The Developers New Work

I read Stefan Heinzmann's piece(s) in Overload 61 with a feeling of déjà vu. Not you understand because I'd wrestled with the same coding problem as him - if you recall he just wanted a store look up tables in ROM. No, the feeling of déjà vu came from the other problem he was wrestling with: What is this monster we have created called C++?

For me this thought is quickly followed by: How can anyone ever expect to master it? And then: How can I expect anyone to ever maintain this code?

Now I always consider the Overload readership to be a pretty savvy bunch. People who are, on the whole, smarter than the average C++ developer, but how many of us could have tackled Stefan's little task without encountering many of the same problems? I'll go out on a limb, I don't think any Overload reader could have tackled that problem and got it right in one sitting. I'll go further, I don't think even Herb Sutter, Bjarne Stroustrup or Andrei Alexandrescu could have done it in one sitting.

Partly that's because Stefan was engaged in a learning exercise. As he developed a little code his ideas became more refined. The compiler forced him to remove every ambiguity from his original idea. Hence, when he found the compiler inadequate or vague he was lost.

The other part of his problem was that he was attempting to use the compiler and language to the full. The is where the real problems set in. This is where my sense of de ja vu came from.

Blast from the past

Once upon a time I wrote an application. I thought it was a good application, it was reliable (mostly), was fast, performed its job and was easily understood through a few design patterns and employed thoroughly modern C++ and standard library. Then it came time to leave the company.

The company wasn't a bad company so they selected a developer to take over my work. He'd been with the company a few years but had mostly done Visual Basic work. So they company sent him on a course to learn C++. And then I tried to hand over to him.

Out of that experience came a short little piece of angst entitled *High Church C++* - for reasons which I don't recall it never made it into the pages of Overload but has been a popular download from my web-site (Kelly 2000). In *High Church C++* I wrestled with my conscious, was it right to write a program in a style which was endorsed by an elite but foreign to the masses?

In truth, it wasn't necessarily my style of writing, had I written "low church C++" or some "MFC C++" style the code would have been superficially clearer to the novice but the increased length would have added to the complexity. Complexity can be like that, you push it down in one place and it appears somewhere else.

The *Apprentice* pattern (Coplien and Harrison 2004) suggests it can take a year for someone to become proficient in a new system. What is the best way to bring someone up to speed? Is High-Church C++ better than Low-Church C++? Is Java better than C++? What of Visual Basic? C#?

Whatever our language, whatever our choice of idioms, patterns and style the same problem exists. Someone ends up wrestling with the code base to understand it.

Déjà vu all over again

Contemporary C++ with heavy use of templates, standard library and even exceptions adds a twist because it seems C++ has become two different languages. What I called High-Church C++ and Low-Church C++ might also be called Modern C++[1] and Classic C++. Whatever we call them there seems to be a disconnect between the two schools. (If I recall correctly Kevlin Henney describes *three ages of C++*.)

So it was I had the same déjà vu again a few weeks ago. My office book group has been looking at Herb Sutter's Exceptional C++ (Sutter 1999). In our discussion on exception handling several people suggested that the nuances and intricacies of exception handling meant it was too complicated to use. I think they have a valid point, it is too complicated to use: the stack unwind, the need to avoid resources leaks, the care and attention needed to every line - for heavens sake, it took the brightest C++ minds nearly 10 years to answer Cargill's stack problem (Cargill 1994).

But what's the alternative?

To my mind the alternative to C++ style exception handling is worse. Its so much worse in fact that people don't do it. They simply ignore error handling[2]. C++ style exception handling forces you to face up to resource management, error handling and the like, but it does it at a cost. Get it wrong and the result can be awful.

The alternative isn't the alternative people think it is. The alternative, lets call it "return codes," suffer from most of the same problems but its easier to hide from the problem and pretend your doing it right.

Its the complexity problem again. We push down complexity in one place by establishing syntax and conventions in the language to support good error handling but more complexity arises in getting developers to understand and use the conventions.

C++ is not alone

I've been talking about C++ because that's where my personal experience is, that's where Stefan had his problem and that's what most Overload readers program in (I think.) But the problem is not coffined here.

Other languages and technologies have their own problems. Java has inner classes and new generic programming features to name but two, Perl has its "write only syntax", nor should we think only of programming languages, Unix, Linux and NT all have hidden depths.

---

[1] I use the term *Modern C++* in a broader sense then Andrei Alexandrescu's book *Modern C++ Design*, the book is an good example of *Modern C++*.

[2] For several years it was my point of view that the main difference between the programming we learnt in college and that we did in industry was "Error handling omitted to save space" - a popular textbook expression that doesn't cut it in industry.

And to make things worse we can't just specialise in C++ or Linux, we need to know a cross section: Language, OS, databse, development techniques. How can we ever keep up?

Searching for a solution

The angst in *High Church C++* was very real, I was scared. But what was the answer? I've been looking for a solution for five years now.

We could "dumb down" our code, make it really simple. Trouble is, we have real problems and we need real solutions. To tackle the same problem with "low Church code" just moves the complexity from the *context* to an overly verbose code base.

We could just hire real top-gun programmers. This isn't really a solution, once again we're pushing the problem down in one place and seeing it come up in another. Since there aren't that many super-programmers in the world finding them is a problem, keeping them a problem, motivating them is a problem and even if we overcome these problems it's quite likely that within our group of super-programmers we would seem a elite group emerge who.

Hiring a group of super-programmers is in itself an admission of defeat, we're saying: *We don't know how to create productive employees; we're going to poach people from companies who do.* In doing so we move the problem from our code to recruitment.

So, maybe the solution is to get management to invest more in training. But this isn't always the solutions. The mangers I had when I wrote *High Church C++* tried to do the right thing. Is it not reasonable to assume that someone who has been on a C++ course can maintain a system written in C++?

The answer of course is: No, knowing C++ is a requirement for maintaining a C++ based system but it isn't sufficient of itself. One needs to understand the domain the system is in and the system architecture - this is why Coplien and Harrison you need a whole year to come up to speed.

How do we communicate these things? The classical answer is "write it down" but written documentation has its own problems: accuracy, timeliness, readability, and memorability to name a few. In truth, understanding any modern software system is more about tacit knowledge than it is about explicit knowledge.

So what are we to do?

I don't claim I have the only answer, I don't claim I have the best answer, I don't claim my answer even covers all the bases and it certainly isn't original. But after five years of searching I think I have *an answer*, at least its the best answer I know at the moment.

The answer to the question, the great question, the question of *how do you teach someone about a software system...* is... well, you aren't going to like it....

New Work

The developers new work is to help others learn. I don't mean you all rush off and become C++ trainers I mean we all need to work to improve the capabilities of our colleagues and especially the less experienced around us. This isn't about training, it is about learning. Its about redefining what it means to be a software developer.

Implicit here is another role, to lead. When I mix with other ACCU members I find I share an unspoken bond with them. We all believe it is possible to write better software. Exactly how we do this may be up for debate, maybe we should adopt Extreme Programming, or maybe write in Java, or simply write our tests first. These are all good answers, the real question is: how do we get from here to there?

It is no longer enough to just cut-code. Sure you may need to do this too, but if you want to use modern C++ (or modern Java, Python, or what ever) it is your job to lead others in a change. And change doesn't happen without learning. Indeed, learning doesn't really happen if we don't change, we may be able to recite some piece of information but unless we act on it we haven't really leart anything.

So, when it comes to improving your code it it isn't enough to sit your colleagues down and tell them that a template-template function is the thing they need here and expect them to make it so. You've imparted information, you may even have ordered them to do it, but they haven't been led, they haven't leart and they won't have changed - they'll do the same thing all over again.

Simply informing people "This is a better way" doesn't cut it. You can't lecture, you can't tell, you can't enforce conformance. You need to help others find their own way to learn. Helping them find that way goes beyond simply giving them the book, they need to be motivated, people who are told aren't motivated, people who are ordered aren't motivated; motivating people requires leadership.

If you find yourself resorting to a rational argument to persuade someone to do it *your way* you have failed. Your work is to help them produce the rational argument themselves. We aren't abandoning rationality, just recognising that when you tell someone "you are wrong, I am right" it doesn't do much, far better to help then realise a better way.

For many people simply being told "the correct way" a "better way" isn't enough to bring about a change in their actions. It may well demonstrate your intellectual superiority over them but it isn't going to change them. Telling them to "Read the frigging manual" or "Read my document" isn't useful.

Of course, this is easier said than done...

What do I do now?

You need to change your mindset, you are no longer out to prove you know C++ better than anyone else, you are here to lead them in learning. Because you know C++ better than anyone else you are in a position to do this.

The first change is to stop doing something: stop switching people off. Telling people they are wrong, fixing their problems - especially problems they don't realise are problems - is a sure fire way of switching people off. Why learn something or do it the proper way if someone else will always do the job for you?

Next you need to create awareness of the problem. Are your developers really aware that there is a problem with *Singleton* pattern? Of course, you can't just tell them. Well, maybe you can tell them, but it needs to be in an abstract kind of way, a way that will spark their curiosity, help them find the problem themselves.

I can hear some people saying "But the developers I work with just don't care." These are developers who have been switched off. People are learning machines, if people have given up learning about these things then why? Maybe they've been punished in the past for free thinking. Maybe your company rewards conformance, rocking the boat isn't positive.

Of course, you don't want to be seen as a boat rocker do you? Maybe you do, maybe what you value is demonstrating how much better your insights are? If so your attitude hasn't changed. You want to be somewhere between sparking curiosity and enquiry, so, thrown away those Dilbert cartoons.

You need to redefine your own job and your own self-image. Start with yourself as you are the only person you have complete control over. What is stopping you helping others? Recognise the barriers and over come them. Now move on and do the same with your colleagues.

But I don't have the time...

None of us have enough time, but if your spending your time rushing around fixing other peoples problems and policing their actions your going to have even less time. There is never time to start doing something new so it is always the right time.

If you wait the *right* time will never come along, there will never be a day when your project has finished and the new one hasn't begun. And should that day come its probably a sign that you've done something wrong.

Sure it's hard to adopt a new way of working a few days before a project deadline so maybe this isn't actually the right time. But if you are starting a project, or your half way through, and you can't find the time to change you never will.

As the people around you learn the new techniques and grow in confidence you will find you don't need to spend so much time fixing their work and fire fighting.

Again, but what do I do now?

I am sorry dear reader but I have to disappoint you. I have no list of 35 things to do, I have no "Effective Learning" to give you. You have to find your own answers which work for you, your team and your company.

Anyway, I'm not the best person to ask on this subject. More worthy authors than me have examined this question and they are the place to start. This isn't the first Overload article by me to cite Senge's *The Fifth Discipline* (1990) but I don't apologise for suggesting you read it[3].

In dealing with people we are letting go of the Swiss-army knife of rationality, emotions are more important so Goleman's *Emotional Intelligence* (1996) is well worth a read (and particularly so if your a parent too.)

Somewhere along the line you should seek to improve yourself, although its not my favourite book and I don't agree with everything he says Covey's *Seven Habits of Effective People* (1992) is the book to set you thinking about yourself. For those who can't find the time at least absorb his fifth habit: "Seek first to understand, Then to be

[3] Much of this essay is inspired by Senge's chapter entitled "The Leaders New Work."

understood." (Covey also has a solution to the "don't rock the boat" problem but I'll let you read that for yourself.)

If you've made it this far your attitude will already be changing and you'll be looking for new ways of working. After all this reading you'll either by ready to dismiss my ideas or take up the challenge. The next two books are more practical. You might want to try *coaching*, Whitmore (2002) provides a good introduction to the subject. You'll also recognise the need to spread your ideas subtly, this is where Linda Rising's new book is useful. *Fearless Change* (Manns and Rising 2005) is a recipe book for introducing new ideas. Pick a pattern try it, see what happens. Try another.

Maybe I'm ducking the issue. I haven't given you any hard and fast rules. I haven't said "Say X to your developers" but I don't think I really can. You have to first learn your self. Sure I could give you a quick check-list of do's and don'ts but I do not know you, your developers or your environment. The books I've listed here won't give you all the answers but they should help you find your own answers.

We're along way from where we started...

There are no silver bullets. Technical solutions have a habit of becoming technical problems - like Stefan's templates. Or, to put it another way:

> "Today's problems come from yesterday's 'solutions'." (Senge 1990, p.57)

Hope lies not in code, not in machines but in people. If we believe that Modern C++ is best - and I truly, rationally, believe it is - then I have no choice other than to develop the people around me - and that belief is rational too.

So, I'm not giving you any silver bullets but I am telling you where you can hunt for silver. True, it will take you time to find the silver and you'll need some help making the bullets so you have plenty of opportunity to practise leaning and leading.

Bibliography

Cargill, T. (1994). "Exception Handling: A False Sense of Security." C++ Report **6**(9).

Coplien, J. O. and N. B. Harrison (2004). Organizational Patterns of Agile Software Development. Upper Saddle River, NJ, Pearson Prentice Hall.

Covey, S. R. (1992). The seven habits of highly effective people : restoring the character ethic. London, Simon & Schuster.

Goldman, D. (1996). Emotional Intelligence, Bloomsbury.

Goleman, D. (1996). Emotional Intelligence, Bloomsbury.

Kelly, A. (2000, March 2000). "High Church C++." Retrieved 31/October/2004, 2004, from http://www.allankelly.net/writing/WebOnly/HighChurch.htm.

Manns, M. K. and L. Rising (2005). Fearless Change - Patterns for Introducing New Ideas. Boston, MA, Addison Wesley.

Senge, P. (1990). The Fifth Discipline, Random House Books.

Sutter, H. (1999). Exceptional C++, Addison-Wesley.

Whitmore, J. (2002). Coaching for performance GROWing people, performance and purpose. London, Nicholas Brealey.