The Philosophy of extensible software

In Overload 49 I wrote about extensible software, it's a theme I'm going to continue with for a couple more articles. If I were to attempt to summarise my philosophy of software development in one sentence it would probably be: Software must stay soft, malleable. The discipline of extensibility is the tool which best helps us achieve this. So, although I'm declaring an intention to stick with software extendibility for a little while, I'm actually intending to look at how we can keep our software flexible and open to change.

How does software resist change?

In order to understand how we can keep our software malleable it is worth considering how software looses this quality. After all, when you start a new project, the world is your oyster, you can take the project in any direction.

Ripple effect

A change in one module is just that: a change in one module of the system. The principles of abstraction and data hiding tell us that we should be able to make changes to hidden code which have no side-effects elsewhere. Often though we find there is a *ripple effect*.

If we drop a small stone into a pond we see ripples spread out across the pond. The water surface is perfect for propagating the effect. Software is, if anything, better than water at propagating the effect. Changing the interface of a module means we must make corresponding changes to the use of the interface elsewhere.

Speaking of *the interface* means more than just the class definition in some header file. This is the most clearly stated part of the interface but it is like an iceberg, there is much we can't see. The compiler can check the function signatures but can it check the comments?

Consider some code:

```
// SerialPort.hpp
// Read up to bufferSize characters from serial port
// store in buffer and return ptr to buffer
char* ReadSerialPort(char *buffer, int bufferSize);
...

// SerialPort.cpp
// Read at least bufferSize chars from serial port
// store in buffer and return ptr to end of buffer
char* ReadSerialPort(char *buffer, int bufferSize)
{
     assert(buffer != 0)
```

```
assert(bufferSize > 16)
...
```

The interface the compiler can check only forms part of the interface. The comments form another vital part of the interface and in this case they differ. Which set is correct? The developer must break the abstraction and look *under the hood* to see what is happening.

This is typically how a ripple starts, we've found something which isn't quite right, not so wrong it breaks the program but not good either.

Both sets of comments fail to tell us that the buffer supplied must be pre-allocated and at least 16 characters long. Sure, it may seem logical to allocate the buffer before calling the function but the C function `time` never worried too much about this.

At one level this is implementation detail, at another level it is interface, we can fix the comments, we can even change the function signature to reduce the problems with this function:

```
// SerialPort.hpp
// Read up to bufferSize characters from the buffer
// store in buffer and return number of bytes read
// bufferSize must be at least 16 bytes
// return value < bufferSize
int ReadSerialPort(char *buffer,  int bufferSize);
```

Now we have created a ripple effect, not only this module but several others will need recompiling. Wherever this function is used we must now change the code, our change has slipped out of our chunk. While it would be a pretty relaxed compiler that still allowed you to write:

```
char* buf = ReadSerialPort(buffer, 32);
```

A developer in search of a quick fix may be tempted to write:

```
char* buf = (char*) ReadSerialPort(buffer, 32);
```

While `static_cast` will refuse this `reinterpret_cast` has no such qualms, and as demonstrated the old-style casts are still in the language and available.

The general rule of ripple effect is that *he* who made the ripple has to stop it, you find yourself running around all over the code, fixing ripples where they appear. This is all but impossible if you don't have a reliable build process - if you can't integrate your code easily then you can't tackle these issues. A good source code control system is essential in case things go wrong, or time runs out, and you need to back out your changes.

Nor do we have perfect foresight, we may grep the code for every instance of ReadSerialPort before we make our change, but we can't expect to find every case. Just searching the code may be a bigger job than actually performing the change. A logical directory structure is important here, if our code is scattered over several dozen disparate directories on different hard discs then what chance have we got of finding it?

Suppose we now find:

```
// pointer to general read function
```

```
typedef char* (*ReadPortFunc)(char*, int);


// read function
char* ReadData(ReadPortFunc reader)
{
     int bufferSize = 256;
     char* buffer = new char[bufferSize];
     memset(buffer, 0, bufferSize);
     return reader(buffer, bufferSize);
}
```

Instead of diminishing, our ripple has grown. We can fix this, but suppose we find:

```
char* ReadUsbPort(char *buffer, int bufferSize);
char* ReadKeyboardPort(char *buffer, int bufferSize);
```

Do we fix these functions too? Or fudge it? The ripple is not just a simple compile time fix now, it has uncovered a bigger problem, and while we may have the code in a compilable state, we (should) feel a certain moral commitment to fixing this problem. The ripple has grown.

Ripples like this, and fear of ripples, is one of the main ways code resists change.

Friction of change

The ripple effect demonstrates the friction that can occur when changes are made. If these changes are within the same module then the friction is less because the changes are not visible elsewhere. The bigger the change, the more modules involved, the greater the friction. When a system is changing rapidly, dividing it up can be counter productive because there is a constant friction as changes ripple out of one module and into others.

But friction between modules comes in other forms too. Where there are several developers on a project there is always the opportunity for conflicting changes to happen. While exclusive locking through source code control can help, it is not a complete answer. At best it forces one developer to wait while another completes a change, the second developer then has the task of integrating the change with their requirements. Non-exclusive locking systems can hide this problem until the second developer checks their code but the same problem arises.

Either way, friction is generated because two developers must co-ordinate their actions. If the developers are located in different teams, or even different countries, the friction is much greater still.

When a change introduces a new dependency into a module, say a new file must be `#include`'d the initial friction may be small, a slightly increased build time. But when this changes the overall dependencies of the module, and in particular if this introduces a circular dependency the potential friction is greater still.

Observant readers may have noted the potential contradiction is talking of "ripple effect" and "friction" – after all ripples occur in frictionless water. Ripples are waves, and waves can only occur when two

modules share a common boundary. Such a boundary propagates the wave – think of the way an earthquake wave carries.
Sound waves cannot travel in a vacuum, likewise software ripples cannot pass from one module to another if they are well spaced. Since our modules don't exist in isolation we can't place them in a vacuum, what we can do is try to minimise the friction at the boundary and thus minimise wave propagation by allowing each module to change without creating a wave beyond its own boundaries.

Process roadblocks

Software takes on many of the attributes of the organisation and process that creates it.
This idea is summarised as *Conway's law* – although the exact wording of the law differs. Jim Coplien's process pattern of the same name has the solution "Make sure the organisation is compatible with the product architecture."
Where an organisation is conservative and resists change their software will too. This may manifest itself in many ways: a business which resists change may create code which resists change, or, it may mean managers refuse to allocate time for modifications.
This can be a frustrating position for a software developer, they may know of a bug, they may know how to fix it but they may be refused permission to deal with it. Or, to fix it may require raising a bug report, having the work prioritised, authorised, scheduled, changed, tested, signed-off and released.
Sure we need a process, but we must not put the process before the product. In process-centric organisations we find mangers who know the price of everything but the cost of nothing.
Some organisations refuse to recognise refactoring as an exercise. "If it processes data, it can't be broken, can it?" "Reworking something means you made a mistake, right?"
Developers have refactored code since the beginnings of time, but only with the publication of Martin Fowler's book (2000) has it been a respectable activity. Unfortunately, it is still not an acceptable activity in many organisations. Failure to refactor code makes it more rigid, as we put change upon change it becomes inflexible and set in its way. Unfortunately it still processes data.
This is like not servicing a car, it continues working, there is no apparent problem, but the further you get beyond an oil change the more damage is being done to the internals. Like a car, over time software changes and without active attempts to improve the quality it invariably deteriorates.

Development is a learning process

As we develop software we learn, we learn more about the problem domain that the software addresses and we learn more about our solution domain – the tools we have used to address it. Naturally, this leads to new insights into both.
We also have time to dwell on problems and issues. We may take a week to draw up a class hierarchy, but we have the rest of our lives to rethink it and consider how we could of done it better. This can make life hard for us if we come to believe we made a mistake, or no longer agree with our original designs – or just see a better way. Maybe what once seemed a brilliant design now seems top heavy, or inefficient, or simplistic. Don't be too hard on yourself, admit you made mistakes if necessary. If we don't do this we will not move forward, it is now us who are resisting change.

How does extensibility work?

Extensibility works because it forces an approach to problems based on:

? An up front design which allows for addition

This is not to make a case for big up front design - quite the opposite in fact. Big up front design assumes you can design the entire system up front. An extensible design accepts you can't design everything in advance, instead it provides a light framework which can allow for changes.

In some ways this is similar to the STL separation of container and algorithm. The STL doesn't claim to know all the algorithms that may be used with a container, but instead provides a mechanism (iterators) which allow algorithms to be added later.

? Additions to be made in small, incremental steps

It is possible to produce an extensible system where the increments are big, take our command pattern example. The commands could be small, "Put the kettle on", rather than big: "Take over the world." If we make our commands too big we loose the element of extensibility, the original problem is relocated inside a single command, which is effectively the entire system.

? Work elements to be separated into comprehensible units

Computers may run programs and source code may be compiled by a tool, but it is humans who have to read and understand the system. There is a human factor to all of this, just because we can write an immensely complex piece of code doesn't mean we should. Anyone who has tried to maintain by hand code that was originally produced by a code generator will have seen this problem – indeed Perl scripts exhibit the same problem at times. So, keep each unit at a human level.

How does extensibility help?

To achieve these objectives we need to emphasis traditional software development issues: high cohesion, low coupling, interface-implementation separatation, minimise dependencies, and develop build procedures to perform constant integration. This imposes a discipline on our development. Extensible design fits well with the principles advocated by the Agile methodologies and iterative development. It allows functionality to be implemented in small steps as required, thus it dove-tails with the minimal implementation, iterative development and frequent re-prioritisation often advocated by Agile development.

In an extensible design we cannot afford for one chunk to be too closely coupled with other chunks. The very essence of the system is embracing change, it is accepted that additions will be continual, if one chunk of the system resists such change it will make the whole design unworkable. Thus, we have placed the friction of change centre stage. Normally we would rather not think about friction, it is a problem we want to go away. By elevating the issue we are directly addressing it, the whole system is designed around the idea of change through addition.

If you are the kind of person who likes new, green-field, system development this may sound pretty horrid. Basically, I'm suggesting you lay minimal foundations of specification, design and framework coding and make a quick dash for the maintenance phase where you actually fit the functionality.

True, I hold my hands up, I agree. However, in my defence, I claim I'm actually moving as much new development as possible into the maintenance phase of the project. Extensible software allows you to write new code well after your first release. Indeed, if you find a chunk of functionality is difficult to understand, buggy, or just not extensible throw it out and start again.

What is important is to get an up front design which can allow for continued development. This is like a shipyard building the hull and inner structure of a ship but leaving the fitting out and completion of the super-structure until after launching. Once the ship has enough structure to float it no longer need to monopolise a slipway - indeed it may even be fitted out by a different yard. Over the course of its life it will undergo continual maintenance even as it plies the high seas with the occasional refit, which may completely change its use.

Extensibility is not "reuse"

Extensibility a no magic bullet, it is just another technique in our toolbox for tackling software development. Nor is it a code word for "reuse". True, many of the properties emphasised by extensibility are the same ones preached for reusable code: low coupling, high cohesion, modularity, but these properties are advocated by most software engineering themes. Indeed, who would argue for tightly coupled systems?

It may be that, having an extensible system, with malleable code allows your technology to be transferred to another project – many of the properties required of an extensible system make transfer easier. One could easily imagine a word processor system which offered a standard system and a *beginner* version with fewer options, plus a *professional* version with more – the same way Volkswagen sell the Golf in tandem with the Skoda Fabia (low end), Audi A3 (high end) and specialisations like the Beetle and TT.

But, such platform transfer is deriving from the minimalist camp - "less is more" is the starting point. Extensible software development is no license to add bells and whistles to your code in the hope that someone may use them. Quite the opposite, extendable software should be free of bells and whistles, it should be minimal while allowing itself to be extended.

Striving for extensibility should impose a discipline on development leading to fewer, cleaner, dependencies, well defined interfaces and abstractions with corresponding reduction in coupling and higher cohesion.

I've been here before....

I tried at the top of this essay to summarise my software development philosophy. Looking back at my contributions to Overload in the last few years I can see this as a common theme. To keep software malleable we must be aware of the dependency structure of the program, this I addressed in Overload 41 when I wrote about layering in software; dependencies start with include files (Overload 39 and 40). I believe inline functions reduce abstraction, increase dependencies and generally complicate matters – hence my piece in Overload 42. (If anyone ever produces a subset of C++ I'd lobby for inline functions to be first against the wall.). More recently my pieces have looked at how we view software as models (Overload 46) or abstractions (Overload 47).

Extensibility of software happens in all sorts of ways, at different levels within the system. It is important to have a view of your software as a living, growing, entity.

And finally

Extensibility is a technique for reasoning about our software. It is not new but it has been neglected as a technique in its own right. In part this is because it is often an attribute of other techniques – as noted in my previous essay it is implicit in many design patterns.

The properties that make up an extensible system are not confined to your source code – there are build systems, source code control, bug tracking, documentation, team mangement, and more. (I tend to call this the *logistics tail* and I'll expand on that idea next time.)

Extensible source code must be supported with extensible build systems, directory trees, database access mechanisms, and so on. These systems shine when we are able to align design, source code, management and developers to form a process which becomes a reinforcing strategy.

Bibliography

Conway's Law comes in several different forms, Ward Cunningham's Wiki page gives several different forms at http://c2.com/cgi/wiki?ConwaysLaw

Jim Coplien version of Conway's law as a process pattern is at http://www.bell-labs.com/user/cope/Patterns/Process/section15.html

Fowler, M., 2000; Refactoring – Improving the design of existing code, Addison Wesley, 2000