Extendable software and the bigger picture

My last two Overload contributions have described my philosophy of extendable software, why it is important, and how we can implement it in the code we write. But that isn't the end of the story, in fact that is just the start.

To create extendable systems you need space – space to write the extensions in, space to practice your art, space to think. If your cramped into a small directory tree on your disc, or squeezed by working procedures, or forever fixing the build you simply don't have the space you need. To this end extendable software is as much about process as it is about code, and this is where things get really interesting.

Extendable software naturally fits into an Agile development methodologies. I've started from the code level and I'm working up, saying, "What do I need to do to support this style of code with my processes?" The Agile methodologists have started from process and are working towards code.

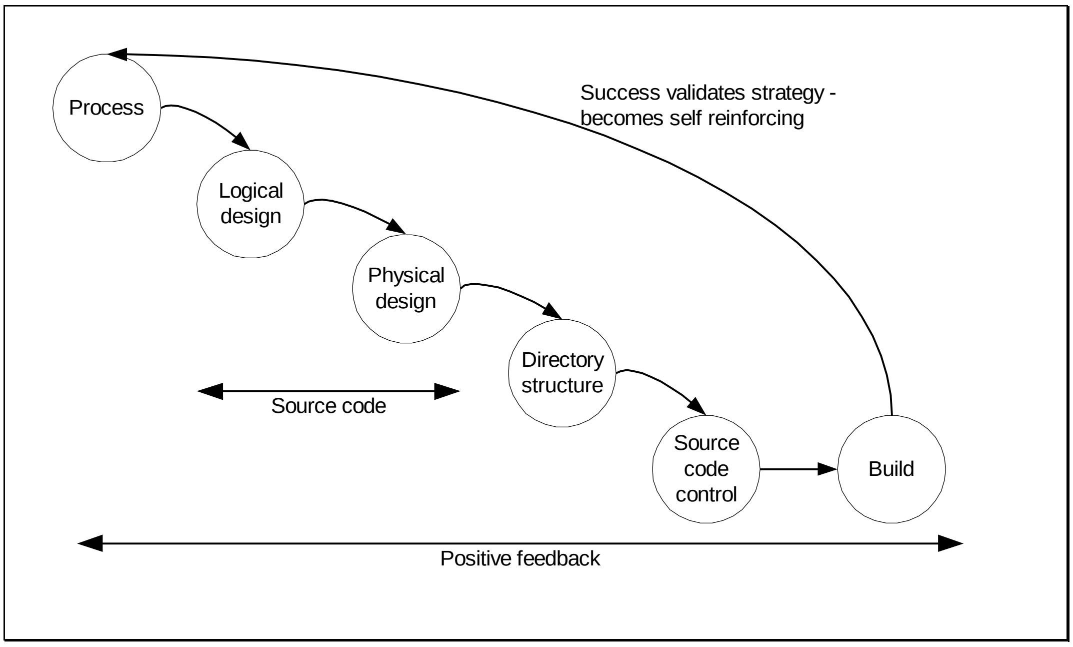
Software has a logistics tail (see side box). We ignore this tail at our peril, if we are to keep advancing our development we must pay attention to the build system, source code control, etc., etc.. If not, this tail will eventually wag the dog - you'll be so busy getting things to build you won't have time to fix the faults.

This article and the ones that follow are concerned with aspects of that tail which I think are neglected: directory structures, source code control, build systems and how these build into a process.

Looking to the bigger picture

Embracing extensibility in your system architecture is only the start of the story. It is worthless if you do not provide the support services needed. So, a development process that embraces extensibility will enhance the value of the source code which represents the architecture. Both the logical and the physical architecture must be aligned to this. Aligning the physical architecture means a coherent directory structure, which will enhance the value of source code control, which in turn enhances the build system.

This will remove uncertainly, thus improving repeatability and contribute to a successful product. A successful product will validate your process and architecture; we have a positive feed-back loop.



Our process influences the logical design of a system (Conway's law), which obviously effects the physical design and coding, this in turn effects the directory structure we use, which itself influences our source code control, and finally we get to build our product.

And none of this happens in isolation, actions feed backwards too, if you are hobbled by an ineffective source code control system you will find your design warps. All of these items interlock.

For example, one of the tenets of Extreme Programming (XP) is *Continuous Integration*, how can you hope to continuous integrate if you don't have a well-defined build process?

Too often we have been sold magic bullets, a development process that doesn't define an integration approach, or a source code control system that doesn't fit our process.

Enough!

In the name of extendable software we have to look at the bigger picture.

Strategic fit

What we are actually talking about here is not a just process – itself a much overloaded word in software – but a strategy which will fit the various pieces of development tightly together and interlock them, and lock software development into the company. It may come as a surprise to some software developers but we are not alone here, discussing business strategy Michael E. Porter of Harvard Business School, writes:

> "What is Southwest's [Airlines] core competence? Its key success factors? The correct answer is that everything matters. Southwest's strategy involves a whole system of activities, not a collection of parts. Its competitive advantage comes from the way its activities fit and reinforce one another.
>
> Fit locks out imitators by creating a chain that is as strong as its *strongest* link." Porter, 1996

Of course Southwest does have imitators, just look at the Ryan Air and EasyJet in Europe, but in its home market Southwest is *the* success story of the American airline industry since deregulation.

To isolate one element of the development process, say writing a specification, and divorce it from the process as a whole is wrong. Each activity needs to be consistent with the other activities, Porter calls this "first order fit", he goes on to define second order fit as "activities that are reinforcing" and finally, third order fit as "optimisation of effort."

We can't expect to write extendable software in isolation, in fact, we can't expect to write any software in isolation. Nor can we expect to look at software development isolated from the rest of the organisation.

Process must accept addition

Our aim in an extendable system is to allow new features and fixes to be implemented in new code. New code may contain its own mistakes but the chances of introducing a fault into existing code are reduced substantially. Lets dwell on that for a moment: how many occasions have you fixed some code only to find a new problem has been introduced in the same code? Would that fault have been introduced if the change had been implemented in fresh code? Yes, the fresh code may have its own faults but we expect new code to have faults and hence test for it.

So, if we are to enhance our system through addition, our development process must accept addition too.

There are two places addition may take place: in existing files or in new files. The former is often necessary, say, to add a new method for an existing class, the later usually implies we are adding new classes, we are working with our architecture.

However, our development process and environment can make these additions difficult. Simplistic management using lines-of-code as a metric can discourage developers from making additions. This is short sighted management, code maintained under such a regime sprouts control flags as functions are coerced to do double duty, what should be two functions, or even two classes, gets implemented as a single function:

```
void CalculateInterest(int accountNumber, bool isDeposit) {

    if (isDeposit) {

        // .... do deposit interest rate calculation
```

```
                }
                else {
                        // .... do current account calculation
                }
        }
```

It doesn't take much foresight to see what happens when we get a third type of account. Almost as obvious is the question: should this be a class hierarchy?

The problem here is short sighted management who are have either failed to realise the relationship between process and code, or, managers who are actively trying to architect the system themselves. True, adding the control flag was a cheap way of getting the functionality quickly – and hence improving profits – but this has introduced future cost to the system. Unfortunately, anyone who regards lines-of-code as a good software metric probably isn't going to be persuaded by such arguments.

New features, New files

The creation of new files should be a natural part of the software process but it doesn't always feel this way. One well-known software house I know is proud of its ISO-9001 procedures that include a paper audit trail for a system. Adding news files to the system required a form to be filled in and signed off. This was a sure fire way to ensure file that do exist grow unnaturally large – even if there where only a dozen or so files.

Similar things happen if we deny developers access to source code control or don't automate the build process. Buying each developer a license for a top quality source code control system may seem a little pointless, after all they only use it for a few minutes each day, why not buy one license? Maybe, to keep our paper trail accurate Fred can run the source code control, and we can request files by e-mail, or signed form, and he can get them out and e-mail them back? Or maybe just place they on a shared drive?

This may all sound like a sick joke but I've seen it done.

It is human nature to do the easy things and avoid the hard things. If we want our system to grown in an orderly manor we can't put obstacles in the way of what we want to happen. Architecture designed for extendibility will fail if we don't align our process. It must be easy to integrate new code.

Paper trails and proxy results

One of the keys to alignment is to automate as much of the possible, thus making it easy to do the right things. If you really must have a paper trail for all file changes then configure your source code control system to produce the necessary documents. If you must have sign-off before code enters the system, then use a promotion model for your files. If your ISO-9001 procedures are getting in the way then change them.

In the worst case the process documentation says one thing and developers do it another way then fake the paper trail, sometimes with management connivance and sometimes without their knowledge. Once you've crossed this Rubicon code quality and communication will rapidly deteriorate.

At the end of the day we want to deliver software, this means writing code. It does not mean writing documents or conforming to lines-of-code estimates. A process that emphasises such criteria is targeting a proxy result rather than the end product. Not only is the proxy the wrong measurement but it distracts us from our main goal and is subject to manipulation.

Monitoring a proxy variables can have a place, take file version numbers for instance. If our average file has undergone 10 revisions but one particular file has undergone 30 then it is worth investigating why. Revision number may be a proxy variable but they are by-products of our normal working practice, we don't expend any extra effort to produce the revision number, thus, is less susceptible to manipulation.

Overnight builds *aka* the batch build

One of the corner stones of any development should be an automated batch build process. Having said that all development processes need to be customised I'm saying categorically: create a automated batch build and run it every night. I'm not alone in this.

> "In our global survey we found that 94% of successful companies completed daily or at least weekly builds, whereas the majority of less successful companies did them monthly or less often." Hoch 2000.
>
> "If you build it, it will ship. If you don't, it won't. ... I don't mean 'Build it once and ship it.' I mean 'Build it often and reguarly.' You must get that product visible. Public." McCarthy 1995.

And after all, what is XP's continuous integration but a batch build?

A repeatable build shows you can build your product. But there is more to it than that. It shows that for all the hard work and money thrown at the problem there is some kind of solution, not just strange files on developers PCs. It even shows that there is something that might ship sometime soon.

A regular build also acts as a restraint on developers. You won't check in some half-baked code that will break the build. And if you do it is easy to see who did it. We don't want to get into blame culture here, we just want to encourage everyone to be responsible.

And when something does break the build there is a clear audit trail to find out why. There is usually a good reason and not always because someone messed up. Why wait until you've finished coding to see if all the bits fit together?

The build is also a ritual of software development. Human civilisation is built on rituals, marriage, births, cards for this, cards for that, and so on. Rituals give us anchor points in a changing, uncertain world. A ritual build serves the same use.

The build provides a heartbeat to a project: you come to work, you write code, you check it in, it builds, you start over again. When the project won't build there is usually something wrong.

As McCarthy says, make it public. Let everyone see it works, e-mail the build log to the developers, display the latest build number on a flashing sign in the lobby. Show you are active. It easy for organisations to loose sight of the work of developers, especially if your just a bunch of geeks sitting in Dilbert cubes.

But the build has to be automated. It has to be carried out by machine – free of human interference. Human intervention introduces a random factor. Automating the process proves it is understood, proves it is repeatable. If you can't automate it and repeat it what does that say about the state of your project? Doing a build is repetitive, boring, easily forgotten, time consuming All the things that human's don't like and machines are good at.

So, how do we get there?

We have the pieces of a solution to hand but you are on your own in putting them together. Yes, we have models we can follow, but there are no guarantees that they will work for us, in our environment. Quite the opposite in fact. Alistair Cockburn (2002) says something similar "The level 3 listener [an experienced practitioner] knows that all the published software development techniques are personal and somewhat arbitrary."

In fact, Porter might argue that we must develop our own methodologies! Implementing a given methodology may well improve our operational effectiveness, which in turn improves our ability to delivery value to our organisations...

> "However, it [operational efficiency] is not usually sufficient. Few companies have competed successfully on the basis of operational efficiency over an extended period, and staying ahead of rivals gets harder every day. The most obvious reason for that is the rapid diffusion of best practice. Competitors can quickly imitate management techniques, new technologies, input improvements, and superior ways of meeting customer needs.
>
> The second reason that improved operational effectiveness is insufficient – competitive convergence – is more subtle and insidious. The more benchmarking companies do, the more they look alike." Porter, 1996

So, even if we could successfully implement, to the letter, SSADM, RUP, Yourdon, XP, or any other methodology it would do us no good. Competitors could just copy us. What we need is some unique strategy, methodology, which we have tailored to our business needs.

Contrast Porter's words with those of Goldman cited by Cockburn:

> "Agility is dynamic, context-specific, aggressively change embracing, and growth-oriented. It is not about improving efficiency, cutting costs, or battering down the business hatches to ride out fearsome 'storms.' It is about succeeding and about winning: about succeeding in emerging competitive arenas, and about winning profits, market share, and customers in the very centre of the competitive storms many companies now fear." Goldman, 1997.

So, if your sitting back and saying "I have enough trouble getting software written, never mind worrying about the business" you may want to consider what role your software plays in keeping your company competitive.

Software development is about business. It is about selling products – directly or indirectly – so all the concerns of the business are the concerns of the developers. As more and more business depends on software development for their very existence (selling it or using it) the very process of software development becomes a key business function, and a key business discriminator.

It is easy to see how these kind of arguments relate to shrink-wrapped product software but this accounts for only a small percentage of all software written. Increasingly companies are their software. Many of today's business processes would be impossible without software, the process gives the business its competitive edge, without software to enable to the process there would be nothing.

Where do I go from here?

You probably have most of the pieces you need from experience. My last two articles have set out an approach, an attitude, toward software that, hopefully, adds some more pieces to the jigsaw. There is no shortage of literature on process but I hope you found something new here.

What I think is missing, and where I want to turn my attention next is the nuts-and-bolts of how we organise our source code (directory trees and source code control) and how we get continuous integration (the batch build) to work.

Actually, writing about build systems is something I've wanted to do for a while and is surprisingly difficult. Part of the reason is that it is incredibly difficult to write about in isolation from directory structures and source control. However, I am conscious that I promised ACCU-General a while ago that I'd write something on build systems, hopefully, by the time you read this I'll have late drafts of the next two pieces available at www.allankelly.net/writing.

Bibliography

Beck, Kent 2000: Extreme Programming Explained, Addison-Wesley, 2000.

Cockburn, Alistair 2002: Agile Software Development, Addison-Wesley, 2002.

Goldman, S., Nagle, R., Preiss, K. 1995: Agile Competitors and Virtual Organizations, John Wiley & Sons, 1995

Hoch D.J., Roeding, C.R., Purkert, G., Linder, S.K., 2000: Secrets of Software Success, Harvard Business School Press, 2000.

Mazzucato, Marinana 2002: Strategy for Business, Sage Publication / Open University, 2002.

McCarthy, Jim, 1995: Dynamics of Software Development, Microsoft Press, 1995.

Porter, Michael, E. 1996: "What is Strategy" from the Harvard Business Review, November/December 1996, reprinted in Mazzucato 2002.