Tutorial - Adding join points and events

Table of contents

1. Tutorial - Adding join points and events

In the last tutorial we built a very simple application. We're going to see, in this tutorial, how to add some powerful but easy extensions to it. The approach works just as well for automatically creating extension points in plugins as well.

In previous tutorials we have seen various extension points, each custom built to accept particular XML in extensions. There are, however, common extension tasks that are made easy by the plugin system. We will look at these in this tutorial.

2. Setup

We're going to take the application we built in the last tutorial as a starting point and add a few things to it. So, let's copy its contents over into a new directory.

```
mkdir tutorial-5
cp tutorial-4/* tutorial-5/
cd tutorial-5
```

3. Events

When we think about how we'd like users to be able to extend our application, one of the first things that comes to mind is that they might like to respond to events generated by the application (or indeed, other plugins). The plugin system provides a powerful mechanism to create extension points for events.

Suppose we have an event in mind, let's call it *something_happened*. Our event needs a couple of parameters, *int number* and *char* string*. We'd like for plugins to be able to respond to this event.

3.1. Adding the event to the application

In our application, we set up a function pointer for this event and call it. Change our *app.c* file to be:

```
#include <stdio.h>
#include "libplugin/PluginManager.h"

void ( *something_happened )( int number, char* string ) = NULL;

int main( unsigned argc, char* argv[] ) {
    PluginManager_setStandardOptions(
        "APP_",
09          "-",
10          argc, argv,
11          getenv( "APP_PLUGIN_LOGGING" ) != NULL
12      );
13      PluginManager_addAppVersion( "app", NULL );
14      if( PluginManager_start( )) {
15          argc = PluginManager_argc;
16          argv = PluginManager_argv;
17
18          if( something_happened ) something_happened( 100, "it really happened!" );
19          PluginManager_stop();
20      }
21
22      PluginManager_cleanup();
23 }
24
```

We've initialised our function pointer, *something_happened*, to *NULL* and you'll notice that if it's still NULL by the time we get to line 18, we don't do anything. What will happen is that if anyone extends the event with an event handler then *something_happened* won't be *NULL* anymore. This way, we pay very little cost for opening up our application in the case where people don't actually extend things.

Note:

We must be sure that the function pointer is exported as a symbol so that the plugin we're about to write can find it. Therefore it cannot be *static*.

3.2. Adding the extension point

Well, so far we haven't actually defined any extension points. We do this by writing a plugin file. Put the code below into a file called *app.core.xml*.

```
01 <?app version="0.1.0"?>
02
03 <plugin id="app.core" lazy="true">
04
05     <library path=""/>
06
07     <event id="app.core.something-happened"
08         signature="void f( int, char* )">
09         <call symbol="something_happened"/>
10     </event>
11 </plugin>
```

Line 01 tells the system that this plugin matches this application (see the last tutorial).

Line 03 names our plugin, *app.core*, and says it will not be loaded unless explicitly requested or needed by another plugin that is loaded.

Line 05 declares a library. Library are used to find symbols in as we will see later. The empty path is special in that is points to the main application, not a shared library. So, any symbols we try to resolve will be from the main application.

Line 07 says that we will have an extension point called *app.core.something-happened*. The system will build this extension point to be an event for us.

Line 08 tells the system what the signature or prototype of the event is. Events always returns *void*. The signature parsing system has a few restrictions on what can be written but most of C prototypes are possible. See the C-API docs for more information.

Line 09 gives the symbol name of the function pointer to bind the event to. This points to the function pointer from *app.c*. If any plugin extends our event that function pointer will be overwritten with a new function. The new function will call all event handlers registered - it uses FFI to do it's dynamic dispatching.

That't it. Other plugins can now install event handlers to listen to the event.

Note:

There are other ways to create event extension points, with *creators* and *factories* (in the same way as for ordinary extension points). See the plugin file format documentation for more details.

3.3. Listening to the event

We'll now write a listener for the event. Normally, we would do this in a shared library, but just for now we'll cheat and put the code in the application.

Let's add an event handler in *app.c*. Put this at the end of the file:

```
void handle_something_happened( int number, char* string ) {
    printf( "Something happened! number=%d string='%s'\n", number, string
);
}
```

You'll notice that the prototype is exactly the same as the event itself. All we're doing is printing a little information about the event.

As it stands the event hasn't yet been extended. We'll extend it in a new plugin - put this code in a file called *event.extend.xml*:

```
01 <?app version="0.1.0"?>
```

```
02
03 <plugin id="event.extend" lazy="true">
04
05     <library path=""/>
06
07     <extension point="app.core.something-happened">
08         <callback symbol="handle_something_happened"/>
09     </extension>
10 </plugin>
```

Line 05 - we're cheating by putting the code in the main application. Really we should make a shared library and set the path to *path="${plugin.dir}/lib.so"*.

Line 07 declares that we are extending point *app.core.something-happened*. If this plugin is loaded it will also load whatever plugin provides that extension point, which is *app.core*.

The event extension point accepts several styles of XML content in extensions. This is the simplest, it just gives the symbol of a function to call when the event is fired. Note that the symbol is looked for in the list of libraries given in the plugin.

You should be able to run the plugin application with this plugin:

```
./app -plugin-path . -plugins event.extend
```

Gives output:

```
Something happened! number=100 string='it really happened!'
```

There are more interesting things that can happen when an event is extended. For example, you can find the name and signature of the event or even get the arguments as a generic FFI vector. This allows powerful event handlers to be written that can be applied to multiple events at once. See the plugin file format and C-API documentation for more details.

We can also use *creators* and *factories* to create the event handlers. Event handlers are really objects (in the example above the system created an object for us and set its callback method to the one supplied). You can find out about how to write and use then in the plugin file format and C-API documentation for more details. For the moment we'll see how to use one that's already built for us.

The *message* plugin - which we've seen before in previous tutorials - provides a creator for event handlers called *message.event-logger*. We'll change our *event.extend* plugin to use it:

```
<?app version="0.1.0"?>

<plugin id="event.extend" lazy="true">
    <extension point="app.core.something-happened"
```

```
create="message.event-logger">
        <text>Something happened! number=</text>
        <arg-print index="0" format="%d"/>
        <text> string='</text>
        <arg-print index="1" format="%s"/>
        <text>'</text>
        <br/>
    </extension>
</plugin>
```

Note that we now don't need a library element because we aren't requiring any symbols.

The new event handler actually prints exactly the same things as the last one printed! Have a look at the documentation for the *message* plugin for more details.

4. Join points

Events are all very well but sometimes we want behaviour to be replaced. These situations are those when you have some default behaviour but want the user to be able to change it. For example, you might have in a compiler an heuristic for loop unrolling. We might want to replace the heuristic to read from a file which contains a list of factors or to have some other complex heuristic. There are lots of similar situations when you want to make an application extensible.

There is form of programming called *Aspect Oriented Programming* or AOP. It allows you to have replaceable behaviours, too. We'll borrow a few terms from AOP for its concepts. In particular, we have a *join point* (note that we slightly abuse AOP terminology, but too much).

A join point is function that can be replaced. It also has an event which is fired before it is called and an event after it is called.

Above is a diagram of the anatomy of a join point. When we call a join point it first calls all of the *before* event handlers giving them the arguments to the join point. Then the top of the *around* advice stack is called; the top piece of advice may decide call the next advice and so on, meaning that the original heuristic can still be used but need not be called if no one wants it. Finally the *after* event handlers are called. They are given the return value from the top of the *around* advice stack and the parameters to the join point.

4.1. Adding the join point to the application

Suppose that we have some function in mind that we would like to be replaceable. Our function is below:

```
int heuristic( int number, char* string ) {
    return number + strlen( string );
}
```

To make a join point for this function, we change it to be a function pointer:

```
static int heuristic_default( int number, char* string ) {
    return number + strlen( string );
}

int ( *heuristic )( int number, char* string ) = heuristic_default;
```

Note that because the initial value of the function pointer is the default heuristic, then if no one extends the join point the only cost will be calling the heuristic via a function pointer, not directly.

Change our *app.c* file to be:

```
#include <stdio.h>
#include "libplugin/PluginManager.h"

static void heuristic_default( int number, char* string ) {
    return number + strlen( string );
}
void ( *heuristic )( int number, char* string ) = heuristic_default;

int main( unsigned argc, char* argv[] ) {
    PluginManager_setStandardOptions(
        "APP_",
        "-",
        argc, argv,
        getenv( "APP_PLUGIN_LOGGING" ) != NULL
    );
    PluginManager_addAppVersion( "app", NULL );
    if( PluginManager_start( )) {
        int x;
        argc = PluginManager_argc;
        argv = PluginManager_argv;

        x = heuristic( 100, "it really happened!" );
        printf( "The heuristic value was %d\n", x );

        PluginManager_stop();
    }

    PluginManager_cleanup();
}

```

Note:

We must be sure that the function pointer is exported as a symbol so that the plugin we're about to write can find it. Therefore it cannot be *static*.

4.2. Adding the extension points

We need to add the extension points. There are actually three extension points that get created for every join point. One each for the *before* event, *around* stack and *after* advice. We do this in our plugin file, change *app.core.xml* to be:

```
01 <?app version="0.1.0"?>
02
03 <plugin id="app.core" lazy="true">
04
05     <library path=""/>
06
07     <join-point id="app.core.heuristic"
08         signature="int f( int, char* )">
09         <call symbol="heuristic"/>
10     </join-point>
11 </plugin>
```

Line 07 tells the system we are creating a join point. This will give us three extension points:

- *app.core.heuristic.before* - an event with prototype *void f(int, char*)*; note that like all events there is no return value.
- *app.core.heuristic.around* - an around stack with prototype *int f(int, char*)*
- *app.core.heuristic.after* - an event with prototype *void f(int, int, char*)*; note that the first agument is the return value from around advice stack (if the join point returned *void* then we would have no additional argument.

Note:
As seen in the list above, the join point is not in itself an extension point. It is, instead, a collection of three extension points.

You already know how to extend events so we won't do that in the next section. You haven't seen how to add around advice, so we'll do that.

4.3. Adding simple around advice

First we will add some very simple advice, completely replacing the heuristic. Later we will see how we can make use of the original value of the heuristic through the around advice stack.

Our new version of the heuristic will be a function with the same prototype as the original. Normally, this would be in a shared library, but just as we did for events we'll cheat and put it at the end of *app.c*. Put this at the end of that file:

```
int replacement_heuristic( int number, char* string ) {
    printf( "replacement_heuristic!\n" );
    return number * 10;
}
```

Now let's make a plugin to apply this over the heuristic:

```
<?app version="0.1.0"?>

<plugin id="around.extend" lazy="true">

    <library path=""/>

    <extension point="app.core.heuristic.around">
        <callback symbol="replacement_heuristic"/>
    </extension>
</plugin>
```

That's it! If you run the plugin, the new function, *replacement_heuristic*, will be run instead of the original *heuristic_default*. We can also listen to the before and after events at the same time!

4.4. Making use of the around advice stack

Simply replacing the heuristic function is all very well, but what if we want to make use of the old version of the heuristic?

Although we didn't discuss it with events, both events and around advice can take different forms of the *callback* function. Remember that we said that the event handler was actually an object? The same is true for around advice. When you just give a symbol for the function, an object is created for you and its callback method is set to the one supplied.

We can, instead, have your callback function take a first parameter of an *AroundAdvice**. This will give you a pointer to the object that was created for us. We need to tell the plugin system that we are giving a function with that prototype instead and we do that with the *type="non-static"* attribute.

First, let's update our heuristic function, then we'll see how to change the plugin.

```
#include "libplugin/Around.h"

int replacement_heuristic( AroundAdvice* self, int number, char* string ) {
    int oldValue;
    int ( *next )( int, char* );
    next = ( void (*)( int, char* ))AroundAdvice_getCallNextFn( self );

08     oldValue = next( number, string );
09
10     printf( "replacement_heuristic! oldValue=%d\n", oldValue );
11
12     return oldValue * 10;
13 }
```

In **line 01** we have to include the header which defines around advice types and functions.

We add our pointer to the advice in **line 03**. From this we will get a pointer to the next advice on the stack (which could be the original heuristic. We could also find out more information, like the name of the extension point, the types of the arguments and so on.

Lines 05 and 06 get the next function from the advice stack.

Line 08 calls the old function. You can call the function as often as you want (as long as the function itself doesn't forbid multiple calls).

Excellent. Now let's update our plugin to reflect this:

```
<?app version="0.1.0"?>

<plugin id="around.extend" lazy="true">

    <library path=""/>

    <extension point="app.core.heuristic.around">
        <callback symbol="replacement_heuristic" type="non-static"/>
    </extension>
</plugin>
```

That's it! Try running it.

Note:

If you have attribute *selfish="true"* on the *<callback>* element, then no one else will be able to replace the heuristic. This is sometimes useful if you need to make sure that you are on the top the advice stack. In general, however, it is better to not be selfish.