ScriptEase II Translator Developer Doc

# Introduction

Translators are one of the core features of ScriptEase II. They provide the core functionality of one of ScriptEase II's primary features: game independence. Using translators allows a modular system for generating code and writing it to the game data directly. Although writing a translator can be daunting, it is becoming increasingly easier as the ScriptEase II team adds more tools for developers. However, programming experience is required to follow along with this document. If a section referencecs a topic that has not been discussed, it will be discussed in another section.

# Parts of a Translator

A translator is composed of various parts. Each of these will be discussed in detail, but here is a list.

* A Java implementation of the GameModule interface.
* A Java implementation of the Resource interface.
* The Translator Description File, named *translator.ini*.
* API Dictionary, written in XML.
* Language Dictionary, written in XML.
* Optional libraries.

# The GameModule

To begin creating a translator, we need to implement the GameModule interface from ScriptEase II using Java. The GameModule implementation loads data from the game file into ScriptEase II and then saves ScriptEase II data back into the game file.

This requires a class that extends GameModule, and thus implements any of the necessary methods. These methods can be found in the GameModule.class file, and they are outlined in the associated Javadoc.

It is important to remember that you do not need to keep all of your translator code in this class. A common method of implementing translators is to represent parts of your game as Java objects. For example, the Unity translator has classes for Unity Resources, Unity Types, Unity Script files, and more.

If you have access to the source for the Neverwinter Nights and Unity Translator, we recommend looking at those before writing your own translator.

# The Resource

Implementing the GameModule interface will also require you to implement the Resource interface. Although there should only be one class extending GameModule in your translator, you can have multiple different Resources. A Resource represents an object in the game, such as a creature.

This class is also written in Java, and must extend the Resource class from ScriptEase II. See the associated JavaDoc for information on how to implement it.

# The API Dictionary

**Note: The creation of libraries, including the API Dictionary, is slightly complicated at the moment since the Library Editor is not fully implemented yet. It is currently under development. Please follow the guide to avoid headaches.**

The main library for a translator is called the API dictionary. This is where all of the default types, causes, effects, descriptions, and other story components are defined. It acts as an interface between ScriptEase II and the game API, and is used to determine what code to generate from the story file. To create the API dictionary, first create a new text file with the extension ".sel" and the following code:

<**LibraryModel** *name*="Lib" *author*="Auth" *description*="Text">

<**IncludeFiles**/>

<**Types**/>

<**Slots**/>

<**Causes**/>

<**Effects**/>

<**DescribeIts**/>

<**ControlIts**/>

<**TypeConverters**/>

</**LibraryModel**>

## Include Files

Include files are references to script files that should be automatically added to the game file. They take the format:

<**IncludeFile**>SEAnimationController.js</**IncludeFile**>

The GameModule implementation has a method, addIncludeFiles(Collection<File> …), that uses these files. They are passed in as a collection of Files, which the method should then deal with. This is not automatically done, as the handling of include files is different for each game. It's something the developer must write in the GameModule class.

## Game Types

Game types refer to types such as integers, strings, and booleans. They can also be lists of enumerations, such as types of camera modes or used to differentiate different types of Resources. They require three fields:

* <**Name**>Creature</**Name**> - The display name of the type
* <**Keyword**>creature</**Keyword**> - A unique keyword used to reference the type
* <**CodeSymbol**>string</**CodeSymbol**> - The symbol used for the type in code, such as int, string, or object.

There are also various optional fields that can help you add more information to a type. The order of these does not matter, as long as they come after CodeSymbol. They are:

* <**WidgetName**>L</**WidgetName**> - The letter or two that appears in the circular type widget in ScriptEase II's GUI beside a resource of that type. If this isn't specified, the first two letters of the type name will be used instead.
* <**Slots**><**Slot**>SlotName</**Slot**></**Slots**> - The slots that are associated with this type. These are used when a subject is created for a Cause to determine which slots that Cause can be attached to, as it is based on type. So a Creature in Neverwinter Nights may be attached to an OnClick slot. Creating Slots and Causes is discussed in a later section.
* <**Enum**>true&lt;Open&gt;|false&lt;Close&gt;</**Enum**> - Enums are used to create a type where the user selects from a list of options. In the example, Open and Close refer to the values true and false. True and false do not make much sense to a non-programmer, so we rename them Open and Close.This could be used for an effect that opens or closes a door. Adding an enum will automatically make the widget name "Li" if no other widget name is specified. It is highly recommended to make the GUI field a JCOMBOBOX if you are using an Enum. See the next point for more information.
* <**GUI**>JCOMBOBOX</**GUI**> - The default GUI for a type is a slot that can have resources dragged into it. However, sometimes we want to be able to define the type inline, as with text, numbers, and enumerated lists. There are three possible values we can set the GUI to: JTEXTFIELD for text entry, JSPINNER for number entry, and JCOMBOBOX for list selection.
* <**Format**>  
   <**Literal**>&quot;</**Literal**>  
   <**Fragment** data="value"/>  
   <**Literal**>&quot;</**Literal**>  
  </**Format**>  
  This is an advanced feature that allows the user to specify the exact format of the type in code. This is required for Strings in most cases, as it allows the developer to specify that there are quotes around the data. The types of code fragments are discussed further in a later section.
* <**LegalValues**>[^\&quot;]\*</**LegalValues**> - A regular expression that allows the developer to define valid values. In this case, some special characters are not allowed. These would be removed before the value of the type is written out.
* <**Escapes**><**Escape** *value*="replaced">replace</**Escape**></**Escapes**> - Escapes replace text in a resource's value with the value. So, if a resource was defined in code as "replace this text", it would instead be written as "replaced this text". It is usually recommended to use the LegalValues field instead.

## Slots

Slots are game event hooks. They are used in code generation and File I/O to know *when* a script will be fired, and *where* that script file goes. All Causes have a slot that is used to attach code. Each slot also has a format attached to it that is defined in the Language Dictionary. This determines how the code attached to the slot will be written. It is necessary to define the default format in <**Slots** *defaultFormat* = "DefaultFormatName">. Slots require the following fields:

* <**Slot** format="OtherFormat">…</**Slot**> - It is possible to use another format for a slot if it does not use the default format. If it uses the default format, the format modifier can be omitted.
* <**Name**>NameOfSlot</**Name**> - The name of the slot.
* <**Keyword**>KeywordOfSlot</**Keyword**> - The keyword of the slot. Used to reference the slot from types and causes.
* <**Condition**>ConditionWord</**Condition**> - This is an optional field, and allows another string to be associated with a slot that can then be accessed by the language dictionary. It is used in the Unity Translator for some slots that use the Update method. They then check the Condition, which resolves to a boolean value, in an if statement.
* <**Parameters**><**KnowIt**>…</**KnowIt**></**Parameters**> - A collection of KnowIt parameters used for the slot. These are used in Implicits to retrieve information from. An example of their use is in Unity, where slots correspond to function calls. A function call may be something like "OnTriggerEnter(other:Collider)". Parameters allow us to retrieve the other type, and also to write out the function name correctly in code by using a Scope code fragment with the data attribute set to "slotparameter" and ref to the name of the parameter. Note that KnowIts are discussed in another section.
* <**Implicits**><**KnowIt**>…</**KnowIt**></**Implicits**> - Implicits are resources that are known by default by a slot. These appear in Causes as green game objects, and can be dragged into the Cause by users. The attached KnowIts are usually a specific type, and usually have a function binding that returns a value.

# The Library Editor

To edit Causes, Effects, Descriptions, Controls, and Blocks, we can and should use the Library Editor. Although not all features are currently implemented, it makes editing these story components easier and less prone to breaking the library.

It is recommended to keep a backup whenever you save your library, so save it in a different location than the install directory and make a backup of the old library before moving the new one into the folder.

To open the library editor, you must first have a translator that can be opened with all of the necessary components. Open ScriptEase II with the translator installed, then navigate to Library/Edit \_\_ Libraries. Choose the default library from the list that appears and click Ok. It will open up and will probably be blank.

## Causes

With the Library Editor open, go to File/New/Cause. This will create a new Cause for your library. You can edit the name of the Cause in the Name field. Adding words in < > will attempt to match them to a parameter or implicit. Subject is added by default. This lets the user drag objects into the subject slot, or drag the implicit away. The names of implicits are associated with the slot that the Cause is attached to.

Visible is whether the component should be visible in the library to users. This is generally used for Causes that are in development.

Labels are used to add text in front of a Cause in the GUI. You can add multiple labels by separating them with a comma. They serve no functional purpose, except for the automatics.

#### Automatics

The getAutomaticHandlers() method in the Game Module implementation allows one to create a map of keywords to Resources to then automatically attach a Cause with that keyword to the resource when the game is created. An example of this would be to have "automatic, Resource1 & Resource2" as one of the entries in the map returned by getAutomaticHandlers(). A Cause that has the label "automatic" would then automatically get added to Resource1 and Resource2. If the types of the Resource and the parameter in the Cause do not match up, an exception will be thrown. These Causes are usually hidden, as they are added by ScriptEase, not the user.

One special automatic label is "gameobjectautomatic". Automatics with this label only get added if the resource is used by a Cause or Effect in the story model. Note that this keyword still needs to be implemented in getAutomaticHandlers() if you wish to use it.

#### Editing a Cause's Slot or Subject

Changing a Cause's slot or subject is not yet implemented in the Library Editor. You will need to edit it manually in code, which will require restarting ScriptEase II with the new library. To save time, it is recommended to create all of your Causes in the Library Editor and give them some names that will identify them in the code.

To actually edit the slot and subject in code, find the relevant cause by searching for its name. Note that angle brackets are renamed to &gt; and &lt;, so searching for a string containing < or > may not return the results you're looking for.

Once you have found the cause, go to the first CodeBlockSource in its CodeBlocks. There you can change the Subject to any of that CodeBlock's parameters, and change the Slot to reference any Keyword that references a Slot in the Slots of the library. Note that adding more parameters is more easily done in the Library Editor. The Cause will attach whatever code is contained in its CodeBlocks to the referred to slot.

## Cause CodeBlocks

Moving back to the Library Editor, we can now edit the CodeBlocks for the Cause. CodeBlocks are pieces of code that are attached to both Causes and Effects. Causes only have one CodeBlock. Effect CodeBlocks are talked about in the Effects section. Note that each CodeBlock has a unique ID. If you are editing the library manually, make sure that the code block IDs remain unique.

The CodeBlock editor allows you to add Includes, which are Include Files. This refers to any include files that are necessary for the Cause to run.

The next field is Types. This is basically the return type of the CodeBlock. Types for Causes are usually Void, so it helps to add a Void type into your types listing. Depending on the Language Dictionary implementation it may be possible to leave this as "no types".

We now reach Parameters. There is by default one parameter named subject, whose name we cannot change. However, we can change its type.

#### Special Automatic Parameter

Parameters are a type of story component known as KnowIts. Each KnowIt has a binding. A KnowIt without a binding is shown as a slot. When the user drags a resource into the slot, the KnowIt gets a binding attached to it. We can make things automatically bind to these KnowIts by going into the XML code, and changing the <**Binding**/> of the KnowIt to <**Binding** *flavour*="automatic"/>. This will use the Game Module implementation's getAutomaticHandlers() function to get all resources attached to the keyword "automatic" and attach the first one to the parameter. It is then not necessary to show the parameter in the name of the component.

## Adding Code to CodeBlocks

Adding code uses a special editor inside the code block. Since we need to refer to various parameters and other data, we cannot just use a text editor. Instead, code is organized into code fragments.

### Line

Line fragments represent a line in code. Any other fragments added to a line fragment will appear on the same line. Thus, code is usually separated into line fragments to make it look more readable once it is generated.

### Indent

An indent fragment represents an indented part of the code. This is generally used with line fragments, where one or more line fragments are added into an indent fragment.

### Scope

Scope fragments change the scope of code generation to refer to a different part of the code. The most common usage is to scope into parameters so that we can then use simple fragments to access their code. This is done by setting Data to ARGUMENT and the NameRef to the name of the parameter.

### Series

Series fragments are used to iterate over a series of data, like a for loop in code.

### Simple

Simple fragments are used to add data to the code. They are commonly used in conjunction with scope fragments, such as for parameters, where the data is commonly NAME if the parameter value is defined before the CodeBlock's code is written, or VALUE if everything is occuring inline.

### Literal

Inserts literal code into the code. Anything typed in here will show up as written.

### Reference

References a format defined in the Language Dictionary. Most causes contain only one Reference fragment as their code with a reference pointing to a format that writes out all of the children of a Cause.

### Up

Moves the selected fragment up or left, depending on the way it is ordered.

### Down

Moves the selected fragment down or right, depending on the way it is ordered.

## Effects

Effects are created in a similar way to Causes. One main difference is that their first code block does not have a subject or a slot. Do not add these manually!

Another difference is that effects can have multiple code blocks. Note that this is a very advanced feature. The second Code Block can have a subject and a slot that is derived from the parameters of the effect. This is then attached to that subject and slot at the same time as a Cause is. One example of this was for the Neverwinter Nights translator where an effect needed to check the return value of a script that was attached to another slot that returned an integer.

## Descriptions

To create a description, start by creating one from File/New/Description. You can now edit its name, and the type(s) it returns.

You can now use the graph editor to create the graph users will use to select a relevant DescribeIt. Note that if you only have one path, the graph will not show up, so you could just leave a placeholder node.

Once you have the paths you want, you need to create effects that match up with the paths. The CodeBlock in these effects must have the same type(s) as the description. The code itself should usually return some value. If your effect does anything in game apart from returning a value, it should be explicitly made clear in its name.

With the effects finished, we can click on the description again to edit it. Select a path in the graph, then switch the open library tab to Effects. Find the effect that matches the path and drag it into the box that says "No Effect" at the top. Do this for all of the effects to create your description.

## Controls

Controls are common groupings of story components among all translators. These include the "After # seconds, "Repeate the following # times," and other general controls. They are created in a very similar way to effects. However, they usually only contain a Format Reference Fragment in the code, which references a format for the Control created in the Language Dictionary.

Note that the Question and Pick controls are implemented in different ways in the ScriptEase default library, and thus do not need to be added in your games library. However, their format does need to be added in the language dictionary.

## Blocks

Blocks are created in the exact same manner as Controls. In fact, blocks could be defined as game-specific controls.

# Language Dictionary

The language dictionary defines the formats for how code should be generated. For example, Causes may be the start of a function call, and so need to know that they should write "function <CauseName>" in code generation.

There is not yet an editor for the language dictionary. However, editing it is much easier than the library, since order of formats does not matter, and there are no unique id numbers.

The language dictionary is also written in XML. Here is the layout:

* <**IndentString**> </**IndentString**> - The string used for indent code fragments. Some code generators may use four spaces as opposed to tabs.
* <**ReservedWords**><**Word**>else</**Word**></**ReservedWords**> - Words that should not be used in code generation. For example, we should not name a variable GameObject in Unity.
* <**Formats**><**Format**>…</**Format**></**Formats**> - Various formats used to generate code. It is recommended to consult the Unity translator's language dictionary and to construct your language dictionary based on it.

# Translator Description File (translator.ini)

Every translator must have a text Description file named translator.ini that details the locations and values of the translator's mandatory and optional components. Mandatory components include a name and both API & Language Dictionary locations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Required** | **Key** | **Format** | **Description** |
| Yes | NAME | String | The name of the translator. |
| Yes | API\_DICTIONARY\_PATH | Path | The relative path to the API dictionary. |
| Yes | LANGUAGE\_DICTIONARY\_PAT | Path | The relative path to the language dictionary. |
| Yes | GAME\_MODULE\_PATH | Path | The relative path to the module .class file. |
| Yes | VERSION | 2.x | The number x must match the version of ScriptEase that the translator has been built for. With any major new revision of ScriptEase, older translators will no longer work. If this happens, change the version number, check if everything is working correctly, and then save. |
| No | OPTIONAL\_LIBRARIES\_PATH | Path | The relative path to the optional libraries. |
| No | SUPPORTED\_FILE\_EXTENSIONS | String | The file extensions of the saved game files that the translator can open. "directory" is a valid string for translators that need to open folders. Multiple file extensions should be separated by commas. |
| No | COMPILER\_PATH | Path, or false | The path to the compiler. Since warnings will pop up if no compiler is found, you can write "false" if the translator does not need a compiler. |
| No | GAME\_DIRECTORY | Path | The path to the game. Usually in Program Files. |
| No | ICON\_PATH | Path | The relative path to the small .gif icon that will show up in ScriptEase II to represent the game. |
| No | SUPPORTS\_TESTING | true or false | Whether the translator supports testing or not. This is related to the "configureTester(ProcessBuilder …)" method in the GameModule.class implementation. |

## Example:

**NAME**=Unity

**API\_DICTIONARY\_PATH**=resources/apiDictionary.xml

**LANGUAGE\_DICTIONARY\_PATH**=resources/languageDictionary.xml

**GAME\_MODULE\_PATH**=io/UnityProject.class

**VERSION**=2.4  
**OPTIONAL\_LIBRARIES\_PATH**=libraries

**SUPPORTED\_FILE\_EXTENSIONS**=directory

**COMPILER\_PATH**=false

**ICON\_PATH**=resources/unity\_icon.png

**SUPPORTS\_TESTING**=false