

# Strip Club Wars Modding Tutorial

This document will provide a tutorial on how to create mods for the Strip Club Wars game. This will be a detailed step-by-step process to create a non-trivial mod that will hopefully illustrate a wide variety of things that can be done. Equally important, this document will also address a number of game design issues that should be considered when writing mods.

However, before undertaking this process, it is suggested that you read the Strip Club Wars Modding Guide, and at least familiarize yourself with the content and structure of that document as it will be an invaluable reference. That document should be in the same location as this one in the game distribution. You should also be aware of the cheatsheet.txt file that exists in the runtime folder of the game installation and it's created when the game is run in dev mode.

The files for this mod are all part of the standard game distribution, although the mod itself is not turned on. These files are included in the *mods/sample* folder, with *sample* being the mod key. If you wish to see the mod in action, you can turn it on by adding a line with just the mod key in the *mods/mods.txt* file. To follow along this tutorial, we will create the mod with a different key so you can work on this mod without affecting the original mod files in the distribution. If you get in trouble, you can always consult the distribution files to see the final product. It is however important not to have both the *sample* mod and your tutorial one enabled at the same time as that will create issues.

## Mod Specification

First we need to define what our mod should be. We will pick something that's not too complex but at the same time should be interesting enough to explore a number of features available to the mod creator. In a nutshell, the mod will introduce a character that will want to invest on your club, providing a one-time only source of cash for a percentage of future profits.

We will start by listing a number of mod requirements:

1. It needs to work for all club owners (MC and NPCs).
2. The investor will approach a club owner when the club owner is in debt. We will set the minimum debt at \$20,000.
3. The investor will then become a "vendor", someone that the club owner can call as needed (like the private investigator or the drug dealer).
4. The investor will then offer the club owner a chunk of cash for a percentage of profits from here on out. In essence, the investor is buying a share of the club and the club will provide a monthly dividend based on how well it does.
5. The investor will have unlimited cash.
6. The owner can sell up to 49% of his club.
7. The owner can eventually buy out the investor by providing him with a significant amount of cash. The owner can not go into debt to pay him off.
8. If the club is foreclosed, this may cause problems with this mod. However since the MC's club can't be foreclosed we won't worry about it.
9. This is not quite how these types of transactions work in the real world and it's greatly simplified. It may not be balanced for game play either.

## Tools You Will Need

To create a mod you will need a good editor. While you can do this with Notepad or any editor, it is suggested you look into a better editor, specially something that lets you work on multiple files at the same time, and search for strings across all files. My recommendation is to use Microsoft Visual Studio Code, aka as VS Code, which is free and provides a lot of functionality that will make it easier to work with mods. (Note do not confuse it with Microsoft Visual Studio which a more powerful IDE and it's not free).

You can download VS Code from <https://code.visualstudio.com/>.

## Part I: Creating the Mod Framework

This will be the first part of the tutorial. In this section we will just get the framework of the mod up and running and check that you are able to launch and run the mod.

First step is to create a folder for the mod. The name of this folder has to be the same as the mod key. In an installation all the installed mods must have a different key, so to avoid collisions is a good practice to come up with something somewhat obscure but at the same time conveys what the mod is. For this exercise we will use *sampleinvest* as the mod key. We picked a different mod key than the one already in the distribution so that you can work on creating this one without having to remove the existing one.

Let's now create the file hierarchy for the mod. Open a Windows Explorer window and navigate to your game installation. Then go to the *mods* folder and create a new folder and name it *sampleinvest*. Under this folder create the following three sub-folders: *data*, *local*, and *tfel*. Then go into the local folder and create another folder called *eng* under it.

Once you have the mod folder created, let's enable it on the mods.txt file. Open the mods.txt file and just add the following line to it:

```
sampleinvest
```

Save the file and exit it. From here on out, we will assume that when we say “add this line to file X” you know how to open the file X for editing, make the changes and save it.

Now, let's make sure you did that right. Let's run the mod. But before, let's make sure that your environment is setup to facilitate the mod creation process. Go back to the main game folder and edit the *user\_config.dat* file. Make sure that you have the following set there:

```
dev_mode bool true
```

(You may have this value set in the main *config.dat* file, but it's better to leave the *config.dat* file as is and make changes on the *user\_config.dat* file instead).

Now, run the game. You only need to get to the Age Verification prompt, and feel free to exit it there. The mod doesn't do anything yet, we are just making sure it was loaded. Now, look at the game log file. The game log file will normally be at C:\Users\<your name>\AppData\LocalLow\Total Fluke Studios\StripClubWars\Player.log. Keep a bookmark to this directory as you will be going there a lot. Open the game log and search for “sampleinvest” in it. You should see a line like this:

```
Added mod '\Users\XXX\StripClubWars/mods/sampleinvest'
```

and then some more like these:

```
Loaded 0 regions from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/regions.txt
Loaded 0 names from \Users\XXX\StripClubWars/mods/sampleinvest/data/names.txt
Loaded 0 modifiers from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/modifiers.txt
Loaded 0 likes from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/likes.txt
Loaded 0 gossip types from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/gossips.txt
Loaded 0 job types from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/jobs.txt
Loaded 0 room types from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/room_types.txt
Loaded 0 law types from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/law_types.txt
Loaded 0 sex acts from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/sex_acts.txt
Loaded 0 scheduled functions types from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/calendar.txt
Loaded 0 archetypes from file
\Users\XXX\StripClubWars/mods/sampleinvest/data/archetypes.txt
```

If you see that, congratulations. You have your first mod up and running. It doesn't do anything yet but it was loaded.

If it didn't work, see if there are any errors in the game log that may indicate what went wrong. Make sure that the folder name and the entry in the mods.txt file match exactly.

## ***Part II: A Mod Stub***

Now it's time for the second part, the mod stub. This is just some basic stuff that will give us a feel for how to create the mod. In this step we will create the initial interaction, which will illustrate how to create the dialogs and the text that goes along with it as well as some of the most basic mod building features.

We will start by creating the interaction that allows a character to interact with the investor. Navigate to the tfl directory of your mod and create a file called *si\_interact.tfl*. If it's a very large mod, you can create subfolders and group the various tfl files into specific folders. But in most cases you don't need to. The file names are not important, you can call them whatever you want, but the extension is critical. Use names that do help you figure out what's in it. In this case we called this file *si\_interact.tfl* because the code for the main interaction in the mod will be on this file.

The TFL files is where you define the logic/code that controls your mod. Add the following lines to this file:

```
# main interaciton module

module ^si_interact

interaction !investor_talk(A, T)
```

```

details "Talk to Investor >", 5, 20, 10, "not_tired healthy owner_only"
init {
    var $allow_it, $ai_prob;
    set $allow_it = true;
    set $ai_prob = 100;
}
allow $allow_it;
aiprob $ai_prob;
{
    var $ans;
    set $ans = decision A, T {
        title "si.interact.title"
        body "si.interact.body"
        choice "ok", 1, "ok"
    }
    return;
}

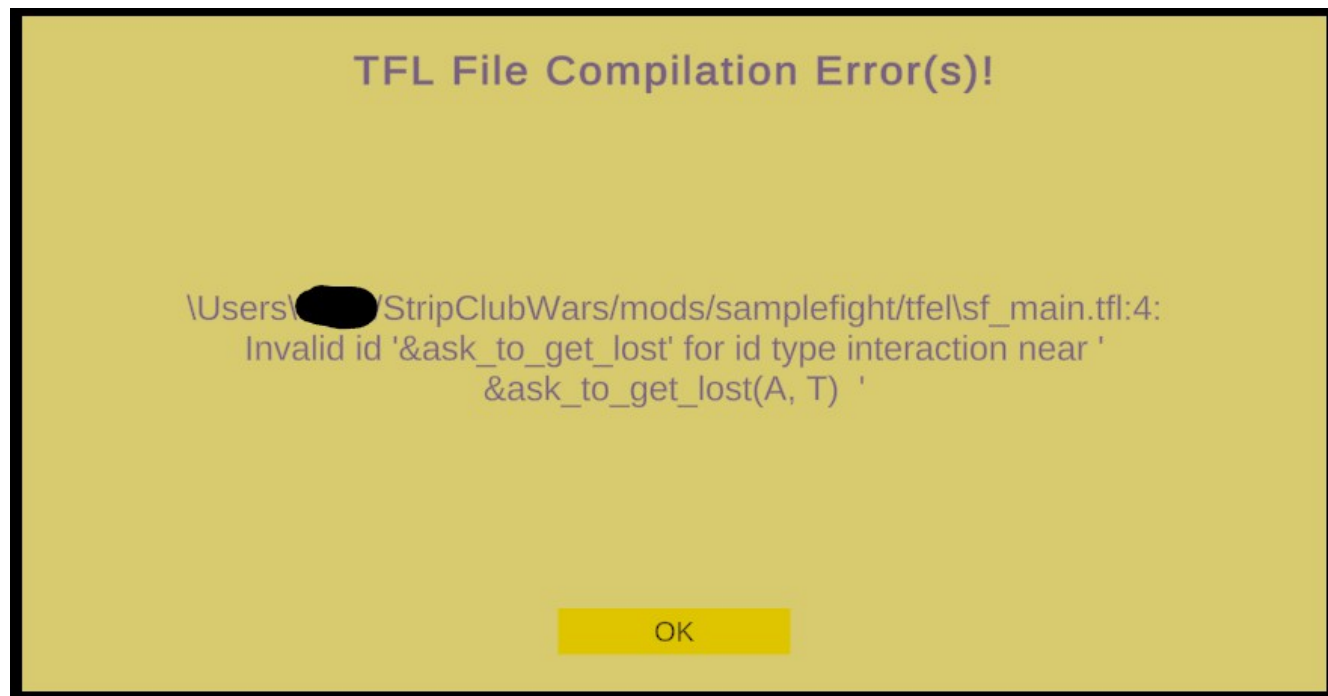
```

Now run the game and load an existing save. Then go to any character and try to interact with them. Assuming you are healthy and not tired, you should see the option to "Talk To Investor". Click on that. If everything worked you should get a dialog that looks like:



If you get that, it's working fine. If you made a typo and there was an error, you may have gotten a

message like this when you started the game:



This error was generated by calling the interaction "&ask\_to\_get\_lost" instead of "!ask\_to\_get\_lost". The TFL code compiler will display any parsing/compilation errors it finds and will not run the game until they are fixed. In general, only one error in a given file will be displayed at a time. That's because parsing errors tend to get the parser out of sync and may result in a lot of bogus error messages. So to prevent that, only the first error found on each file is going to be displayed. (Note that the image refers to a different mod than the one described in this document).

If you got such an error, go ahead and fix it, and restart the game. Repeat until the file is error free.

You probably noticed that the dialog had some strings that were not replaced by actual English text. We will do that next. For that we need to edit another file. Let's go ahead and create in the *local/eng* directory of your mod called *si\_loc.txt*. Note also that from here on out, unless explicitly stated otherwise, all files you will be editing will be the version inside your mod directory (or a sub-directory of it). Be careful not to get confused and create these files in the base game directories. That will work, but it will get very hard to impossible to manage in the long run.

Add these two lines to the *sf\_loc.txt* file:

```
si.interact.title Money Talk  
si.interact.body You approach {1:IN} and you give {1:op} a quick update on how  
your business is doing. {1:P} then asks a few questions which you answer and  
offers a few suggestions you didn't ask for. After a few more minutes of back  
and forth, your conversation has run it's course and {1:p} leaves.
```

The contents of the 2<sup>nd</sup> line wrapped, so when cutting and pasting, make sure the entire second line is inserted as a single line and does not break it up into multiple lines. The localization lines must always start with the key, followed by a space and then the text.

Run the game and and try the interaction again and this time you should see the dialog with the proper text.

Let's take a step back and review what we did. We'll go line by line here starting with:

```
module ^si_interact
```

This defines the module. It's the same name as the file, which is the normal practice, but it doesn't have to be that way. Any functions declared in this file can be accessed from other files by including the module name with the function name.

```
interaction !investor_talk(A, T)
  details "Talk to Investor >", 5, 20, 10, "not_tired healthy owner_only"
```

This starts the definition of the *investor\_talk* interaction. Like all interactions, it takes two formal parameters, which we normally call A (for actor) and T (target). The actor is the person running the interaction, the target is who is the interaction with. The details line indicates some parameters for when and how to run the interaction. In order these are:

- The label that appears on the screen. Note we added the ">" at the end to indicate that running this interaction will likely pass time.
- The "5" refers to the probability an AI character will consider this interaction in a given turn. Normally you want low values here unless other conditions make it necessary to have a larger percentage here.
- The "20" refers to the timeout or cooldown in days. This only applies to NPCs. If NPC chooses this interaction, they will not choose it again for at least 20 days.
- The "10" is for sorting the list of interactions in the UI. Higher values appear higher on the list.
- Finally the "not\_tired healthy owner\_only" are flags that specify additional requirements of when this interaction is a valid option. If the character is tired, hurt or is not an owner, they will never get this option. Note that unlike the cooldown and probability, these flags do apply to the MC.

The next part is the init section, which decides whether the interaction is valid or not for a given actor:

```
init {
  var $allow_it, $ai_prob;
  set $allow_it = true;
  set $ai_prob = 100;
}
allow $allow_it;
aiprob $ai_prob;
```

This makes it always valid, but we will revisit this later and add some conditions to it. Then finally is the main code of the interaction:

```
{
  var $ans;
  set $ans = decision A, T {
    title "si.interact.title"
    body "si.interact.body"
    choice "ok", 1, "ok"
  }
  return;
```

```
}
```

In this case all we are doing is displaying a dialog with the text we added to the localization file. The body text was defined as:

```
You approach {1:IN} and you give {1:op} a quick update on how your business is
doing. {1:P} then asks a few questions which you answer and offers a few
suggestions you didn't ask for. After a few more minutes of back and forth,
your conversation has run it's course and {1:p} leaves.
```

It has 4 replacement codes. The first one {1:IN} is replaced with the introduction and short name for the target. So something like "your friend Emma" or "your employee Fred". The next one {1:op} is replaced with the object pronoun for the target (him/her). Finally the last two are {1:p} which are replaced with the subject pronoun for the target (he/she).

Note that the "ok" choice is one a few generic replacement codes that are used often enough that they get very short texts. These are defined in the base game *local/eng/generic.txt* file. It is suggested that you become familiar with these generic strings defined there and to check there often so you don't create duplicates.

## Part III: Mod Initialization

This section will address the initialization of the mod. This will create the investor character and have the periodic checks to see which clubs may be in need of assistance. We will set it up so that once a month or so, we will check which clubs could use a visit by the investor and schedule a visit to those.

Create the file called `si_init.tfl` in the `tfl` folder with these contents:

```
# Investor initialization module

module ^si_init

event %si_init_event()
    details "Initialize Investor", 5, 30, -1
    init {
    }
    allow true;
    aiprob 100;
{
    var $flag_name, $flag_val;
    cont club:clist[];
    char I, O;

    set $flag_name = "si_inv_info";
    set $flag_val = &get_global_flag($flag_name);
    if($flag_val == "") {
        # flag is not set, so initialize it
        set $flag_val = &init_investor($flag_name);
    }
    # the value of the flag is the id of the investor character
    set I = &id_to_char(0 + $flag_val);

    if(I) {
```

```

    # get all the clubs
    set club:clist = &all_clubs(false);
    foreach club:c in club:clist {
        set O = club:c.owner;
        if(O) {
            if(O.debt > 20000) {
                # if club is over 20k in debt, schedule a visit
                &check_visit_to_club(I, O, club:c);
            }
        }
    }
}
return;
}

```

Now to explain a few things about this file:

```

details "Initialize Investor", 5, 30, -1

```

For events, we don't need as many fields as for interactions. The title can be empty as it is not used, but including it serves as a bit of documentation as to what this event does. The 5 indicates that it should run 5% of the time and 30 that it has a cooldown of 30 days. Events are checked on every tick, so a 5% probability translates to a 50% chance that it will run within 13 tries. There are 6 ticks every day, so on average an event with probability of 5 runs within 3 days 40% of the time. The -1 is ignored since that field only matters for interactions and actions. Similarly there are no execution flags.

```

set $flag_name = "si_inv_info";
set $flag_val = &get_global_flag($flag_name);
if($flag_val == "") {
    # flag is not set, so initialize it
    set $flag_val = &init_investor();
}

```

We will use the global flag *si\_inv\_info* to store the id of the investor character. We can't store objects in variables, so we must store their id. Later on we may decide to store more than one value using a delimited string, but for now we only have one value to store. If the flag value is the empty string it means is not set so we call the *&init\_investor* function to create the investor character.

```

set I = &id_to_char(0 + $flag_val);

```

The *&id\_to\_char()* built-in function converts a number into the character object with id with that number. The flag value is stored as an integer so the *0 +* converts it to a number.

```

if(I) {
    # get all the clubs
    set club:clist = &all_clubs(false);
    foreach club:c in club:clist {
        set O = club:c.owner;
        if(O) {

```



```

        if(O.debt > 20000) {
            # if club is over 20k in debt, schedule a visit
            &check_visit_to_club(I, O, club:c);
        }
    }
}
}

```

It's always a good idea to test for valid objects. The **if(I)** expression makes sure that I contains a valid character object. Then we get a list of all the clubs (*&all\_clubs* is a built-in function that returns a container with all the clubs. Then the **foreach** loop iterates thru each club and we use the *.owner* property on each to get the club's owner. Then the *.debt* property of the owner tells us how debt an owner has (the club's finances are the same as the owner's finances). If its over the threshold, then we check to see if the investor should visit that club.

So that's the main code of the initialization. If you try to run this as is, you will get two errors, for two functions that are not defined: *&init\_investor* and *&check\_visit\_to\_club*. So we will add those next. But for now, let's just add a couple of stubs so we can test what we have so far. Add these lines to the same file:

```

function &init_investor($flag_name) {
    return "";
}

function &check_visit_to_club(I, O, club:c) {
    return;
}

```

These functions don't do anything yet, but it allows the code to be loaded and the game to run. So go ahead and do that to make sure it's working properly. Load a saved game, and then bring up the console. If you are not familiar with the console, please read about it in the Modding Guide. From the console you can then run your event directly and not have to wait for it to be kicked off on it own. To do so, enter this in the console:

```
run &si_init::si_init_event
```

After you do that, you should get a couple of lines indicating that it ran, but because the code doesn't do anything yet, you won't see anything useful. So to change that, add the following line in the main event code, after the char I, O line:

```
##! DEBUG 1
```

Then after saving the file, in the console type this:

```
reload_file sampleinvest si_init.tfl
```

This tells the game to reload the file *si\_init.tfl* from the mod *sampleinvest*. You should get a message telling you it's replacing a prior version and then dump a long list of modules that are currently loaded. Then do the run command again and this time you will see more output. The line you added is a

DEBUG pragma which tells the system to spit out more debugging information, in particular, information for every built in call made. So you should see something like:

```
StripClubWars/mods/sampleinvest/tfel/si_init.tfl:18: Calling built in &get_global_flag([s]si_inv_info) from obj line 40 [40]
StripClubWars/mods/sampleinvest/tfel/si_init.tfl:18: Built in &get_global_flag returns [s] [40]
StripClubWars/mods/sampleinvest/tfel/si_init.tfl:21: Execute loop ended RT %si_init:si_init_event sln=21 oln=49 state=waiting [49]
StripClubWars/mods/sampleinvest/tfel/si_init.tfl:21: Execute loop started RT %si_init:si_init_event sln=21 oln=49 state=waiting [49]
StripClubWars/mods/sampleinvest/tfel/si_init.tfl:24: Calling built in &id_to_char([f]0.000) from obj line 59 [59]
StripClubWars/mods/sampleinvest/tfel/si_init.tfl:24: Built in &id_to_char returns [char]0 [59]
StripClubWars/mods/sampleinvest/tfel/si_init.tfl:40: Normal return from callable RT %si_init:si_init_event sln=40 oln=112 state=done ret=[i]0 [112]
StripClubWars/mods/sampleinvest/tfel/si_init.tfl:40: Execute loop ended RT %si_init:si_init_event sln=40 oln=112 state=done [112]
```

Note that the `&id_to_char()` function returned a null character (shown as `[char]0` in the debug line). Thus the `if(I)` condition failed and it never even tried to get the list of clubs. That's because the function that initializes the investor is just a stub and it's returning an empty string.

So lets do the initialization of the investor character. Replace the `init_investor` stub function you created earlier with this:

```
function &init_investor($flag_name) {
    char I;
    var $as_str;
    set I = &new_char_for_job("banker", 50);
    &set_global_flag($flag_name, I.id, 0);
    set $as_str = "" ^ I.id;
    return $as_str;
}
```

If you still have your game running, you can use the `reload_file` console command to reload it without having to exit the game. Now do the run command again, and this time you should see a lot more output. You will first see something like "Initialized Person for Susan with archetype nerd". That's the debug that appears when a person is created. You will then also see lines for each club and owner. In fact you should see 5 sets of lines, one for each club.

Now, also let's make sure that the code is also setting the flag properly. The investor's id should be stored in a global flag called "si\_inv\_info". Lets make sure this happened by running this console command:

```
dump e
```

That should spit out the contents of the environment object (also known as the World object). There should be a row with a "Flags:" label and in it there should be an entry that says something like "si\_inv\_info='1234';", the value will be the character id of your investor.

Now if you run the code again, it shouldn't create a new investor character. So lets try that. You should still see it giving the info on all the clubs and owners but the line towards the top that told you it initialized a new person should not be there.

Note however that now that you have that chunk of code working, if you need to make changes to it (as we will shortly), you can't just run it on the existing game. You will need to reload the game from before it was initialized. Seems obvious but sometimes you may forget and wonder why things aren't working the way they should. Always remember the initial state.

Now, we need to update the function that determines if the investor should visit a club, and if so schedules the visit. Replace the `&check_visit_to_club` stub function with:

```

function &check_visit_to_club(I, O, club:c) {
  var $last_visit_fn, $last_visit_when, $today, $diff, $days;

  # we checked the debt before calling this, so no need to do it again
  # we don't want to visit each club more than once every 90 days
  set $today = &curr_date();
  set $last_visit_fn = "si_lv_" ^ club:c.id;
  set $last_visit_when = 0 + &get_flag(I, $last_visit_fn);
  if($last_visit_when > 0) {
    set $diff = $today - $last_visit_when / const:TICKS_PER_DAY;
    if($diff < 90) {
      # visited less than 90 days ago, so don't do it yet
      return;
    }
  }

  # see if already invested with this owner
  var $prev_invest;
  set $prev_invest = &get_flag(O, "si_investment");
  if($prev_invest != "") {
    return;
  }

  # schedule a visit
  set $days = &random()*7 + 2;
  set $days = &round($days);
  sched $days days, 4, &si_interact::visit_club(I, O, club:c);
  debug "Investor " ^ I.name ^ " will visit " ^ O.name ^ " in " ^ $days ^ "
days";

  return;
}

```

And also create a stub function for *&visit\_club* in the *si\_interact.tfl* file:

```

function &visit_club(I, O, club:c) {
}

```

A few notes about this function:

- Because there's no way to store an array of values in an object, we create a flag name with the target club in it. This way we can store a different value for each club.
- The *&get\_flag()* built-in returns an empty string when not set which is then converted to 0 when forced into a number. So if the value is 0, it means it's not set. Be mindful of this when actually storing a value of 0 in a flag.
- We will set a flag with name "si\_investment" in an owner object when they accept the investment. If this flag is already set, there was already an investment with this owner so we will not do that again.
- The *&curr\_date()* built in returns the current date as a number, which is really the number of ticks (day periods) since some specific date in the past. Subtracting one such date from another

tells you the difference in ticks. Dividing that value by the number of ticks in a day, tells you the difference in days. The number of ticks in a day can be obtained using the TICKS\_PER\_DAY constant.

- Remember that expressions are evaluated left to right without precedence. So the expression
- `$today - $last_visit_when / const:TICKS_PER_DAY`
- is actually evaluated as
- `($today - $last_visit_when) / const:TICKS_PER_DAY`
- We schedule the meeting in a few days hence to avoid having the investor visit multiple clubs at the same time. Note that it can still happen but is less likely. And it's not a problem if it does happen as the events will be independent of each other. The formula will schedule the event between 2 and 8 days in the future.
- In the **sched** command, the function being called should always be fully qualified (i.e. include the module name)

Also, we can probably remove the DEBUG pragma from earlier. Or change the value from 1 to 0 if we think we may need it later. We are confident that part is working right, so no need to add more clutter to the debug file.

Ok, now exit the game and start it again. We like to do this after making major changes and to reset the game state. And like before run the init event. It should create the investor character and then it should schedule a visit to every club it should. It should print a message when it does that. If it doesn't do that, it could be because none of the clubs are in sufficient debt. In that case try finding a game save that has a club with a lot of debt. If you can't find any of those you can put an owner in debt using the cheat command:

```
cheat add_cash 40 -25000
```

That command will remove \$25,000 from character 40's cash likely putting them in debt. You can use the *list c* command to get list of clubs and their owners and then use that cheat to put one or more owners in debt. You can find out how much money you need to take away by looking at the Manage Club window and then checking each club via the buttons in the bottom row. Once you have one or more clubs in debt, run the event again and this time it should find at least one club to visit. Now you should see output that says something like:

```
Scheduled SCHED_FUNC: 8005 &si_visit_club([char]8042, [char]40, [club]44) to  
run on date 20240421  
DEBUG '\Users\XXX\StripClubWars/mods/sampleinvest/tfel\si_init.tfl:68':  
Investor Scott Walden will visit Cheryl Evans in 6 days [237]
```

Your exact results may vary. Let's explain a few things about these messages. The first line is a platform debug statement. It indicates that a function was scheduled to run. The 8005 is an internal id for the scheduled function which you can ignore. Then it displays the function that was scheduled along with the arguments. The arguments are identified by their value with their type as prefix inside brackets. So `[char]8042` means that the I in the code referred to character with id 8042. Dates are specified as `yyyymmdd`.

The second line is a TFL debug statement. You can tell this because it begins with the word DEBUG. It also shows the filename and line number where the debug originated, in this case like 68 of the file we are working on. The string after is just the contents of the string passed as an argument to the debug

statement. The final [237] refers to the line number in the .tfo file code. The .tfo file is the compiled version of the tfl file located in the runtime/tfel directory of the main game.

The other thing we can do now is to check if the function was scheduled correctly. The debug statement will seem to indicate so, but it's always good to confirm. So run this command in the console:

```
show_sched 10
```

This will display all the events in the next 10 days which should include these club visits. Make sure they do appear on the list. It should look like:

```
SCHED_FUNC: 8052 &si_init:visit_club([char]8042, [char]64, [club]68) 4 Morn Sun Apr 19, 2024  
SCHED_FUNC: 8050 &si_init:visit_club([char]8042, [char]40, [club]44) 4 Morn Tue Apr 21, 2024
```

It should be obvious what that means, but essentially is again the function to call, with the 4 indicating which ticks to run it on and the date. The tick specification is a bit map so 4 = 000100 which means afternoon. Bit 0 is morning, bit 1 is noon, etc.

This is all fine and dandy, but let's make it better, from a game play standpoint. We will have the investor send a message to the owner indicating their visit and allow the owner to accept or cancel. We will also show how to put this on the calendar. This will also give us a chance to introduce decisions.

So we will add a few lines of code between the second **set** \$days line and the **sched** line. We start with:

```
# send message to owner  
var $ans, $knows, $when, $when_str;  
set $knows = &have_relation(I, O);  
set $when = $today - &curr_day_period();  
set $when = const:TICKS_PER_DAY * $days + 2 + $when;  
set $when_str = &format_date("twmdy", $when);
```

We set a variable \$knows to indicate if I and O know each other, using the *&have\_relation* built-in function. This will change the dialog text. Then we get the date of the meeting as a string. This means taking the current date (stored in today), removing the current period, adding the number of day periods for the days before the meeting (the \$days variable) and adding them together. Then the resulting value we pass as an argument to the built-in function *&format\_date* which returns the date as a string.

Then we create the relation if they don't already know each other:

```
if(! $knows) {  
    &make_relation(I, O, "work_cont");  
}
```

We create the relation of type "work\_cont" (work contact) which is usually used for relations dealing with work that are not employees.

Now we add the dialog:

```
set $ans = decision O, -, -, I {  
    title "si.interact.title"  
    body "si.intro.body"~$when_str  
    bodyif ! $knows, "si.intro.not_knows.body"  
    body "si.intro2.body"  
    choice "yes", 100, "yes"
```

```
choice "no", 1, "no"
}
```

And the corresponding entries to the local/eng/si\_loc.txt file:

```
si.intro.body You received a text from {3:IA} saying "Do you have time for a
meeting on %1?
si.intro.not_knows.body We don't know each other, but I'm an investor and
si.intro2.body may be able to help you with your debt problem." Do you want to
meet with {3:op}?
```

Again, be careful to not add extra newlines when cutting and pasting! (We won't remind you of this again).

A couple of things to note here. The decision statement puts the investor character in the third spot. This is the first of the small thumbnails that are used for when the person one is interacting is not right in front of you. Since this is a text, that's the consistent way to do it. We also include the date string we calculated earlier via the ~\$when\_str notation. This value will replace the %1 in the text. The second line of text, where I introduces himself is only shown if they don't know each other (\$knows is false). We give the yes answer a 100 weight and the no answer a 1 weight so AI characters will almost always choose yes.

And finally we need to deal with the response.

```
if($ans == "yes") {
    sched $days days, 4, &si_interact::visit_club(I, O, club:c);
    debug "Investor " ^ I.name ^ " will visit " ^ O.name ^ " in " ^ $days ^
" days";
    set $ans = decision O, -, -, I {
        title "si.interact.title"
        body "si.intro.yes.body"
        choice "ok", 1, "ok"
    }
}
else {
    set $ans = decision O, -, -, I {
        title "si.interact.title"
        body "si.intro.no.body"
        choice "ok", 1, "ok"
    }
}
```

Note that this chunk should replace the **sched** and **debug** lines from the previous version of the file.

If the answer is yes, then we schedule the meeting for a few days hence as before, add a debug to help us make sure things are working, although with the dialog that's less necessary now. Then we display a confirmation dialog. It's usually a good idea to include those dialogs to avoid the scene ending abruptly and giving the user some uncertainty as to whether it was normal or not.

The sched command can be configured to add an entry to the MC's calendar automatically. To do this, create a file called calendar.txt in the data folder of the mod and add this line to it:

```
sched &si_interact::visit_club 1
```

The 1 at the end indicates that if the second argument is the MC, then it will be added to the calendar. If one needs more logic and complexity to make a determination of whether to add a message to the calendar or not, one can use the `&add_to_calendar()` built-in function. But this case is simple enough that it is not necessary.

If the answer is yes, then we just display a different confirmation message. We need to add those messages to the localization file as well:

```
si.intro.yes.body You agreed to meet with {3:F} to discuss how {3:p} can help
you with your debt problem.
si.intro.no.body You decided against meeting with {3:F}. {3:P} was
disappointed but said {1:p} doesn't give up easily so {3:p} may try contacting
you again if your financial situation doesn't improve.

mc_sched.si_interact.visit_club Meeting with investor %1
```

The last line is used to generate the calendar entry. The %1 will be replaced by the first character in the argument list that is not the MC.

Now, save all the files and restart the game. Because you want to see the message come to you, use the cheat command to put yourself in debt. Then run the event. You should then see the dialog come up. Accept it and afterwards check the calendar to make sure it worked. You can also check your contacts to make sure the investor is now your work contact. Then, you should reload the game, put yourself in debt again and run it. This time say no to the meeting to verify that it didn't get scheduled.

## ***Part IV: The Club Visit***

Now we will look at what happens when the investor comes to meet the owner, The big complication here is that there's no guarantee that the owner will be at work or in their office at that time. We can handle this in number of different ways:

1. If the owner is not there, ignore it. Assume the investor came and left cause the owner wasn't there. This is used for some minor events like employee complaints.
2. Automatically move the owner to the club. This is very annoying so it's not used.
3. Send a generic message to the owner telling him it's time for the meeting and allow the owner to go there or cancel. (This is how interviews and amateur contests work)
4. Find a club employee to alert the owner that there's someone looking for them (this is how the mobsters story works as well as the club disaster events).

There are probably other options. The 4<sup>th</sup> option is the most immersive one but also it's the one that requires the most code to create. But the 3<sup>rd</sup> option is the one that's most commonly used and it's a good balance between effort and immersion. In general we used the 3<sup>rd</sup> one for scheduled events and the 4<sup>th</sup> for surprise events. We will use the third one here.

But first we will create a few auxillary functions. The first one we will add to the `si_init.tfl` file and it will be used to update the last visit flag for a given club in the investors object. This is what we use to prevent an investor from visiting a club too often.

```
function &update_club_visit(I, club:c) {
    var $last_visit_fn, $now;
    set $last_visit_fn = "si_lv_" ^ club:c.id;
    set $now = &curr_date();
```

```

    &set_flag(I, $last_visit_fn, $now, 0);
    return;
}

```

It's fairly simple function. It generates the flag name the same way as before and then stores the current time in it. The last argument to `&set_flag` is 0 to indicate that the flag is permanent.

A second function to add to this file is to update the investment flag. This is how we will track how much money the club owns the investor.

```

# The value of this flag will be 4 fields: investor id, share amount,
# current amount owed, current month
function &update_investment_flag(O, I, $share, $amt) {
    var $flag_val, $date;
    cont $data[];
    set $data[0] = I.id;
    set $data[1] = $share;
    set $data[2] = $amt;
    set $date = &curr_date();
    set $data[3] = &format_date("my", $date);
    set $flag_val = &unsplit($data);
    &set_flag(O, "si_investment", $flag_val, 0);
    return;
}

```

This is just a string concatenation of 4 values. The flag is attached to the owner, not the investor.

The other auxillary functions we need are, one to calculate the value of the club and another to figure out how much money the investor wants to invest. Add these two functions to the *si\_interact.tfl* file:

```

# calculate the value of the club
function &calc_club_value(O, club:c) {
    var $value;
    set $value = club:c.expansion_level * 150000;
    set $value = $value * club:c.rel_fame/20;
    set $value = $value + O.cash - O.debt + 50000;
    set $value = 0 - club:c.maint_status*10 + $value;
    if(club:c.damage > 0) {
        # if the club is damaged reduce the value
        set $value = $value * 0.75;
    }
    set $value = 1 + &get_attr(O, "greed", 0.15) * $value;
    debug "Value of club " ^ club:c.name ^ ": " ^ $value;
    return $value;
}

# calculate what share of the club the investor wants
function &calc_want_share( I, $value) {
    var $share;
    if($value > 100000000) {
        # >$1MM = 1% share
        set $share = 1;
    }
}

```



```

}
elseif($value > 4000000) {
    set $share = 3;
}
elseif($value > 1000000) {
    set $share = 8;
}
elseif($value > 500000) {
    set $share = 12;
}
elseif($value > 100000) {
    set $share = 20;
}
else {
    set $share = 25;
}
set $share = 1 + &get_attr(I, "greed", 0.3) * $share;
var $inv_amt;
set $inv_amt = $value * $share / 100;
if($inv_amt > 1000000) {
    set $share = $inv_amt * 100 / $value;
}
set $share = &round($share);
return $share;
}

```

The functions are fairly straight-forward. The club value calculation is based on the club's relative fame, how large it is and it's current status. The owner's greed is factored here, as the higher it is, the higher valuation the owner would expect. The logic for determining what share of the club the investor wants is even simpler. Basically, the higher the club value, the smaller the share. With a limit at the top so that the investment is never more than a \$1MM.

Now we can add the `&visit_club` function to the same file:

```

function &visit_club(I, O, club:c) {
    var $ans, $value, $share, $amount, $base_no;
    # let O knows is time to meet with I
    set $ans = decision O, -, -, I {
        title "si.interact.title"
        body "si.visit.ann.body"
        choice "yes", 100, "yes"
        choice "no", 1, "no"
    }

    if($ans == "no") {
        # O doesn't want to meet with I
        set $ans = decision O, -, -, I {
            title "si.interact.title"
            body "si.visit.no.body"
            choice "ok", 100, "ok"
        }
        &modify_relation(I, O, "cancel");
    }
}

```

```

else {
    # O will meet I, move O to the office
    &go_to_loc(O, "home");
    &set_pose_first(I, "business,bizcas,casual,strip1", 0, 2, true, false);
    &join(I, O);

    set $ans = decision O, I {
        title "si.interact.title"
        body "si.visit.yes.body"
        choice "ok", 100, "goon"
    }

    set $ans = decision O, I {
        title "si.interact.title"
        body "si.visit.prop.body"
        choice "info", 100, "si.visit.prop.opt.info"
        impact "info", "intel", 0.5
        impact "info", "will", 0.25
        choice "no", 100, "si.visit.prop.opt.no"
        impact "no", "cour", -0.5
    }

    if($ans == "no") {
        # O is not interested in the deal
        set $ans = decision O, I {
            title "si.interact.title"
            body "si.visit.no_interest.body"
            choice "ok", 100, "ok"
        }
    }
    else {
        # O wants more info
        set $value = &calc_club_value(O, club:c);
        set $share = &calc_want_share(I, $value);
        set $amount = $value * $share / 100;
        set $base_no = 3 * $share + 50;

        set $ans = decision O, I {
            title "si.interact.title"
            body "si.visit.numbers.body"~$amount
            body "si.visit.numbers2.body"~$share
            choice "yes", 50, "si.visit.numbers.opt.yes"
            impact "yes", "will", 0.1
            impact "yes", "intel", 0.1
            choice "no", $base_no, "si.visit.numbers.opt.no"
            impact "no", "greed", 0.1
        }

        if($ans == "yes") {
            # assign the cash and update the flag
            &update_cash(O, $amount, "other", "si.investment.msg");
            &si_init::update_investment_flag(O, I, $share, 0);
            &modify_relation(O, I, "biz_partners");
        }
    }
}

```

```

        &modify_relation(I, O, "biz_partners");

        set $ans = decision O, I {
            title "si.interact.title"
            body "si.visit.sign.body"
            choice "ok", 100, "ok"
        }
    }
else {
    set $ans = decision O, I {
        title "si.interact.title"
        body "si.visit.no_int.body"
        choice "ok", 100, "ok"
    }
}
}
}
&go_to_loc(I, "home");
&si_init::update_club_visit(I, club:c);
}

return;
}

```

This type of function is hard to read as it typically it's a tree-branching code based on the answers to specific decisions. In this one, the owner is asked whether to meet with the investor, and if so, whether to get more info on the deal and if so, whether to accept the deal. If the deal is accepted, the owner is given cash, a flag is set to indicate the investment was made and the relationship gets a boost.

A couple of other things to note:

1. When moving a character to another's position, use the `&join` built-in. It works better than trying to figure out the other character's location.
2. Typically you will want to get the people in the scene dressed up properly. There's a number of built-in functions to do this. The one used here, `&set_pose_first()` works by picking the first of the options in the list that meets the character's comfort level for the location and the established laws.
3. For AI characters, we set the `$base_no` variable to indicate how likely the character is to turn down the offer. The larger the share the higher the chance it is turned down.
4. We add a new modifier "biz\_partners", so that requires creating a file called modifiers.txt in the mod's data folder and adding this to it:

```

mod biz_partners "Business Partners" 20 1 1
traits honor:0.05 will:-0.01

```

5. At the end of the scene, we must move the investor back to their own home so that they don't linger in the club's office.

After all files are updated and saved, it's time to run the game again. This time we will just unit test the new functions one by one. Let's start with the `update_club_visit`. For this we don't care which character we use. So just bring up the console, run "dump p" and get any character id. We will use the MC because it's easier. Then run the command:

```

run &si_init::update_club_visit char:3 club:32

```

Then use the *dump p 3* command to check the flags for char:3 and make sure they have a *si\_lv\_32* flag set with a value of the current date. You can use the *dump e* command to get the current date. It would be on the first row with the label "date".

Now you can check the *&calc\_club\_value* function in a similar manner:

```
run &si_interact::calc_club_value char:3 club:32
```

You should see the return value being shown in the output. You can try that command for all clubs to compare their values. And then to test the *&calc\_want\_share* function:

```
run &si_interact::calc_want_share char:3 250000
```

You can see the return value as well and can change the value parameter to other values to see how it changes.

Then comes the final test which is the *&visit\_club* function. For this one you should find a second character to use that's not the MC. And pass it as the first argument, with the MC as the second one, like so:

```
run &si_interact::visit_club char:4296 char:3 club:32
```

This should start the interaction and you can try it multiple times giving different answers to make sure all the paths work.

When running these unit tests, make sure not to save the game as the game state will likely get messed up. When you are confident that everything is working correctly, it's time to do an end to end test. So restart the game, put yourself in debt and this time just wait for the event to trigger on it's own. It may take 10-14 days for it to happen but you should see it happen on it's own.

## Part V: Final Touches

There's still one major thing to do. That is the investor needs to get the monthly dividend. Note that the way we are going to do it is the easiest way, not necessarily the way that's most accurate or correct.

We will use the club cash flow value to determine profit. The cash flow doesn't consider some transactions, like loans and personal stuff so it works well for this. The cash flow however, it's very expensive to calculate and we cache for a days at a time. And it only considers what happened in the past week.

We will add the following function to the *si\_init.tfl* file:

```
# Tracks the profits for an investor
# this should run once every 7 days
event %track_profits()
    details "Track Investor Profits", 100, 7, -1
    init {
    }
    allow true;
    aiprob 100;
{
    var $flag_name, $flag_val, $inv_val, $share, $profit, $div, $amt;
```

```

var $now;
char I, O;
cont club:clist[];
cont $data[];

set $flag_name = "si_inv_info";
set $flag_val = &get_global_flag($flag_name);
if($flag_val != "") {
    set I = &id_to_char(0 + $flag_val);
    if(I) {

```

This event will run every 7 days and then update the profits so far for that month. As we did before we get the investor id from the global flag "si\_inv\_info". If it's valid we can proceed.

```

    set $now = &curr_date();
    set $now = &format_date("my", $now);
    set club:clist = &all_clubs(false);
    foreach club:c in club:clist {
        # if club is making a profit
        set O = club:c.owner;
        if(O) {

```

Now we look at every club and see if the investor owns a piece of them. We also get today's date and save it for future use.

```

        set $inv_val = &get_flag(O, "si_investment");
        if($inv_val != "") {
            set $data = &split($inv_val, ",");
            if($data[0] == I.id) {
                set $share = 0 + $data[1];
                # get the profit for the club in past 7 days
                # cash_flow is per day so mult by 7 to get total
                set $profit = 7*club:c.cash_flow;
                set $amt = $profit * $share/100 + $data[2];

```

We then get the "si\_investments" flag from the owner. If this flag is set then it means that the investor invested in this club. We get the share amount from the value of the flag and calculate the new dividend amount. The cash flow is returned as the average per day for the past 7 days, so we multiply it by 7 and divide by the share amount (which is expressed as a percent, so we must divide by 100). Then we add the amount to the prior amount (the value in \$data[2]).

```

                if($now != $data[3]) {
                    # pay last month
                    if($amt > 0) {
                        &update_cash(O, -$amt, "other",
"si.dividend.msg");
                        &update_cash(I, $amt, "other", "");
                    }
                    set $amt = 0;
                }
                &update_investment_flag(O, I, $data[1], $amt);

```

One other thing that we should also do is delete the `!investor_talk` interaction from the `si_interact.tfl` file that we created earlier. That was for demonstrating the basic aspects of the mod and it's no longer necessary. It's very generic and doesn't do anything so all it will do is prevent the NPCs from doing more worthwhile actions.

This mod is not perfect and here's a few ways to make it better. You can try doing these things to practice.

- This concludes this short tutorial. I hope you found it useful. It is but a small drop in all that can be done via modding. The best way to understand the capabilities is to look at the base game TFL code and trying to learn from it.