SR Squad Manager Plugin SDK

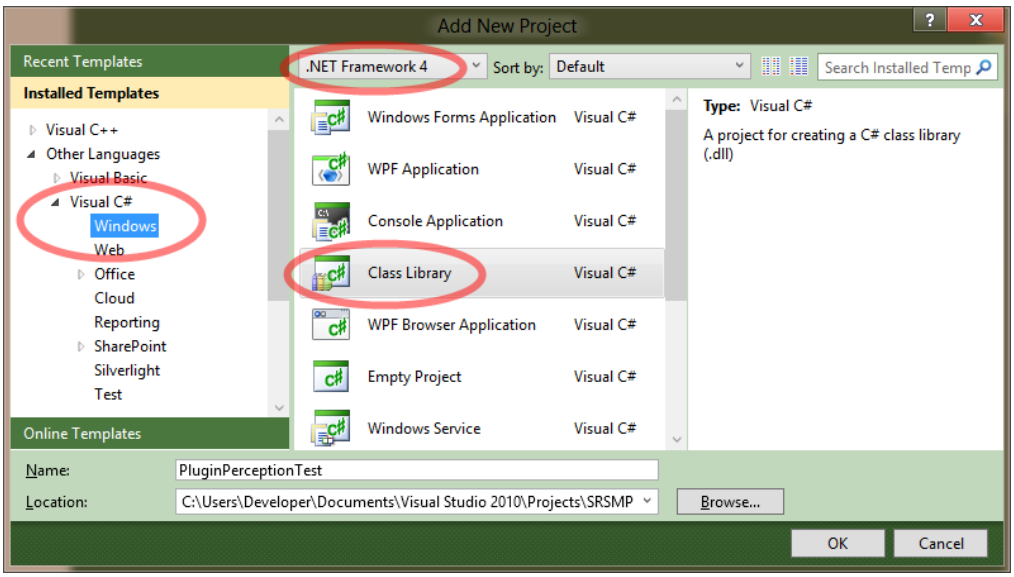
Documentation

To build a plugin for SR Squad Manager, basically a C# class library has to be created, which implements a specific interface. Then the library can be loaded and utilized by Squad Manager.

Contents

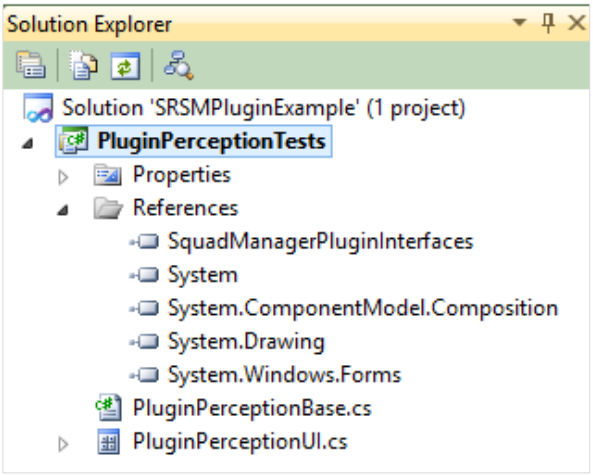
Project settings

A new project has to be created in Visual Studio (the example was created with version 2010/v10) as C# class library for .NET framework 4:



References

Then the references need to be adjusted. Most of the predefined references can be skipped to keep the list clean. Only a single reference has to be kept, but a few more references have to be added from the available .NET 4.0 assemblies. And of course the most important reference from the directory "References" has to be added, to turn the class library into a Squad Manager plugin.



The project structure should look like this:

- *SquadManagerPluginInterfaces*
- *System*
- *System.ComponentModel.Composition*
- *System.Drawing*
- *System.Windows.Forms*

Interfaces

The assembly SquadManagerPluginInterfaces gives a few interface classes, which are used for the communication between plugin and main program. One interface has to be implemented, so that the plugin can be loaded, all other interfaces are used to give data to the plugin.

ISquadManagerPlugin

This is the main interface of the SDK. This interface has to be implemented by the main class of the plugin, so that the plugin can be loaded.

The method GetLocalizedName will be called once during the loading process, to retrieve a localized name of the plugin to display in the menu.

The method GetUI is called to provide a UI window for the plugin. A valid user control has to be prepared and returned here, basic functionality like the actual window with display of the plugin name is provided by the main program though.

ReceiveCallbackInterface is called exactly once while the plugin is loaded. The passed callback object should be stored within the plugin, so that other functions can be used through this callback object later (see ISquadManagerPluginCallback).

The two functions EventGlitchHappened and EventDamageSuffered are automatically called, when a loaded character rolls a glitch or suffers damage. The latter function can also be called when damage is healed. Plugins can directly use the provided data, but will be aborted if the calculation takes too long (currently only 200ms are guaranteed). If the plugin window should be opened, the function should return true or false otherwise (especially if the plugin doesn't want to handle these events at all).

ISquadManagerPluginDebug

This interface can be implemented by the main class additionally to ISquadManagerPlugin to gain access to simple debugging tools. To this end the function ReceiveDebugCallbackInterface will be called additionally during loading phase, to provide an according callback object (see ISquadManagerPluginDebugCallback).

ISquadManagerPluginCallback

Through the callback interface the plugin has the possibility to request different characters, get the distance between two characters (assuming both are positioned on a calibrated map) or roll a die.

The functions GetSelectedCharacter, GetTwoCharacters and GetAllCharacters return one, two or all currently loaded characters, respectively. This can include special actors, like icons, drones or ghosts, not just metahuman characters.

The method GetClosestCharacter returns the closest actor to the given one, assuming that both actors have a position marker on a calibrated map. Only the straight distance is compared; room layouts, walls or closed doors are completely ignored. The distance between two characters can be calculated under the same assumptions and restrictions with GetDistanceBetweenCharacters.

With RollSingleDie exactly a single D6 can be rolled. The returned value is the shown number, all other Shadowrun rules (e.g. hits, glitches, rule of six…) have to be calculated by the plugin.

ISquadManagerPluginDebugCallback

This callback class offers another function, Log, to display simple lines of text in a log window. The log window is opened automatically, as soon as new messages arrive and can be freely closed or opened through a button on the plugin window.

ISquadManagerCharacterInterface

This interface is used to exchange data for actors. This does not always have to be a regular, metahuman character, but also AIs, drones or ghosts can be handled through this.

Through the properties Name, NPC, Type and Description these simple values can be read. The String Type gives a rough idea, what kind of actor you're dealing with: the metatype (e.g. „Dwarf") for metahuman characters, or the class of other actors (e.g. „Ghost"). The value NPC is true if and only if the actor is a non-player character, and false if it's a player character. All these properties are read-only.

With GetImage a suitable picture for the character can be obtained. By providing the optional size parameter, the image is scaled down to be smaller or equal to this maximal size in both dimensions.

Through GetAttribute, GetEssence, GetSkill, GetDamage and other, similar methods different values of the actor are available.

The current dice pool of an actor with a specific skill or matrix program can be queried with GetPool. This already includes pool modifiers like damage modifiers.

With the different GetTestResult methods success and extended tests can be rolled, which includes the full evaluation of SR4A rules.

Some values can be changed through specific methods. With AppendToDescription text can be appended to the character description, with SetDamage the current damage monitor is changed. Not all given parameters may be used in all cases, since it doesn't make any sense to set stun damage for vehicles for example.

With the functions UseEdge, ChangeEdge and BurnEdge the current Edge of the character can be changed. The consumption functions return true if the point was successfully spent or false if no more Edge was available for this. These functions should be used very cautiously of course, especially for player characters.

With the help of SetTag, GetTag and ClearTag arbitrary strings can be stored in a character. The keys for this are in no way restricted, one plugin could therefore easily read or change the values of another plugin. This way some basic communication between plugins is possible. If this is not wanted, the key should be unique, possibly by always starting with the plugin name. If the game master saves the marked character afterwards, all key-value-pairs are stored with it, but there is no guarantee that the character is actually saved.

Parameter classes

Several methods of ISquadManagerCharacterInterface return a specific class object or expect one as parameter.

Attribute

Describes the natural and augmented value of an attribute and gives possible modifiers, that affect all tests with this attribute. The attributes are not limited to the metahuman options, but could also represent matrix attributes for example.

Skill

Describes the rank of a given skill and calculates the full available attribute connected to this skill, including all augmentations and modifiers. If a skill is specialized, this information is given here, too.

TestResultSuccess and TestResultExtended

These two classes hold the results of success and extended tests. For regular success tests the total number of hits is given and the information, if the roll was a glitch. For extended tests the required number of intervals and the net hits of the last roll are given, additionally the information if a glitch was rolled during the test. Should the extended test fail (e.g. due to a critical glitch), the number of intervals is set to 0.

ConditionMonitor

The full condition monitor of the character stores the current and maximal amount of damage for the physical, stun and overflow track, respectively. Additionally the resulting dice pool modifier is given and the flag if a character is "stable" or suffers automatic damage every for combat rounds. For some actors single values might be 0 by default if they wouldn't make any sense otherwise (e.g. stun and overflow damage for drones and

vehicles). If none of the given damage types fits, the physical track resembles the missing damage type (e.g. matrix damage for an AI).

Metadata

In order to be recognized and displayed correctly, the main class of the plugin has to give at least two types of metadata: the name of the plugin (as string for the name "Name") and that of the author (as string to the name "Author").

Main class

Finally it can be said, that a plugin must have exactly one main class which implements at least the interface ISquadManagerPlugin and exports this type together with different Metadata. A minimal class declaration can look like this:

```
[Export(typeof(SquadManagerPluginInterfaces.ISquadManagerPlugin))]
[ExportMetadata("Name", "<Plugin name>")]
[ExportMetadata("Author", "<Author name>")]
public class MyPluginBase: SquadManagerPluginInterfaces.ISquadManagerPlugin {
        //...
}
```

Additionally to that, the plugin has to provide at least one user control for the user interface of the plugin window.