExampleContext : IAIContext
2 {
3     // Put properties/fields/methods here
4 }
```

When a method has to use the context, the Context is cast to the concrete type in the following way:

```
1 public override float Score(IAIContext context)
2 {
3     // Cast the provided context to your concrete context type
4     var concreteContext = (ExampleContext)context;
5
6     // Put logic here
7
8     return this.score;
9 }
```

Note

If you would like to avoid having to cast the context object to your concrete type, you can use the generic base class version of scorers and actions, e.g. deriving from `ContextualScorerBase<ExampleContext>` will do the casting of the context object to the specified type for you, as will `ActionBase<ExampleContext>`.

Selectors

Selectors select the best Qualifier from the Qualifiers attached to the Selector. The following types of selectors are available out-of-the-box in the Apex Utility AI:

Selector Type	Explanation
Highest scoring Qualifier	Selects the Qualifier that returns the highest score.
First scoring Qualifier	Selects the first Qualifier that returns a score higher than the score of the Default Qualifier.

Additional Selectors can be created by inheriting from the *Selector* class and overriding the Select method.

```

1 public sealed class ExampleSelector : Selector
2 {
3     public ExampleSelector()
4         : base()
5     {
6     }
7
8     public override IQualifier Select(IAIContext context, IList<IQualifier> qualifiers, IDefaultQualifier defaultQualifier)
9     {
10         // Put logic here
11
12         return defaultQualifier;
13     }
14 }

```

Qualifiers

Qualifiers return a score to their Selector. The score can be a constant or can be calculated. Qualifiers can also have a list of Scorers attached.

Qualifier Type	Explanation
All or Nothing	Returns the sum of all Scorers if all the scores are above the threshold
Fixed	Returns a fixed score
Sum of Children	Returns the sum of all Scorers
Sum while above threshold	Scores by summing Scorers, until a Scorer scores below the threshold

- **All or Nothing** is useful for scenarios where all criteria must be true or above a certain threshold to execute the attached action.
- **Fixed** is useful for setting a Qualifier that should be executed if no other Scorers reached a certain threshold.
- **Sum of children** is useful for scenarios where all Scorers should be evaluated side-by-side.
- **Sum while above threshold** is useful for scenarios where only Scorers of a certain score should count, to avoid e.g. jitter or crowding out from many small scores.

Custom Qualifiers can be created by inheriting from the QualifierBase class and overriding the Score method.

```
1 public sealed class ExampleQualifier : QualifierBase
2 {
3     [AISerialization]
4     public float score;
5
6     public override float Score(IAIContext context)
7     {
8         // Put logic here
9
10        return this.score;
11    }
12 }
```

Note

Instead of inheriting from QualifierBase, the class can implement the IQualifier interface or inherit from the generic base class QualifierBase<T>, where T is the type of the Context. Implementing the interface allows custom implementations of methods and properties. Inheriting from the generic base class allows the type of the context to be specified, so casting to the specific type becomes unnecessary.

Tip

Consider using *ContextualScorers* rather than custom qualifiers, as they allow for more flexibility and reuse across Qualifiers.

Actions

Actions are what the AI actually executes, such as moving, attacking etc. Actions can also be abstract such as updating the Context of the AI. Actions are represented via the `ActionBase` class.

Action Type	Explanation
Action	Action that is executed when its Qualifier is selected
ActionWithOptions<T>	Action that evaluates several options before it executes the action (See separate paragraph on this)
Composite Action	Composite Action wraps several Actions. The Actions are executed in the order they are set in the Utility AI Editor
Link Action	Link Action is a system internal action used for linking a Qualifier with a Selector or AI.

Actions can be created by inheriting from the `ActionBase` class.

```
1 public sealed class ExampleAction : ActionBase
2 {
3     public override void Execute(IAIContext context)
4     {
5         // Cast the provided context to your concrete context type
6         var concreteContext = (ExampleContext)context;
7
8         // Put logic here
9     }
10 }
```

Note

Instead of inheriting from `ActionBase` the class can implement the `IAction` interface or inherit from the generic base class `ActionBase<T>`, where `T` is the type of the Context. Implementing the interface allows custom implementations of methods and properties. Inheriting from the generic base class allows the type of the context to be specified, so casting to the specific type becomes unnecessary.

ActionWithOptions<T>

ActionWithOptions is a special type of action that evaluates several possible options for the same action. This can for example be evaluating several destinations for a move action or for scoring all targets for an attack action.

```
1 public sealed class ExampleMoveAction : ActionWithOptions<Vector3>
2 {
3     public override void Execute(IAIContext context)
4     {
5         var c = (ExampleContext)context;
6
7         var best = this.GetBest(c, c.sampledPositions);
8         // Move to the best position..
9     }
10 }
```

Tip

When using ActionWithOptions it can be useful to use a “tiebreaker” Scorer to avoid oscillation between equally scoring options. Take an example where the AI wants to move to a position that is both as close to a wall as possible and as close to one of its allies as possible. If several allies are standing at the same distance to the wall, many positions would score the same, and the AI could keep oscillating between the similarly scoring positions. To find only one position, a third scorer – the tiebreaker – could be a Scorer that scores positions on the basis of how close they are to the current position of the AI. In that way, the closest position would score higher than the other positions. See the Survival Shooter tutorial “ProximityToSelf” Scorer as an example of such a tiebreaker.

Scorers

Scorers are classes that calculate scores for Qualifiers and for the options in ActionsWithOptions.

There are two types of Scorers; Contextual Scorers and Option Scorers

Scorers Type	Explanation
ContextualScorers	Are attached to Qualifiers and use the Context as input for scoring.
OptionScorers	Are attached to ActionWithOptions<T> and use the Context and the Option type (T) as inputs for scoring.

Contextual Scorers are the Scorers that are attached to Qualifiers. Their intended usage is to calculate a score based on data from the Context.

Contextual Scorers can be created by inheriting from the ContextualScorerBase.

```
1 public sealed class ExampleContextualScorer : ContextualScorerBase
2 {
3     public override float Score(IAIContext context)
4     {
5         // Cast the provided context to your concrete context type
6         var concreteContext = (ExampleContext)context;
7
8         // Put scoring logic here
9
10    return this.score;
11    }
12 }
```

Context Provider

The Context Provider is responsible for instantiating, maintaining and providing the Context to the AI Client. Every time the Utility AI executes, it calls the Context Provider to get the Context. This allows you to implement additional logic upon context instantiation and maintenance as needed. The simplest implementation is that the Context acts as a singleton (in the scope of the client).

Context Providers can be created by implementing the IContextProvider.

```
1 public class ExampleContextProvider : IContextProvider
2 {
3     private readonly IAIContext _ctx = new ExampleContext();
4
5     public IAIContext GetContext()
6     {
7         return _ctx;
8     }
9 }
```


Enabling the Editor to Find Your Custom Classes

By default all classes in your Unity project (Assembly-CSharp) will be identified by the Utility AI Editor. If you need to identify classes in compiled assemblies, you need to decorate the assembly with the *AIRelevantAssemblyAttribute*.

This can be relevant if some of the classes you use for the Utility AI are not located in the Asset folder or one of its subfolders in your Unity project. In that case, you need to add the following line to any .cs file in the folder where your classes are.

```
[assembly: Apex.AI.AIRelevantAssemblyAttribute]
```

You can create an empty .cs file for this purpose, containing nothing but the above line. Then Apex Utility AI automatically picks up the classes.

AI Name Map

To allow easy access to the Utility AIs from code, each AI is issued a Guid, which is a unique id. To help you identify the AI, the Apex Utility AI contains an AI Name Map that allows you to access the Guid via a static helper class.

Usage:

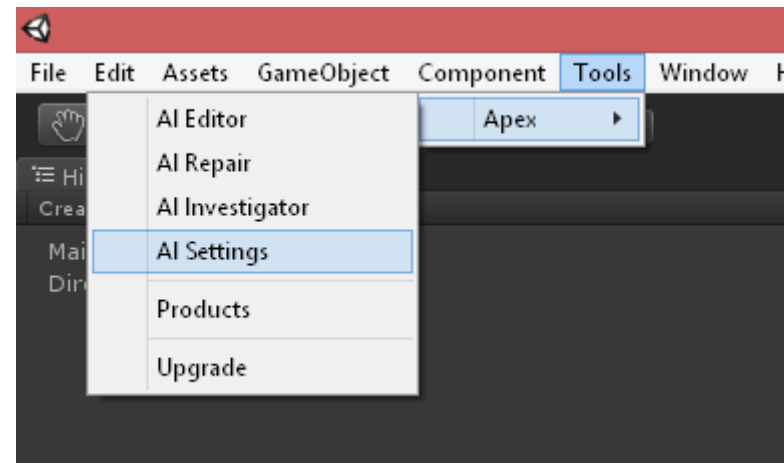
```
AINameMap.ExampleAI;
```

Where “ExampleAI” is the name of a specific AI.

If you change the name of the AI, you need to regenerate the AI Name Map. This is done by going to Tools > Apex > AI Settings > Press the “Generate Name Map” button;

Here you will also find an option to auto-generate the AI Name Map when new AIs are created or AIs are deleted.

The click the “Generate Name Map” button, click “Ok”, and the AINameMap is generated.



Apex AI Settings



User Settings

Title UI Height	<input type="text" value="40"/>
Qualifier UI Height	<input type="text" value="40"/>
Action UI Height	<input type="text" value="30"/>
Scorer UI Height	<input type="text" value="20"/>
Canvas Pan Sensitivity	<input type="text" value="1"/>
Snap Spacing	<input type="text" value="10"/>
Key Pan Speed	<input type="text" value="20"/>
Zoom Speed	<input type="text" value="0.01"/>
Zoom Scale	<input type="text" value="0.2"/> to <input type="text" value="1"/>
Auto Save Interval	<input type="text" value="5"/>
Max Undo	<input type="text" value="50"/>
Show Tooltips	<input checked="" type="checkbox"/>
Show Title in Tab	<input type="checkbox"/>
Ping Asset	<input checked="" type="checkbox"/>
Confirm Deletes	<input checked="" type="checkbox"/>
Prompt to Save	<input checked="" type="checkbox"/>

General Settings

To change the storage location you must select a Resources folder in the project.

Storage Location

The Name Map is a generated class used to reference AIs by name.

☒ Auto Generate

Name Map Location

Utility AI Client

The Utility AI Client is a client that executes the Utility AI, by default through the Apex Load Balancer. A unit will have one Utility AI Client per Utility AI.

Usage:

```
var client = new LoadBalancedUtilityAIClient(AINameMap.ExampleAI, new ExampleContextProvider());
```

You need to set the `executionIntervalMin` and `executionIntervalMax` properties on the client to manage the update interval, unless the AI should execute every frame.

```
client.executionIntervalMin = 1f;
```

```
client.executionIntervalMax = 1f;
```

Finally, you need to start the AI.

```
client.Start();
```

There is also a `MonoBehavior` component that instantiates and starts the Utility AI Client, called `UtilityAIComponent`.

Exposing Fields to be Editable in the Editor

To make fields and properties editable in the editor, they require the attribute *AISerialization*.

```
[AISerialization]  
public float score;
```

In addition to the serialization attribute you can also apply the *FriendlyName* and *SettingCategory* attributes to fields and properties. This will control how they appear in the editor. There is also an attribute called *MemberDependency*, which can optionally show/hide fields or properties based on the values of other fields or properties. The below example shows the usage of *FriendlyName* specifying both a name and a description, additionally the two shown fields are mutually exclusive, facilitated by the *MemberDependency* attribute.

```
[AISerialization, FriendlyName("Use Attack Range", "Whether to use the entity's scanning range as the maximum allowed range for  
enemies to factor into the count"), MemberDependency("useScanRange", false)]  
public bool useAttackRange = false;
```

```
[AISerialization, FriendlyName("Use Scan Range", "Whether to use the entity's attack range as the maximum allowed range for enemies  
to factor in to the count"), MemberDependency("useAttackRange", false)]  
public bool useScanRange = true;
```

AIStorage

The AI Storage folder is the folder where the Utility AI assets are stored. This folder must be stored in a subfolder to a "Resources" folder in Unity.

Apex AI Settings



User Settings

Title UI Height	<input type="text" value="40"/>
Qualifier UI Height	<input type="text" value="40"/>
Action UI Height	<input type="text" value="30"/>
Scorer UI Height	<input type="text" value="20"/>
Canvas Pan Sensitivity	<input type="text" value="1"/>
Snap Spacing	<input type="text" value="10"/>
Key Pan Speed	<input type="text" value="20"/>
Zoom Speed	<input type="text" value="0.01"/>
Zoom Scale	<input type="text" value="0.2"/> to <input type="text" value="1"/>
Auto Save Interval	<input type="text" value="5"/>
Max Undo	<input type="text" value="50"/>
Show Tooltips	<input checked="" type="checkbox"/>
Show Title in Tab	<input type="checkbox"/>
Ping Asset	<input checked="" type="checkbox"/>
Confirm Deletes	<input checked="" type="checkbox"/>
Prompt to Save	<input checked="" type="checkbox"/>

General Settings

To change the storage location you must select a Resources folder in the project.

Storage Location

The Name Map is a generated class used to reference AIs by name.

☒ Auto Generate

Name Map Location

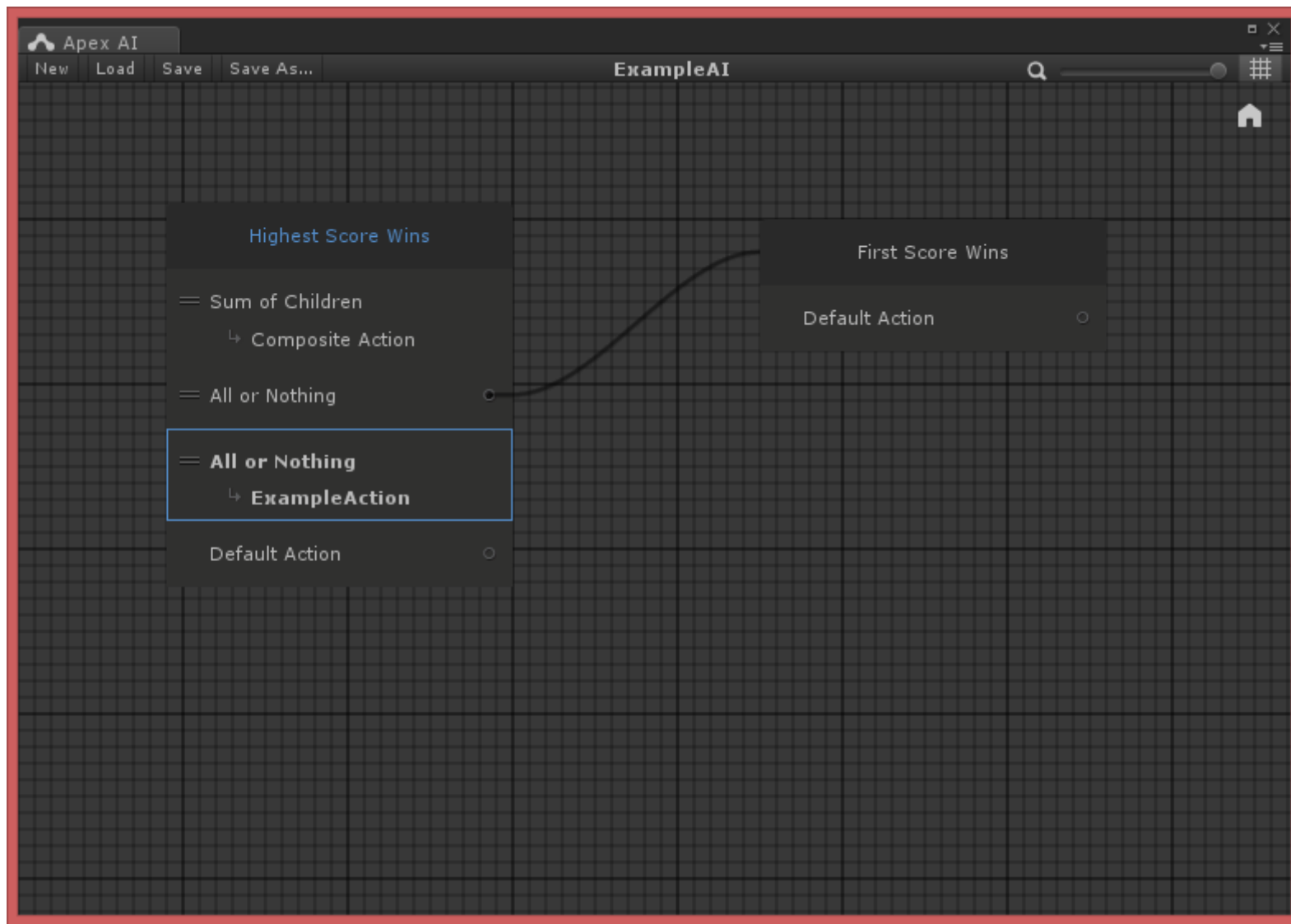
Multiple Parallel AIs

Multiple Utility AIs can run in parallel. This is done simply by adding more than one UtilityAIClient or more than one Utility AI to the Utility AI Component in the Unity Editor, if you prefer to use a component-based model.

Renaming Classes

The Utility AI can repair the AI if you change the name of classes in the code. In the following example, we have an Action called “ExampleAction”.

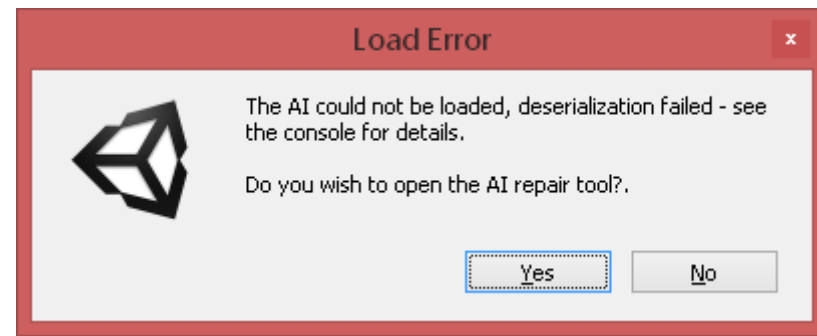
```
1 public class ExampleAction : ActionBase
2     {
3         public override void Execute(IAIContext context)
4         {
5             //do something
6         }
7     }
```



We rename the class to “ExampleActionEdit”

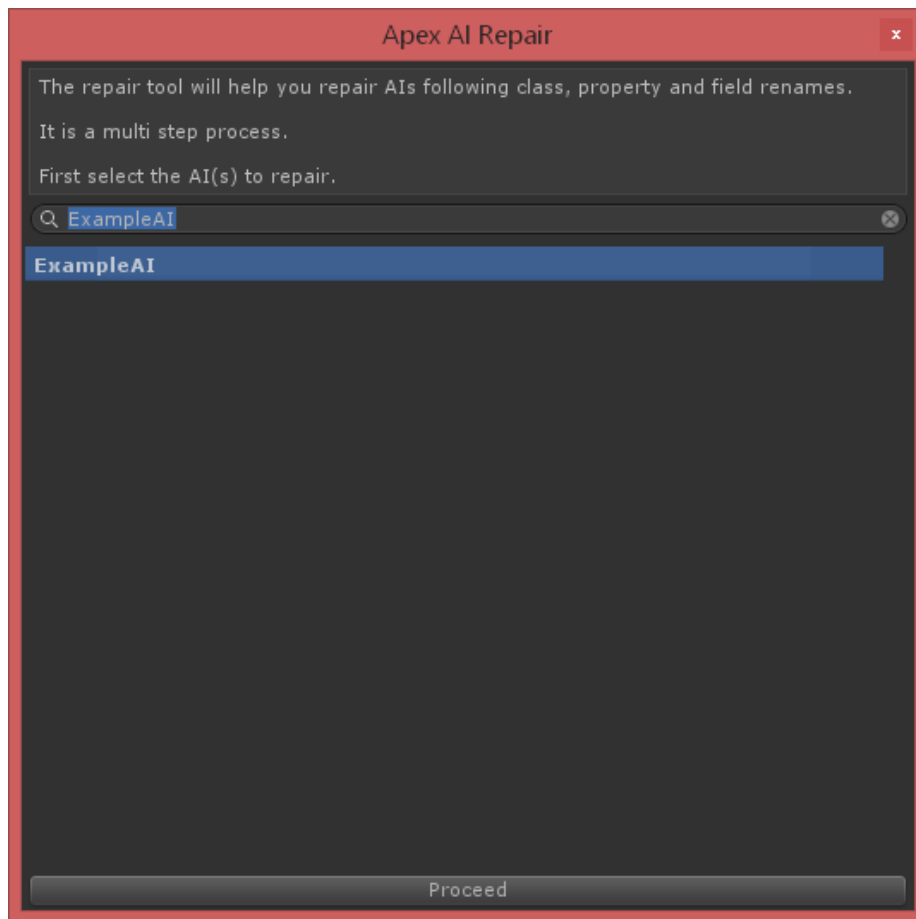
```
1 public sealed class ExampleActionEdit : ActionBase
2     {
3         public override void Execute(IAIContext context)
4         {
5             //do something
6         }
7     }
```


When returning to Unity, we get an error when Unity has recompiled.

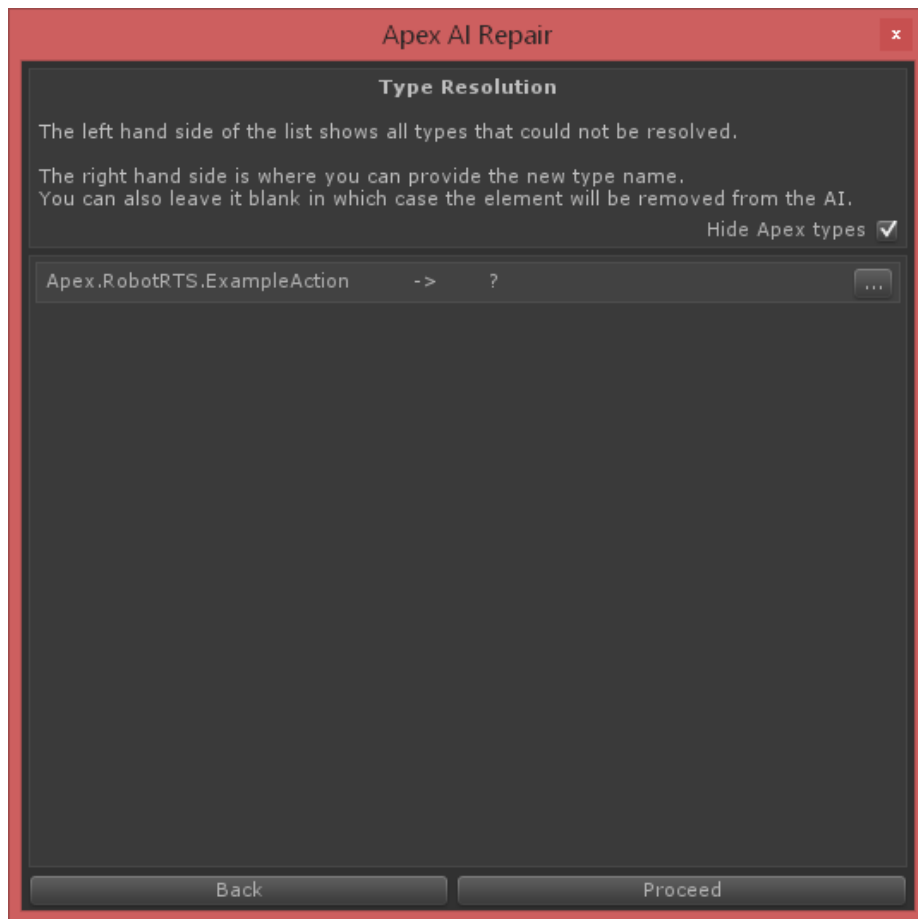


We then click “Yes” to the prompt whether we want to open the repair tool.

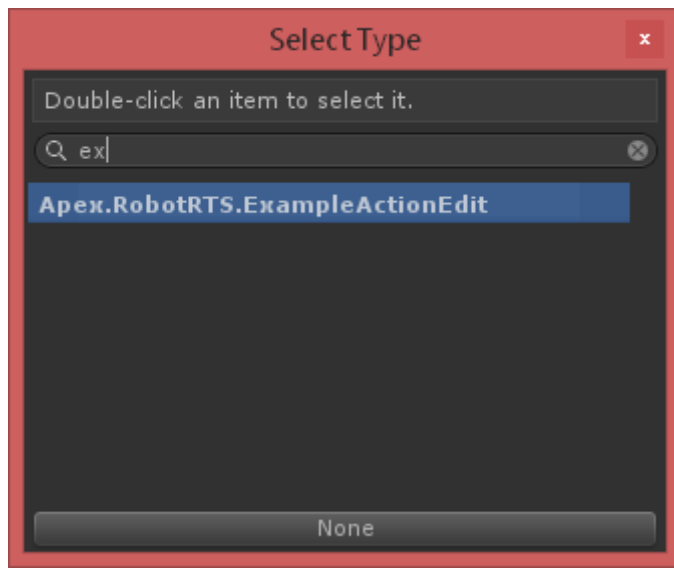
The “Apex AI Repair” window opens. It has already populated the list of AIs with that need repair. In this case, there is only one AI, the “ExampleAI”.



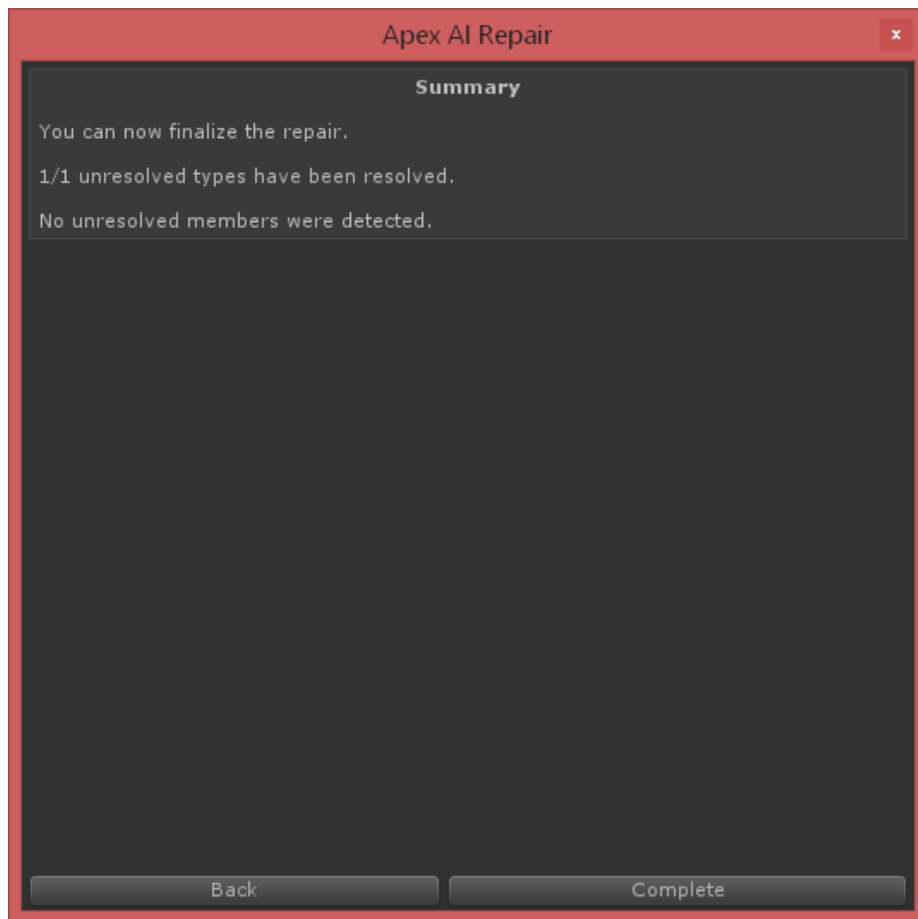
Click "Proceed". The Type Resolution windows opens. In this window you select the class that the entry that can no longer be found should be mapped to by clicking the button with the dots ("...").



The available types can be chosen from the search box that opens.



Once the correct types have been mapped, you click proceed. Any unresolved members are shown in this window. Once the unresolved members have been resolved, you can click “Complete”.



The AI is now repaired.

Using the ListBufferPool

Garbage collection can kill framerate, create spikes, and should best be avoided in game development. For this purpose, the Apex Utility AI provides a buffer pool for C# lists to avoid untimely garbage collection.

Usage:

```
//get a list allocate from the buffer with a capacity of 5
```

```
var gameObjects = ListBufferPool.GetBuffer<GameObject>(5)
```

```
//return the list to the buffer pool after usage, so other parts of the code can reuse the list
```

```
ListBufferPool.ReturnBuffer<GameObject>(gameObjects);
```

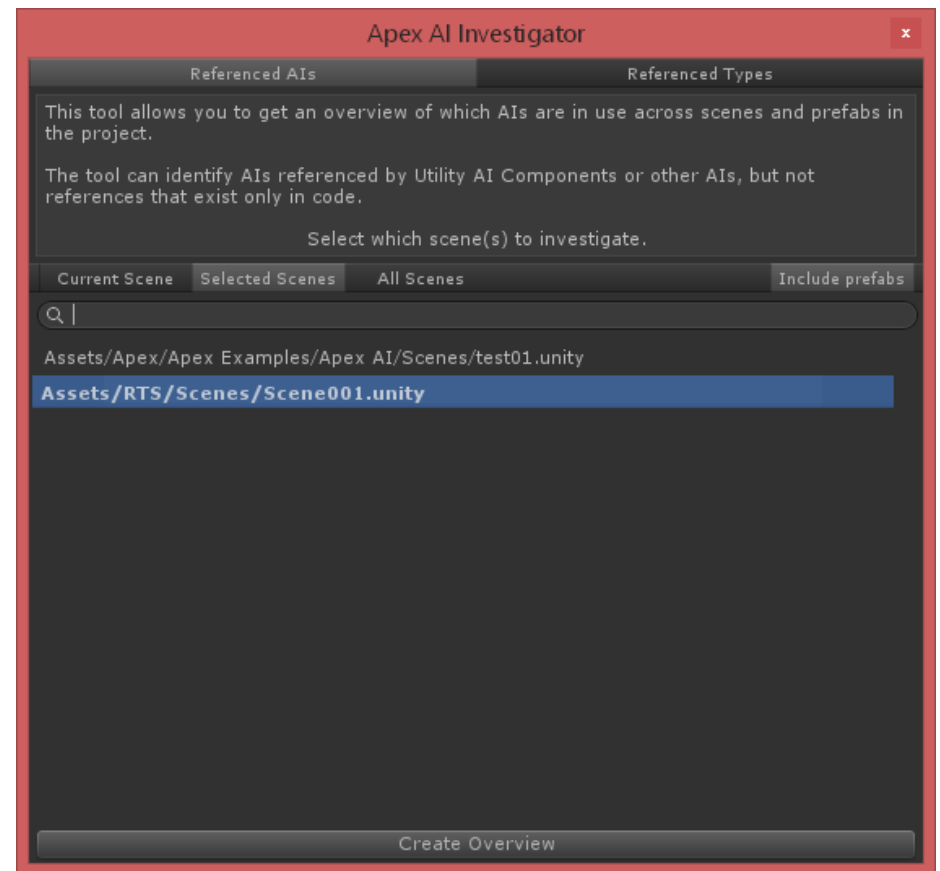
Using the AI Investigator

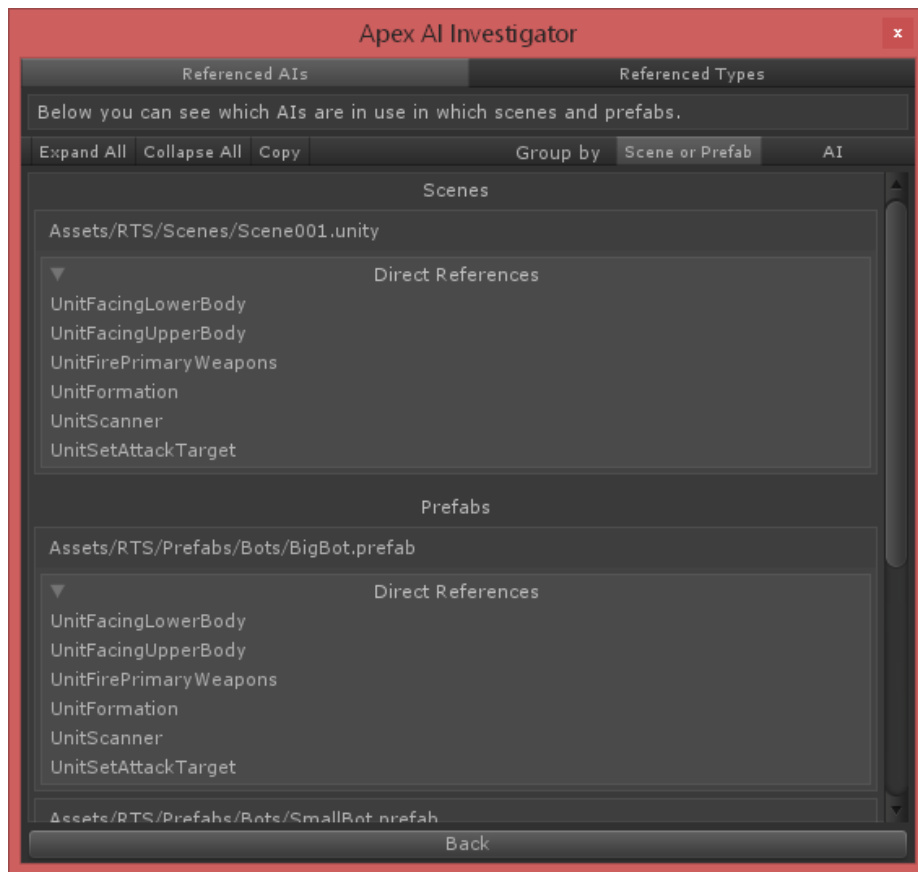
The AI Investigator allows you to identify which AIs are used in each scene.

To open the AI Investigator go to Tools => Apex => AI Investigator.

The scenes available in the project are shown in the list. By selecting the scene you want to investigate and pressing “Create Overview” the AI Investigator finds all references to AIs in the scene.

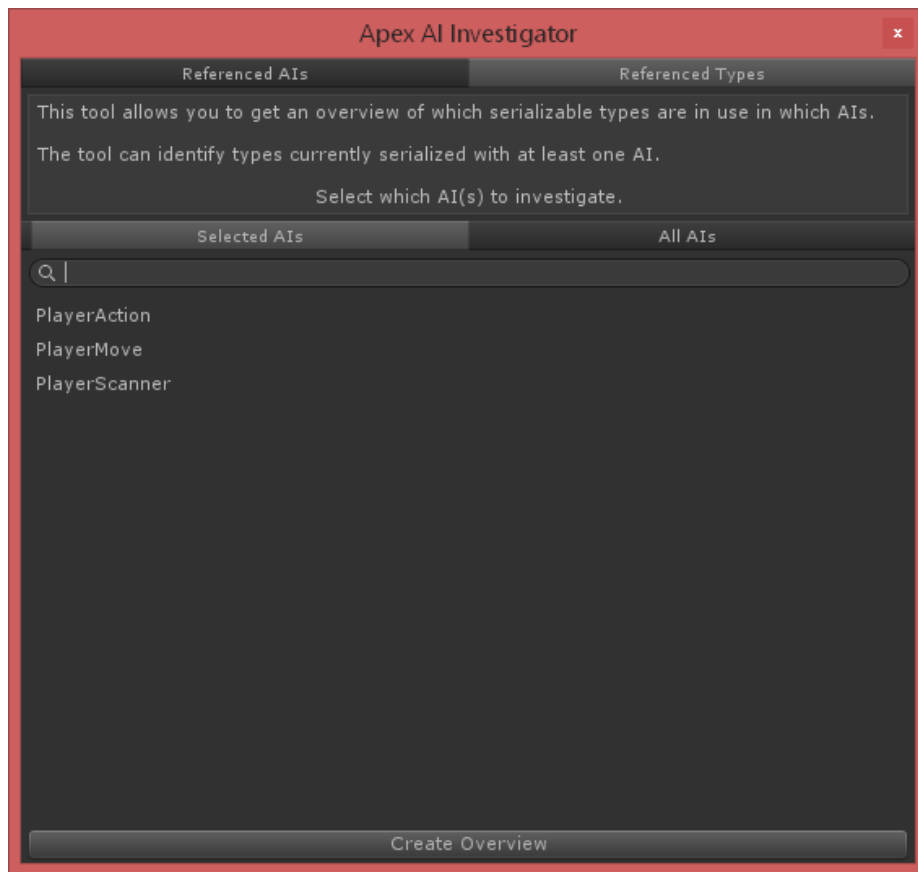
The AI Investigator also allows you to find what classes are referenced by what AIs. This is a useful tool for creating an overview of the usage of each class and for e.g. cleaning up projects.





Click the “Referenced Types” tab in the top of the AI Investigator window to access to referenced types functionality. You can now either find type references only on selected AIs or on all AIs at the same time.

If you select “All AIs” and click “Create Overview” you will get to the overview of references, indicated by the highlighted “Referenced Type” tab. In this view, you can see what AIs each class is referenced by.



If you click the “Unreferenced Type” tab, you can see which classes are not referenced by any AI.

Apex AI Investigator



Referenced AIs

Referenced Types

Below you can see which types are referenced by which AIs, and which are not referenced at all.

Expand All

Collapse All

Copy

Group by

Referenced Type

Unreferenced Type

AI

Referenced Types

Apex.UnitySurvivalShooter.CanThrowBomb



Referenced by

PlayerAction

Apex.UnitySurvivalShooter.EmptyAction



Referenced by

PlayerAction

PlayerMove

Apex.UnitySurvivalShooter.EnemyProximityToSelf



Referenced by

PlayerAction

Apex.UnitySurvivalShooter.FireAtAttackTarget



Referenced by

PlayerAction

Apex.UnitySurvivalShooter.HasBandAids



Referenced by

PlayerAction

Apex.UnitySurvivalShooter.HasEnemies



Referenced by

PlayerAction

PlayerMove

Apex.UnitySurvivalShooter.HasEnemiesInRange



Referenced by

Back

Apex AI Investigator

Referenced AIs

Referenced Types

Below you can see which types are referenced by which AIs, and which are not referenced at all.

Expand All

Collapse All

Copy

Group by

Referenced Type

Unreferenced Type

AI

Unreferenced Types

Apex.UnitySurvivalShooter.HasAttackTarget

Apex.UnitySurvivalShooter.HasBombs

Apex.UnitySurvivalShooter.SetBestMoveTarget

Back

Apex Game Tools

Delivering next-generation artificial intelligence and tools for games within all genres.

Trusted by more than 3000 companies and developers world wide, using Apex Game Tools technologies to power they're games.

[Apex Path](#)

[Apex Steer](#)

[Artificial Intelligence](#)

[Player Experience Testing](#)


[Procedural Content Generation](#)


[About](#)

[News](#)

For faster checkout, login or register using your social account.

 Facebook

 Log in with Twitter

 Log in with Google

in Log in with LinkedIn

D Log in with Disqus
