# Extendable Software: An Example system

ECE – extendable contract evaluator – is a very simple model of a contract evaluator for the electricity market. To keep things simple the year is divided into 365 days each of 24 hours[1]. As it stands I am guilty of over engineering the example, not only was I attempting to apply the ideas of layering (Kelly, 2001) and cohesive header (Henney, 2000) in aligning namespaces with directories. I hope to be able to enhance this program for future examples.

The first features to note are:

- `main.cpp` is the "top" of the program. As you may guess, it contains the main function where execution starts. When run the program loops over all available contracts, evaluates them and displays the results.
- `EvaluatorState` class provides the "state" of the system – currently just the price and demand in any hour of the year.
- `AbstractContract` class is the basic work unit. It represents an interface which each contract must conform to in order to be evaluated by the main loop.

There is nothing really radical in this design. The most important thing is the `ContractList` type which is used to store a list of the available contracts. Where these contracts come from is irrelevant to the main processing loop, *it* just knows it has some contracts to evaluate.

Now if we look more closely at a contract, `FixedPriceContract`, which implements the `AbstractContract` interface. This class is compiled in with the main program, in effect hard coded. What is important here is how we could add another hard coded contract. This would entail three steps:

- Deriving from `AbstractContract` write the code for a new contract, e.g. `NewContract`, inside .h and .cpp files, e.g. `NewContract.h` and `NewContract.cpp` and add them to the makefile
- Include `NewContract.h` in `main.cpp` and in the function `AddHardCodedContracts` add `NewContract` to the contract list:

```
contracts.push_back(new NewContract());
```

- Recompile the system

We have now added a new contract to the system with the addition of one line to the existing system. Although we have added a line of code, no existing code was removed, and no lines where changed. Thus the chances of introduced new faults into the existing code have be minimised. Of course, `NewContract` could contain some major flaw itself which would crash the system.

This then is an example of how we can extend code at compile time. Next we want to consider how we may extend the system at run time.

---

[1] In reality we would need to allow for leap years of 366 days, and days when clocks go forward (23 hours in the day) or back (25 hours in the day), and the last I knew electricity contracts where based on half hours.

The file `DynamicContracts.cpp` contains the code for this. The function `AddRuntimeContracts` assumes that any command line parameters passed to the program specify the names of DLLs to be loaded at run time.

For each name specified the function tries to load a DLL. (The search path for these DLLs is system dependent but normally starts with the current directory so I have placed by DLLs there.) If a DLL is found the system attempts "Poor mans COM" and looks for a function called `ContractFactory` inside the DLL. Now, if this is found the function is called and the result assumed to be a pointer to a class derived from `AbstractContract`.

It is worth noting that we are making an assumption here. The DLL could return something completely different. Indeed, the `ContractFactory` could be a completely different function to the one we assume it to be, it could for example require one or more parameters to be passed.

At this point, assuming we have found the DLL, successfully called the `ContractFactory` function and retrieved a pointer to a class we again add it to the contract list. The contract will be evaluated in exactly the same way as the hard coded contract. We have demonstrated run time extensibility of the system.

Before moving on there are some points that require more examination here.

I have provided the `MinPriceContract` which is packaged in a DLL to demonstrate this system. However, the `MinPriceContract` is a separate compilation. Although in my Visual C++ development environment I have included it in the same workspace as the other parts of the system it does not need to be there. As long as it can see the necessary libraries it could be built separately. This is both an advantage and a disadvantage because it opens the door to incompatibilities between it and the rest of the system.

Next we must consider the Windows linking model. Using the Microsoft utility `dumpbin` against the `MinPriceContract` DLL (e.g. `dumpbin /exportrs MinPriceContact.dll` ) we can see the mangled names of functions exported from the DLL. These functions are only exported because `__declspec(dllexport)` was specified for the `ContractFactory` function and the class declaration.

Unfortunately, `declspec` does not exist, and is not needed, on other platforms such as Solaris and Linux. This makes the source code has become platform specific.

Observant reads may have noticed that there is no header file for `MinPriceContract`. The class declaration is itself inside the `MinPriceContract.cpp` file. Outside the boundaries of the DLL an object of type `MinPriceContract` is simply an implementation of `AbstractContract`.

Finally on `MinPriceContract` it is worth noting that there is a "diamond problem" (Meyers, 1997). Since `AbstractContract` only has abstract methods the problem is irrelevant however if we look at the destructor we can start to see the problem.

Although the destructor is a pure virtual it is allowed to have a body. Further, it is actually required to have one by the C++ standard, although this can seem counter-intuitive, not least to compiler writers and you may find slight differences between compilers on this point. For example, gcc 2.95.3 will not allow the body to be specified in the class declaration, it must be specified separately.

When `ExtensibleEvaluator` is compiled one copy of the destructor is generated, and then, when the `MinPriceContract` DLL is generated another copy is generated. This is right because the compiler has no way of knowing that `MinPriceContract` DLL will be used with `ExtensibleEvaluator`.

Now this isn't visible to users. Nor is it a problem in this case because the destructor is trivial, and is inlined anyway so two copies would probably be generated even if the files where all compiled and linked together.

(If you want to see the problem for yourself give the destructor the body:

```
virtual ~AbstractContract() = 0 {
    int x = 1234;
    int y = x;
};
```

And compile DLL. Next change x = 0 and compile the main program. Now step through the destructor calls with a debugger and watch the values of x and y. You will see, despite what your source code shows on screen that x takes the value 1234 when `MinPriceContract` is destroyed and 0 when `FixPriceContract` is destructed.

This is also a good example of how code inside a DLL can get out of step with your source code or the same code inside another executable.)

Of course, as has already been pointed out, this is not a big problem. However, if we were to add a more substantive method to `AbstractContract` , or a static member, we may start problems – at the very least we will see code bloat.

We can conclude that run time extensibility is practical provided we take certain precautions. Link time extension can be even more problematic.

To perform any kind of link time extension you need both some code to link in, say, `LinkTimeContract`, which has been compiled to an object file - .obj on Windows and .o on Unix. You need to include some functionality in this file, which will be executed autonomously of the normal `main`, which starts the program. Your compiler or linker may provide some means of doing this however this will be specific to the compiler or linker.

Alternatively, you can do it with language tricks. A sketch of the solution looks like this:

? What we want to do is execute some code, which is not part of the main program. We need to force some code into the program so that it is executed without modification to main or any of process paths from there on.

? Inside source code of the `LinkTimeContract` we provide for a non-local instance of a class, since we want it limited to this file we will make it static, something like:

```
static StartupObject linkTimeTrick
```

We write `StartupObject` in such a way that all our work is done in the constructor. If we now link this into our main program with the linker we can expect an instance of `StartupObject` to be created at run time before main execute. Note that we do not know what other objects are being constructed since C++ makes not guarantees about initialisation order.

? Inside the constructor we insert a factory function for `LinkTimeContract` into some kind of list of such objects, lets call it `LinkTimeFactoryList`.

? For this list we cannot use a regular C++ array because we don't know how many factories we are going to add to the list or, indeed, in which order or positions. We must therefore use a C++ container type such as list.

? We must now ensure that list<LinkTimeFactory> must be constructed before any of our StartupObject's. Tricky. But as Scott Meyers (1997) described in Effective C++ (item 47) not impossible. We can write:

```
list<LinkTimeFactory>& LinkFactoryList() {
     static list<LinkTimeFactory> factoryList;
     return factoryList;
}
....
StartupObject::StartupObject() {
     LinkFactoryList().push_back(someFactory);
}
```

There are several problems here. First, in reading Meyers we know exactly why we shouldn't use the non-local static.

Second, Meyer's own recommendation has threading problems, if two threads attempt to run the function at the same time we could end up with two objects. Of course, as we are doing this at start-up we should expect only one thread to be running, but can we be sure the compiler implementers haven't come up with some new trick?

Finally, we may not have freed ourselves from the linker and compiler platform problems. A zealous linker or compiler could potentially notice that our `linkTimeTrick` object wasn't being used anywhere and remove it.

Even assuming the code is still present we have no guarantee that it is executed before `main`, and therefore before we attempt to access the factory list. Section 3.6.2 of the C++ standard (ISO 1998) allows the compiler to avoid initialising our `StartupObject` until it is needed, which it never actually is.

This leads us to compiler options. Microsoft Visual C++ supports `pragma init_seg` for this purpose – see Bugslayer, 1997 – which is of course non-portable.

In short, before you attempt link time extension of your code you should look at the problems and issues involved. Then ask yourself if is still better to do it this way. Confronted with these issues you may well decide that compile time extension doesn't look so bad. It is certainly more portable and more reliable.

## *References*

BUGSLAYER, 1997, Introducing the Bugslayer: Annihilating Bugs in an Application Near You, Microsoft System Journal, October 1997.

HENNEY, K., 2000, C++ Patterns Source Sohesion and Decoupling, Overload 39, September 2000.

ISO, 1997. ISO/IEC 14882: 1997. Programming Languages – C++.

KELLY, A., Source Code and Layers, Overload 41, February 2001.

MEYERS, S., 1997, Effective C++. 2nd edition. Reading: Addison-Wesley.