Writing extendable software

"A hallmark - if not the hallmark - of good object oriented design is that you can modify and extend a system by adding code rather than hacking it.... In short, change is *additive*, not *invasive*. Additive change is potentially easier, more localized, less error-prone, and ultimately more maintainable than invasive change."

John Vlissides, *The C++ Report*, February 1998

This is one of my favourite quotes about software development – and I should apologise for mentioning it in more Overload columns than perhaps I should. By the time I originally read this quote I already had several systems under my belt, I had already read the seminal *Design Patterns* (Gamma 1995) and many, many other books so the essence of the quote shouldn't have come as a surprise to me but it did. Vlissides pinpoints and explicitly states something that is only implicit in a lot of writing. You could view the entire contents of the *GoF* book as recipes for extensible code. Maybe, but it was never spelt out quite so explicitly.

Here, I'd like to spend a bit of time talking about what extendable code means to me, and look at some mechanisms for creating extendable systems.

What is extensible code?

One could argue that all code is extensible because all software is infinitely flexible. If we wished we could change our washing machine control software into a nuclear power station control system – we could do it, but it just isn't the best way to do it.

All software is infinitely modifiable, this is a big big problem because the point at which change is impractical is down to individuals' judgement. The decision is based on ones experience with the software, overall experience of software and your current business environment.

While it is possible to change and modified all software, only software which keeps its original shape and absorbs additions gracefully is truly extensible.

For example, I once worked on an evaluator for electricity futures contracts. To add a new type of contract meant: adding a big chunk of evaluation code, changing the user interface, changing the main control loop, adding a new case statement to half a dozen evaluation routines, and a myriad of minor changes throughout the code base. Many of the evaluation routines looked something like this:

```
double ContractPaymentMultiplier(int contractType) {
     swtich (contractType) {
           case 1 : return 1;
           case 2 : return 0.9;
           ....
           default : assert(false);
     }
```

```
        return 1;
    }
```

True, this is C not C++ code, a properly object-oriented system wouldn't be like this but not all C++ is properly object-oriented[1]. Yes, OO supports extensibility better than procedural code but it doesn't force extensibility.

A system built like this can be changed, you can add to it but it involves intrusive changes in many places. Before you could add anything to this software you had to hack it. To borrow a metaphor from the days of 640K limited MS-DOS[2], the software could be *expanded*, but it could not be *extended*.

An extendable system would allow the new contract to be added without any changes to the existing source code. Realistically, we may have to accept "minimum changes" rather than "any changes" but the point is intrusive changes to existing code should be minimised, let's say three places at most – OK, I just pulled "three" out of the air. One or two would be better, but if we are attempting to separate the system elsewhere (e.g. GUI interface separated from calculation engine) then centralising all changes at just one point may break other abstractions.

If my contract evaluator was written as a truly extendable system I may only need to write a couple of new objects: one to represent the GUI aspects of the contract and one to represent the evaluation. Next I would recompile my system with the new objects – maybe I would need to add them to an existing list of contracts or maybe there is some magic in the make process that would do this for me.

To emphasise the point: the system has been changed without need to change the existing code – even though we may recompile the code the existing code is unchanged.

This may seem obvious when I write it here but stop for a minute and dwell on it. Can you do this with your current system? How would your life be improved if you could make changes like this? What would it mean if your system could be changed like this? How can you do this?

When is code extensible?

There are three points at which code may be extended:

? compile time : change our source code to pick up new functionality; this may mean adding new objects in new source code files and changing a factory function.

? link time : arrange for changes to be picked up by the linker only; this may involve some magic for new objects to be found, this can be self defeating as it inevitably adds some obscurity to the code and possibly the makefiles too.

[1] In fact, the system I'm actually talking about was actually written in Pascal.

[2] For those who don't recall. MS-DOS was limited to 1Mb of accessible memory and 640K of user space. Initially to get beyond this Lotus, Intel and Microsoft introduced a system of page swapping which allowed memory to be "expanded", think of the memory map getting fatter as different pages where swapped in and out of the 1Mb memory map. Eventually this system gave way to "extended" memory where the CPU could access beyond 1Mb, instead of getting fatter the memory map got taller.

? run time : dynamically loaded libraries where invented for this sort of thing. This can also lead to obscurity in the code and usually makes debugging more complex because you may have to wait for a library to be loaded before you can set break points.

Although run time extension is truest to the idea of extendable code (because you don't change any of the existing code) I don't think this buys much over a good compile-time extension system. Run-time extension has its uses, such as in very dynamic systems, or non-stop applications but it also complicates version tracking and configuration management.

Sometimes the simplest thing is to actually change some of the existing code. What is simplest and best depends on your circumstances. Actually adding a line and recompiling will be the simplest solution.

Mechanisms for extending code

Many of the classic design patterns are directly concerned with allowing code to be extended with minimal intrusion. It is easy to see how command, chain-of-responsibility and factory patterns can be useful but patterns are not the end of the story. (If this isn't obvious have another look at the GoF book and think about them for a few minutes.)

Interfaces and substitutability

The key to extensible code is common, well-known interfaces, which allow on object, module or library to be substituted for another – the *Liskov substitution principle* (see Martin 1996). The program framework handles all objects in a common fashion, no special cases are allowed, it is oblivious to the concrete type of the object. The same idea lies at the heart of the *dependency inversion principal* – see Griffiths.

In my extendable contract evaluator example, the framework would ask the contract to evaluate itself, it has no need to know anything about the contract class itself, only the interface for communicating with the contract class.

State of the program – data model

One problem we quickly run into when adding new objects to an existing system is that the objects must have access to the state of the program, that is, the data contained in the system at the current time.

Again, think of the extendable contract evaluator example. Before evaluating any contracts the system will load data models of the supply and demand for the period the contracts are being evaluated for. It is useful to centralise the data model for the system so that all contracts have equal access to the data. Since the data model is used by all contracts we need to ensure it is accessible. The data model itself may be some easily accessible object, which contains the pre-loaded data and configuration information. All contracts have equal access to this data, there are no special allowances for Contract X to have special access.

Separating the state out also makes it clear what is data, and what is algorithms. This simplifies reasoning about the system.

Dynamically loaded .DLL/.so

Dynamic link libraries, shared libraries in Unix speak, are loaded by an application at run time, often we are only interested in them as libraries not their dynamic properties. However, most OSs allow you to explicitly specify the filename of a library you wish to load and, once loaded, use the functions contained within – this provides a powerful extension mechanism.

You can write several DLLs, each with a common set of functions, and decide which one to load and use at run time, thus you can extend the program at run time.

However, this comes at a cost. Firstly, you must take more care with your version management. Instead of having one large executable to manage you now have several discrete libraries.

Secondly, you must add configuration details to your system so it knows which DLLs to load.

Finally, there are portability problems. On Solaris the action of loading a DLL places all symbols in the run-time symbol table so extra care is required to ensure you don't call a function with the same name in another DLL, while Microsoft traditionally provide a stub library to link against.

If we wish to load a DLL and call a function by name the process is actually quite similar. On Windows we use LoadLibrary and GetProcAddress, while on Unix we use dlload and dlsym to the same effect.

COM & CORBA

Both COM and CORBA can be used to for extensible systems. However, the literature on both emphasises different aspects of each system. Essentially, both implement the loading of dynamic libraries.

If your system already uses, or you plan to use either COM or CORBA you can take full advantage to make your program extensible. However, if you are only interested in their extensibility properties I would advise against using either of them. There are simpler techniques (some outlined here) which provide the same benefit without the cost.

When I say cost I'm not talking about monetary cost – although simply buying the literature on either product is expensive, and purchasing a brand name ORB is not cheap – rather I am thinking of:

? both have steep learning curves : even if you know COM think of the maintenance requirements

? both have a reputation for poor performance

? both force you to design your system around them

? both have reference counting problems

? both force you to get into IDL writing which may be over kill

? COM locks you into Microsoft systems

? CORBA code can become specific to one vendors ORB if care is not taken

Poor-man's COM

Even without using COM or CORBA you can pass objects out of dynamic libraries you have loaded.

All that is required is three steps:

1. Simply define an abstract base class, e.g.

```
        class Base {
```

```
public:
     virtual bool Action() = 0;
};
```

2. In each of your dynamic DLLs provide a concrete implementation of this base class, e.g.

```
class Concrete : public Base {
public:
     virtual bool Action() { return true; }
};
```

3. In each DLL provide a Factory function which returns pointer to your Base class, e.g.

```
Base* Factory() { return new Concrete; }
```

You can now write as many objects as you like, each packaged inside a DLL and choose which to load at run time.

Of course, should you decide to change the interface on the Base class you will need to recompile everything in your system. This is, equally true if you change the IDL interface on a COM or CORBA class.

Exception handling

It may not be obvious at first that exception handling has a part to play in writing extensible code but it does, a very important part.

In the days before exception handling we typically had one file with a large number of error codes in it, such as ErrorCodes.h[3]. Whenever a new error was added ErrorCodes.h needed updating and, since every file in the system depended on ErrorCodes.h, the entire system needed re-compiling.

By defining a hierarchy of exception classes derived from a simple base we can allocate error codes and messages as needed. We may still wish to provide each object with its sub-system code.

When a simple error code is passed up the call stack it is difficult for the top-level code to take any special action without knowing specifics about the circumstances. Contrast this with an exception class, which can itself provide specific methods for such a circumstance.

For example:

```
class ContractEvaluatorException : public std::exception {
     public:
     virtual int SubSystem() = 0;
     virtual int SpecificCode() = 0;
     virtual const char* ExtendedDescription() = 0;
     virtual bool EvaluationComplete() = 0;
```

[3] On a side note I urge everyone to avoid using the word “error” in filenames. Grepping a long compiler long for errors is much easier if there are no false positives.

```
          virtual void StoreEvaluation() = 0;
          virtual bool IsFatal() = 0;
     };
     ...
     int EvaluateAll (      std::list<Contract> contracts,
                            DataModel& dataModel) {
          for(int i=0; i<=contracts.size(); i++) {
               try {
                    contracts[i].Evaluate(dataM odel);
               }  // try
               catch (ContractEvaluatorException& exp) {
                    cerr << exp.what()
                         << " in subsystem " << exp.SubSystem()
                         << " code = " << exp.SpecificCode()
                         << ": " << exp.ExtendedDescription();
                    LogError(exp);
                    if (exp.IsFatal()) throw;
                    if (exp.EvaluationComplete()) {
                         exp.StoreEvaluation();
                    }  // if
               } // catch
          } // for
    } // EvaluateAll
```

The higher levels of the program are still ambivalent to what was actually happening – beyond the fact that some contract was being evaluated. Again, the exception system has allowed us to separate the cause from the effect (see Kelly, 2000) - this is dependency inversion at work.

State machines

State machines are particularly good at absorbing extra code. The simplest state machines (a big switch statement and a whole set of functions) can have extra states added with little pain but beware, beyond a certain point the big-switch statement becomes a pain to maintain.
More advanced state machines can be completely reconfigured at run time and may use look up tables rather than hard coded settings. Equally, I have read several articles on object based state machines over the years.

One of my favourite features of state machines is that they are very easy to debug and to explain to users. You can sit down with a piece of paper and trace the route against a diagnostic printout, or with a user who wants a change.

Putting it all together

This article draws heavily on my own experience. In these kind of extendable systems I frequently find a large number of "action objects." These are C++ classes which exist for one purpose only, indeed, as in the example above they may have just one significant method called *Action().*

To keep these objects decoupled they are usually passed a means of accessing the program state when they are *action'ed.* These objects are ideal candidates for being placed in a queue and processed sequentially. Sometimes the processing order is important, sometimes the processing could be farmed out to worker threads to allow several objects to be *action'ed* at the same time.

Another characteristic is that the actioning of the objects may further populate the queue of objects to be action'ed. Sometimes this is direct, the action method will actually add a new item to the processing queue, other times it is indirect, the action method will trigger some other process which results in the queue being populated.

In fact, what I have just described is the *Command* pattern in a slight disguise.

Example code

By the time this article appears I should have some example code available on my web-site – www.allankelly.net. This demonstrates the use of dependency inversion to allow extensions to the code and poor-man's COM system. At the moment the code compiles on Windows 2000 using either Visual C++ or GCC. Overtime I would like to extend this code in several directions.

Summary

Extensibility is a worthy design goal. It is the goal of many design patterns and development techniques but it is seldom stated explicitly. Once we recognise extensibility as an explicit aim it is not rocket science. There are a variety of mechanisms for implementing it and with a little practice it becomes easy.

Of course even the most extensible systems suffer from sods-law - change requests can always occur for items your didn't expect to need changing – the classic *outside-context-problem*[4].

Bibliography

Banks, Iain. 1996; Excession, Orbit, 1996

Bruntlett, Ian 2000; "User Defined Types: Qualities, Principles & Archetypes", Overload 39, September 2000.

[4] From Iain Bank's novel "Excession" (1996) - although as far as I know Ian Bruntlett (2000) was the first to use *Outside Context Problem* in connection with software.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995; Design Patterms, Addison Wesley 1995 - also called the *Gang of Four* book or *GoF* for short.
Griffiths, Alan; “Dependency Inversion”, www.octopull.demon.co.uk/c++/dependency_inversion.html
Kelly, Allan 2000: “Error Handling and Logging”, Overload 35, January 2000
Martin, R.C. 1996; “Liskov Substitution Principle”, C++ Report 1996, www.objectmentor.com/resources/articles/lsp.pdf