Back To Basics

Hype-free Principles for Software Developers

Jason Gorman

codemanship

CONTENTS

INTRODUCTION

While the media often refers to the start of the "digital age" when multimedia applications and hypertext took off in the late 1980's, it's worth remembering that people have been programming digital electronic computers since they were invented during the Second World War.

Software development's not quite the lawless and anarchic wild frontier people make it out to be. Developers today have seven decades of practical experience of writing software under commercial pressures to draw on, and there are many insights that have been built up over that time that an aspiring young programmer needs to know.

Changes in our economy, big rises in university fees and high youth unemployment are leading more and more of us to the conclusion that apprenticeships may be the best route into software development careers.

Looking at the apprenticeship schemes on offer at the moment, though, presents us with a problem. They tend to be closely aligned to specific movements in software development, like Agile Software Development or the Software Engineering movement, and even within those movements there can be wide variation of what young developers are learning.

An "Agile apprenticeship" with one company can lead to a remarkably different understanding of software development to an "Agile apprenticeship" with another company. One apprentice might be learning all about the Dynamic Systems Development Method (DSDM) while the other is learning about Extreme Programming (XP). One apprentice might learn all about Scrum for managing projects, another might learn about Lean, and another about Kanban. One might learn to do Test-driven Development; another might learn to do Behaviour-driven Development. And so it goes on.

When I started writing software for a living, I was confused by the scores of different ways of apparently doing the same thing. Since then, the Agile movement has added scores more to this cornucopia.

I had to read dozens and dozens of books and try a whole range of approaches to see through all that smoke and begin to make out the shapes of real insights into software development.

As I read more, I discovered that many of these insights went back to before I was born. It seemed to me as if what we'd mostly been doing these last forty years or more was coming up with new names for software development principles and practices that those in the know were already doing.

That was a journey of over a decade. The mist really didn't clear until about 2002 for me. I felt I wasted a lot of time wading through what turned out to be marketing hype – just a whole sea of buzzwords and brand names – to get at those few important insights and tie it all together in my own head into a coherent whole.

The last decade has been a process of reapplying those insights and projecting them on to the hype, so that when I coach someone in what we're calling "Test-driven Development" these days, for example, I know what it is we're really doing. More importantly, I know *why*.

Don't get me wrong; there's nothing inherently bad about DSDM, XP, Scrum, Lean, Kanban, RUP, Cleanroom or many of the other methodologies on offer. The problem is that when we master development in such specific terms, we can miss the underlying principles that all software development is built on. And when our favourite method falls out of favour with employers, we risk becoming obsolete along with it if we can't adapt these foundational principles to a new fashionable methodology.

We can also very easily end up missing the point. The goal is not to be "Agile", the goal is to be open and responsive to change. The goal is not to be "test-driven", but to drive out a simple, decoupled design and to be able to re-test our software quickly and cheaply.

Why do we do the things we think are good to do?

For the sake of Codemanship's young apprentices, my aim is to strip away the hype and the brand names and point them directly at the underlying principles, based on seven decades of insights, and in some cases, an impressive and growing body of hard data to support their efficacy in real-world software development.

I do this because I’ve hacked through this dense undergrowth of gobbledygook and carved a path for others to follow. My sincerest hope is that these are insights that, once internalized, may last new software developers for their entire careers, regardless of what the “soup du jour” just happens to be.

SUMMARY OF PRINCIPLES

1. Software Should Have Testable Goals
2. Close Customer Involvement Is Key
3. Software Development Is A Learning Process
4. Do The Important Stuff First
5. Communicating Is The Principal Activity
6. Prevention Is (Usually) Cheaper Than Cure
7. Software That Can't Be Put To Use Has No Value
8. Interfaces Are For Communicating
9. Automate The Donkey Work
10. Grow Complex Software Using The Simplest Parts
11. To Learn, We Must Be Open To Change

SOFTWARE SHOULD HAVE TESTABLE GOALS

Why?

No, seriously, though. Why?

Whenever I ask this question to a software team, the response is usually a lot of hand-waving and management-speak and magic beans.

Most teams don't know why they're building the software they're building. Most customers don't know why they're asking them to either.

If I could fix only one thing in software development - as opposed to no things, which is my current best score - it would be that teams should write software for a purpose.

By all means, if it's your time and your money at stake, play to your heart's content. Go on, fill your boots.

But if someone else is picking up the cheque, then I feel we have responsibility to try and give them something genuinely worthwhile for their money.

Failing to understand the problem we're trying to solve is the number one failure in software development. It stands to reason: how can we hope to succeed if we don't even know what the aim of the game is?

It will always be the first thing I test when I'm asked to help a team. What are your goals, and how will you know when you've achieved them (or are getting closer to achieving them)? How will you know you're heading in the right direction? How can one measure progress on a journey to "wherever"?

Teams should not only know what the goals of their software are, but those goals need to be articulated in a way that makes it possible to know unambiguously if those goals are being achieved.

As far as I'm concerned, this is the most important specification, since it describes the customer's actual requirements. Everything else is a decision about how to satisfy those requirements. As such, far too many teams have no idea what their actual requirements are. They just have proposed solutions.

Yes, a use case specification is not a business requirement. Ditto user stories. It's a system design. A wireframe outline of a web application is very obviously a design. Acceptance tests of the Behaviour-driven Development variety are also design details. Anything expressed against system features is a design.

Accepted wisdom when presented with a feature request we don't understand the need for is to ask "why?" In my experience, asking "why?" is a symptom that we've been putting the cart before the horse, and doing things arse-backwards.

We should have started with the why and figured out what features or properties or qualities our software will need to achieve those goals.

Not having the goals clearly articulated has a knock-on effect. Many other ills reported in failed projects seem to stem from the lack of testable goals; most notably, poor reporting of progress.

How can we measure progress if we don't know where we're supposed to be heading? "Hey, Dave, how's that piece of string coming?" "Yep, good. It's getting longer."

But also, when the goals are not really understood, people can have unrealistic expectations about what the software will do for them. Or rather, what they'll be able to do with the software.

There's also the key problem of knowing when we're "done". I absolutely insist that teams measure progress against tested outcomes. If it doesn't pass the tests, it's 0% done. Measuring progress against tasks or effort leads to the Hell of 90% Done, where developers take a year to deliver 90% of the product, and then another 2 years to deliver the remaining 90%. We've all been there.

But even enlightened teams, who measure progress entirely against tested deliverables, are failing to take into account that their testable outcomes are not the actual end goals of the software. We may have delivered 90% of the community video library's features, but will the community who use it actually make the savings on DVD purchases and rentals they're hoping for when the system goes live? Will the newest titles be available soon enough to satisfy our film buffs? Will the users donate the most popular titles, or will it all just be the rubbish they don't want to keep anymore? Will our community video library just be 100 copies of "The Green Lantern"?

It's all too easy for us to get wrapped up in delivering a solution and lose sight of the original problem. Information systems have a life outside of the software we shoehorn into them, and it's a life we need to really get to grips with if we're to have a hope of creating software that "delights".

In the case of our community video library, if there's a worry that users could be unwilling to donate popular titles, we could perhaps redesign the system to allow users to lend their favourite titles for a fixed period, and offer a guarantee that if it's damaged, we'll buy them a new copy. We could also offer inducements, like priority reservations for new titles. All of this might mean our software will have to work differently.

So, Software Development Principle #1 is that software should have testable goals that clearly articulate why we're creating it and how we'll know if those goals are being achieved (or not).

CLOSE CUSTOMER INVOLVEMENT IS KEY

In my humble opinion, when a customer does not make sufficient time to give input or feedback on a software product or system, it's pretty much the worse thing for your whole endeavor.

By "customer", I mean your actual customer; the person with ultimate decision-making responsibility. Not someone delegated to act as a sort of proxy to the decision maker.

The customer is someone who holds the power - the ultimate power - to decide what the money gets spent on, and whether or not they're happy with what the money got spent on.

They shouldn't need to check with a higher authority. If they do, then they're not the customer.

We need to distinguish between customers and people who can often get confused with customers.

A user isn't necessarily a customer, for example. They may be the customer's customer, but they're not our customer.

A business expert isn't necessarily a customer, either. They may have a more in-depth understanding of the problem than the person making the ultimate decisions - that in itself is a red flag for the customer's business, but that's a different story - but if they have to refer back to someone with more authority for decisions to be made, then they're not our customer. They're an adviser to our customer.

Lack of customer involvement is often cited in studies like the CHAOS report [1]as the most common cause of project failure. (Although they, too, confuse customers and users in their report - and they have some pretty backward ideas about what constitutes "success", but by the bye.)

As we'll discuss in the next post, feedback cycles are critical to software development. Arguably the most important feedback cycle is the one that exists between you - the developers - and the customer.

[1] CHAOS report - http://www.projectsmart.co.uk/docs/chaos-report.pdf

If you have just completed a feature and require customer feedback on it before moving on, the sooner you can get that feedback, the sooner you can move on.

If you don't get the feedback quickly, it can be a blocker to development. What teams tend to do is move on anyway. But they're now building on an assumption. Some teams have to wait weeks or months to get that feedback, and that's weeks and months where they're merrily wiring all manner of gubbins into a foundation made of "maybe". If that "maybe" turns out to be a "no, that's not what I meant", or a "it's what I wanted, but now that I see it made flesh, I realise it's not what I need" then - oh, dear...

Software development is a learning process, and feedback's the key to making it work.

What can often happen is that the real customer's far too busy and important to spend any time with you, so they delegate that responsibility to someone who works for them. This is often signified by months or years of things appearing to go very well, as their proxy gives you immediate feedback throughout the process.

It usually hits the rocks when the time finally comes for your software to be put in front of the real customer. That's when we find out that software can't be left to someone else to decide how it should be. It's like paying someone to tell you if you like your new hair style.

Teams can waste a lot of time and a lot of money chasing the pot of gold at the end of the wrong rainbow.

You need to find out who your customer really is, and then strap them to a chair and don't let them go until they've given you meaningful answers.

Here's the thing. Most customers are actually paying for the software with other people's money - shareholders, the public, a community etc. They have a professional responsibility to be a good customer. A good customer takes an active interest in how the money's spent and what they get for that money.

If you're a movie studio executive with ultimate decision-making responsibility for a production, you would ask to see frequent evidence of how the production's going and how the final movie is going to turn out. You may ask to read drafts of the screenplay. You may ask to see videos of casting sessions. You may ask to see daily rushes while it's being shot. And so on. Basically, with other people's money at stake, you'd take nobody's word for it.

It doesn't mean you need to take creative control away from writers, directors, actors, designers and others involved in making the movie. It just means that you are aware of how it's going, aware of what the money's being spent on, and in an informed position to take the ultimate decisions about how the money's spent.

If you want to really improve your chances of succeeding with software, then you need to keep your customers close.

If the customer is unable or unwilling to make that commitment and give you timely feedback and be there when decisions need to be made, then it's probably not going to work, and they're probably going to waste their money. Do yourself and them a favour and can the project. It's obviously not important enough to warrant the risk.

SOFTWARE DEVELOPMENT IS A LEARNING PROCESS

Everything in life is, to some degree, an experiment.

Every song a composer writes is both an object to be enjoyed in itself, and also a step towards writing a better song. And every omelette I cook is both dinner and a step towards cooking a better omelette. But we'll talk about breaking eggs a little later.

With each attempt, we learn and we're able to do it better the next time. This is a fundamental component of our intelligence. We're able to train ourselves to do extraordinary - sometimes seemingly impossible things - by doing them over and over and feeding back in the lessons from each attempt so we can improve on it in the next.

Software's no different. Our first attempt at solving the customer's problem is usually pretty crappy; maybe even as crappy as my first omelette.

When we create and innovate, we're doing things we haven't done before. And when we're doing something for the first time, we're not likely to do it well as we would on the second attempt, or the third, fourth of fifth.

Software development is a process of innovation. By definition, we're doing things we haven't done before on every new product or system. So we must expect our first omelettes to be a bit crappy, no matter how experienced we are as programmers.

Now, I don't know about you, but personally I've got a bit more pride than to make my customer pay for a crappy omelette.

In software development, the way we get to good omelettes is by iterating our designs until they're as good as we can make them with the time and resources available.

Sticking with the culinary metaphor, we cook an omelette. If it tastes good to us (we'll get on to tasting our own dog food soon enough), we get the customer to try a bit. If they love it, then great. We can move on to the next dish they've ordered. But if they don't think it's just right, we seek feedback on how we can improve it. And then we act on that feedback and cook them another omelette.

Rinse and repeat until the customer's happy.

Other books and articles on software development principles often linger on "how to get the design right" and will fixate on all sorts of hifalutin ideas about "user experience" and "modularity" and being "domain-driven".

But of all the design principles I've applied in 3 decades of programming, the most powerful by a country mile is do it again until you get it right (or until you run out of road).

It is nature's way of solving complicated problems, and - to the best of humanity's knowledge - it's the only way that really works.

In practice (let's debase ourselves momentarily to consider the real world), we iterate our designs until they're good enough and we can move on to the next problem. Our customer has some rough idea of what this solution is worth to them, and therefore how much time and resources are worth investing in solving it. We could chase perfection until the end of time, but in reality we find a compromise where the solution we end up is good enough.

And we don't really start from scratch with every iteration, like we would with omelettes. Typically, we take the program code from one iteration and make the necessary changes and tweaks to produce a new, improved version. Unless the first attempt was so off-the-mark that we'd be better off throwing it away and starting again. Which does happen, and why it's highly advisable not to use all your eggs in that first omelette (so to speak).

Many among us believe we should create the simplest software possible initially to start getting that all-important feedback as soon as we can.

The one thing we should never do is assume we can get it right first time. We won't. Nobody ever does.

An important thing to remember is that the shorter the feedback cycles are when we iterate our designs, the faster we tend to learn and the sooner we converge on a good solution.

The problem with complicated things like software, music and omelettes is that it can be very hard to isolate one part of the problem from all the other densely interconnected parts. It may sound like the kick drum is too boomy in the mix, but that could be because the bass guitar is actually too quiet. It may taste like the omelette needs more salt, but that might be because it really needs less strawberries.

As we iterate our designs, when we change too many variables between each new generation it can become very hard to separate the wood from the trees in identifying which of those changes doesn't really work. If we change just one variable and make the omelette worse, we know which variable we shouldn't have messed with and can easily revert back and try something different.

Therefore, another important thing to remember is that this works best when we learn one lesson at a time.

So we learn faster and we learn better when we rapidly iterate and change less things in each iteration.

DO THE IMPORTANT STUFF FIRST

Fire!

Not really. But if there actually was a fire, and you had just seconds to grab a handful of items from your home before running into the winter night in your underpants, would you just grab things randomly?

There's a risk you'll end up in the freezing night air having managed to save only the TV remote and a box of Sugar Puffs. Not quite as useful in that situation as, say, your wallet and your car keys. You'd feel pretty stupid.

So just think how stupid you'd feel if you only had 3 months to create a piece of working software and, when the time and the money ran out, you hadn't got around to incorporating the handful of key features that would make the whole effort worthwhile.

Some features will be of more use and more value to your customer and the end users than others.

Studies like a recent one on menu item usage in Mozilla's Firefox web browser [2]show that some software features are used much more than others. We see this kind of distribution of feature usage on many different kinds of application.

If I was leading a team building a web browser like Firefox, I would want to have "Bookmark this page" working before I worried about "Firefox Help".

When you have a close working relationship with the customer, and unfettered access to representatively typical end users to ask these kinds of questions, it becomes possible to more effectively prioritise and tackle the more important problems sooner and leave the less important problems for later.

[2] http://blog.mozilla.org/metrics/2010/04/23/menu-item-usage-study-the-80-20-rule/

COMMUNICATING IS THE PRINCIPAL ACTIVITY

Hello there.

Yes, I'm talking to you.

You and I are communicating. Communication turns out to be pretty fundamental to software development. In fact, if we sit down and think it through (you are sitting down, right?), communication is actually what software development is all about.

It's possible to explain every activity in software development in terms of languages and communication.

The act of programming is itself a form of communication. As a programmer, you explain things to the computer in a language the computer can understand. Well, more accurately, you explain things to a compiler or an interpreter in a language you can understand - for example, C - and it in turn explains it to the computer in the machine's language.

But when you write a program, the compiler isn't the only target of your communication. Programs have to be read and understood by programmers, too. So, as well as learning how to write programs that make sense to machines, we have to write them so that they make sense to humans, too.

Writing programs that make sense to programmers turns out to be even more challenging than writing programs that make sense to computers. But there is a very good reason why we should make the effort to do so.

Various studies, like one conducted by Bell Labs[3], estimate the amount of time developers spend reading and understanding code at anywhere between 50-80%. Understanding code is actually where we spend most of our time.

We need to understand code so that we can change it, and change it will. When code fails to communicate clearly to programmers, that code becomes difficult to change. And, as we'll see, accommodating change is absolutely vital in software.

[3] http://onlinelibrary.wiley.com/doi/10.1002/bltj.2221/abstract

Apart from writing code that is easy to understand - and therefore to change - there are many other instances where our ability to communicate effectively has an impact on our ability to create good software and deliver value to our customers.

I'm going to focus on two examples that I believe are particularly important: communicating the design within the team, and communicating with our customers.

When developers work together on the same piece of software - or different pieces of connected software - there's a great deal of communication that needs to happen so that teams can coordinate their work into a coherent whole.

Collaborative design requires that developers don't just understand how their piece of the jigsaw works, but how it fits into a larger picture made up of everybody else's pieces. Teams tend to underestimate how much communication is needed to make this possible. Inexperienced teams (as well as experienced teams made up of people who should know better, of course) have a tendency to take their bit and then go off into their own little corner of the room and work in isolation.

This can lead to evils such as unnecessary duplication and software having multiple design styles, making it harder to understand. It's also entirely possible - and I've been unlucky enough to witness this a few times - to find out, when all the pieces are wired together, that the thing, as a whole, doesn't work. (Yet another example of how the high "interconnectedness" of software can bite us if we're not careful.)

It's therefore very important to try to ensure this doesn't happen. Over the years, we've found that various things help in this respect.

Most notably, we've found that it can be a very powerful aid to collaborative design to use pictures to describe our designs, because pictures are well-suited to explaining complicated things succinctly (a picture speaks a thousand words) and can be displayed prominently and drawn on whiteboards, making it easier for the team to build a shared understanding.

It's also a very good idea for teams to integrate their various different pieces very frequently, so we can catch misunderstandings much earlier when they're easier to fix.

Building a shared understanding with our customer is of critical importance. Especially as there's a yawning gap between the hand-wavy, wishy-washy, "it's sort of kind of" language human beings use day-to-day and the precise, computable language of machines that we must ultimately express our customer's requirements in.

Bridging this divide requires us to lead our customers on a journey from hand-wavy to precise. It's not easy. There's a reason why our customers aren't writing the code themselves.

Non-programmers have a problem understanding computable statements, because they lack the ability to think like a computer, on account of having had no experience as programmers. So we can't use the same kinds of computable specifications we'd tend to use among ourselves.

But there is a form of precision that we've discovered programmers and non-programmers are both comfortable with and can both readily understand - examples.

I might give you a vague statement like "a want a tall horse", and to firm up our shared understanding of what we mean by "tall", you could show me some horses and ask "Is this tall enough? Too tall? Too short?"

We can use specific examples to pin down a precise meaning, exploring the boundaries of things like "too tall", "tall enough" and "too short" and building a general model of "tallness" that could be expressed in a precise programming language.

It's also often the case that our customer doesn't know precisely what they mean, either, and so examples can be a powerful tool in their learning process. They might not know exactly what they want, but they might know it when they see it.

There are many other examples of the underlying role communication plays in software development. See how many others you can think of.

PREVENTION IS (USUALLY) CHEAPER THAN CURE

Life is full of examples of how it can pay dividends later when we take more care up front to avoid problems.

Taking a moment to check we've got our keys tends to work out cheaper than paying a locksmith to let us back into our house. Taking a moment to taste the soup before we send it out of the kitchen tends to work out cheaper than refunding the customer who complained about it. Taking a moment to check the tyres on our car before we set off tends to work out much, much cheaper than the potential consequences of not doing so.

In software development, we often find the same thing applies. If we misunderstand a customer's requirement, finding that out and fixing it there and then tends to work out much, much cheaper than discovering the problem in a released product and fixing it then.

We can grossly underestimate how much time - and therefore, how much of the customer's money - we spend fixing avoidable problems.

Decades of studies [4]have shown beyond any reasonable doubt that the effort required to fix errors in software grows exponentially larger the longer they go undiscovered.

A simple misunderstanding about the customer's requirements might cost 100 times as much to fix if it only comes to light after the software has been released to the end users as it would have done had it been spotted while we were pinning down that requirement.

A basic programming error might cost 10-20 times as much to fix after the software's released as it would had the programmer caught it as soon as they'd written that code.

The upshot of all this is that we've discovered that the best way for most teams to write good working software economically is to take more care while they're writing it (see McConnell, Software Quality At Top Speed[5])

4 http://www.research.ibm.com/haifa/projects/verification/gtcb/gtcb_v3_0_2_presentation/index4.htm

Or, to put it more bluntly, teams that take more care tend to go faster.

This is one of life's rare instances of allowing us to have our cake and eat it. We can produce more reliable software (to a point), and it can actually cost us less to produce it.

The trick to this is to look for mistakes sooner. Indeed, to look for them as soon as we possibly can after making the mistakes.

For example, if we think there may be misunderstandings about the customer's requirements, we should test our understanding of those requirements while we're discussing them with the customer.

We looked in a previous post at the potential power of using examples to pin down a shared understanding between us and our customers. When we use examples effectively, we can test that we've really understood what they've asked for.

By beating out a precise, unambiguous and testable understanding of what the customer wants from the software, we not only significantly reduce the potential for requirements misunderstandings, but we also arm ourselves with a powerful tool for checking that we built the thing right.

Those examples can become tests for the software that we can use as we're writing it to make sure it conforms to what the customer expects.

Similar checking can be done by developers at the code level to spot programming errors in functions and modules as the code's being written. You can test a function or a module by itself, or in concert with other functions and modules, to better ensure that every line of code you write is error-free and to catch problems there and then.

A similar effect occurs when teams are working on different - but connected - parts of the software. I may indeed write my part of the software so that it has very few errors in it, but when I wire it up to seemingly error-free code written by someone else, we may discover that the two pieces, when they "speak" to each other, have different expectations about how to work together.

[5] http://www.stevemcconnell.com/articles/art04.htm

In order for software to be correct, not only have all the individual pieces got to be correct by themselves, but all those pieces need to work together correctly.

This is why good teams tend to integrate their individual pieces very frequently, so any slip-ups on the way different pieces interact with each other are caught as early as possible.

This principle of testing throughout development to catch problems as early as possible is entirely compatible with the other principles. (Phew!)

When we treat software development as a learning process, and use short feedback cycles associated with more effective learning, these short cycles give us ample opportunities to test what we're creating (be it software or specifications) sooner. The shorter the feedback cycles, the sooner we can test things.

If we're using examples to pin down customer requirements, these examples can be used to test not only our shared understanding but also the test the software as it's being created to assure ourselves that what we're writing conforms to this shared understanding.

If we test as we code, and break our code down into the smallest, simplest pieces, it becomes possible to work in very short code-and-test cycles of just a few minutes each. This level of focus on the reliability of potentially every individual function (or even every line of code) can lead to software that has very few errors in it by the time it's ready to be put into use.

And studies have shown that such a focus throughout development can allow us to create better software at no extra cost. Hurray for our side!

Finally, to the "(usually)" in the title: we find there are limits to this principle. Yes, it pays to check the tyres on your car before you set off. It can significantly reduce the risk of an accident. But it cannot remove the risk completely. Kicking the tyres to check the air pressure and using a coin to measure the depth of the tread is good enough in most cases.

But if the dangers presented by potential tyre failures are much higher - for example, if that car is about to race in the Grand Prix - you may need to go further in ensuring the safety of those tyres. And in going further, the costs may start to escalate rapidly.

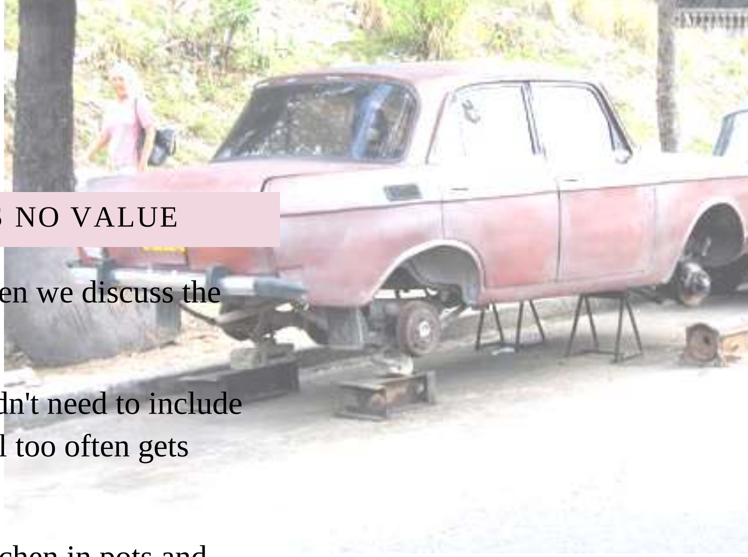
It makes little sense economically to apply the same kinds of checks to your car's tyres before, say, popping to the supermarket that they use on McLaren F1 tyres.

So, yes, you can take too much care...

But here's a warning. Too many teams misjudge where they lie in the grand scheme of care vs. cost. Too many teams mistakenly believe that if they took more care, the cost would go up. (Too many teams think they're McLaren, when, in reality, they are Skoda.)

But, even if we're Skoda, reliability is still almost as important as it is to teams like Mclaren, we may not need to go all of the way, but we probably need to go most of the way. "Perfect" and "good enough" are closer than you think in software.

It's very unlikely that your team is already doing enough to catch mistakes earlier, and that you wouldn't benefit from taking a bit more care. Most of us would benefit from doing more.



SOFTWARE THAT CAN'T BE PUT TO USE HAS NO VALUE

Here's a little ditty that tends to get left out when we discuss the principles of software development.

Arguably, it seems so obvious that we shouldn't need to include it. But an experienced developer will tell you that it all too often gets overlooked.

You cannot eat a meal that's still in the restaurant's kitchen in pots and pans. You cannot listen to a pop song that's sitting on the producer's hard drive in a hodge-podge of audio files and MIDI. You cannot drive a car that's still on the assembly line.

And you cannot use software that's still just a bunch of source code files sitting on different people's computers.

Software, like food, music and cars, costs money to make. In the majority of cases where teams are doing it commercially, that's someone else's money. If our ultimate goal is to give them their money's-worth when we write software, we really need to bear in mind that the value locked in the code we've written cannot be realised until we can give them a working piece of software that they can then put to good use.

Typically, the sooner the customer can get their hands on working software that solves at least part of their problem, the better for them and therefore for us, since we all share that goal.

This has a number of implications for the way we develop software.

Firstly, in order for software to be released to the customer, it needs to work.

Teams that have put insufficient care into checking that the software works - that is, it does what the customer needs it to do, and does it reliably enough - will likely as not find that before their software can be released, they have to go through a time-consuming process of testing it and fixing any problems that stand in the way of making it fit for its purpose.

It's not uncommon for this process of "stabilising" the software to take months of extra intense work. It's also not unheard of for software to fail to stabilise, and for releases to be abandoned. Because problems can take many times longer to fix when we look for them at this late stage, this turns out to be a very expensive way of making the software good enough for use in the real world.

It makes more sense economically to take reasonable steps to ensure that the software is working while we're writing it. We write a little bit, and we test it, and then we write a bit more and test that, and so on. If we're working in teams, we also integrate as we go, to make sure that the whole thing works when we incorporate our parts.

Secondly, it's entirely possible that, if we effectively prioritise the customer's requirements and tackle the most important things first, we may create software that would be of real value if it were put to use before we've built the whole thing.

But who is best placed to decide if that's the case?

It's the customer's problem we're trying solve, and the customer's money we're spending to solve it. Common sense dictates that the customer should decide when to release the software to the end users.

Our job, as developers, is to be ready when they make that call.

We have a responsibility to ensure the software is always working, and that we integrate our work into the whole as often as we can so the risk of important code being left out of the finished product is as small as possible.

In a learning process, for us and our customers, releases are likely to be frequent and the amount of change in each new version of the software is likely to be small. It's therefore of paramount importance that we can release as frequently as the customer requires so that they can more quickly learn about what works and what doesn't in the real world. Failing to do this very significantly reduces our chances of hitting the nail on the head eventually.

INTERFACES ARE FOR COMMUNICATING

Basic Principle #5 states that the principal activity in software development is communicating.

The interfaces we design to allow people - and other software - to use our programs fall under that banner, but I feel they're important enough to warrant their own principle.

An interface provides a means for users to communicate with our software, and through our software, with the computer.

There are different kinds of interface.

Most computer users are familiar with Graphical User Interfaces. These present users with friendly and easy-to-understand visual representations of concepts embodied by the software (like "file", "document", "friend" and so on) and ways to perform actions on these objects that have a well-defined meaning (like "file... open", "document... save" and "friend... send message").

Other kinds of interface include command line interfaces, which allow us to invoke actions by typing in commands, web services which make it possible for one program to issue commands to another over the World Wide Web, and application-specific input/output devices like cash registers used by shops and ATMs used by bank customers.

When we view interfaces as "things users communicate with the software through", it can help us to understand what might distinguish a good interface design from a not-so-good one, if we contemplate some basic rules for effective communication.

Interface design is a wide topic, but let's just cover a few key examples to help illustrate the point.

Firstly, effective communication requires that the parties talking to each other both speak the same language. A Graphical User Interface, for example, defines a visual language made of icons/symbols and gestures that need to mean the same thing to the user and the software. What does that picture of a piece of paper with writing on it mean, and what does it mean when I double-click on it?

An important question when designing interfaces is "whose language should we be speaking?" Should the user be required to learn a language in order to use the software? Or should the software speak the user's language?

Ideally, it's the latter, since the whole point of our software is to enable the user to communicate with the computer. So an interface needs to make sense to the user. We need to strive to understand the user's way of looking at the problem and, wherever possible, reflect that understanding back in the design of our interface.

Interfaces that users find easy to understand and use are said to be *intuitive*.

In reality, some compromise is needed, because it's not really possible yet to construct computer interfaces that behave exactly like the real world. But we can get close enough, usually, and seek to minimise the amount of learning the end users have to do.

Another basic rule is that interfaces need to make it clear what effect a user's actions have had. Expressed in terms of effective communication, interfaces should give the user meaningful feedback on their actions.

It really bugs me, as someone who runs a small business, when I have to deal with people who give misleading feedback or who give no feedback at all when we communicate. I might send someone an important document, and it would be very useful to know that the document's been received and that they're acting on it. Silence is not helpful to me in planning what I should do next. Even less helpful is misleading feedback, like being told "I'll get right on it" when they are, in fact, about to go on holiday for two weeks.

If I delete a file, I like to see that it's been deleted and is no longer in that folder. If I add a friend on a social network, I like to see that they're now in my friends list and that we can see each other's posts and images and wotnot and send private messages. When I don't get this feedback, I worry. I worry my action may not have worked. I worry that the effect it had might be something I didn't intend. Most annoyingly, because I can't see what effect my actions have had, I struggle to learn how to use an interface which is perhaps not entirely intuitive to me.

An interface that gives good immediate feedback is said to be *responsive*. Value responsive interfaces as much as you value responsive people.

Which leads me on to a third basic rule for interface design. Because it's not always possible to make interfaces completely intuitive, and because the effect of an action is not always clear up front, users are likely to make the occasional boo-boo and doing something to their data that they didn't mean to do.

I remember years ago, a team I joined had designed a toolbar for a Windows application where the "Delete" button had a picture of a rabbit on it. Quite naturally, I clicked on the rabbit, thinking "I wonder what this does..."

Oops. Important file gone. In the days before the Recycle Bin, too. The one button they didn't have was the one I really, really needed at that point - Undo!

Interfaces that allow users to undo mistakes are said to be *forgiving*, and making them so can be of enormous benefit to users.

There will be times, of course, when an action can't be undone. Once an email is sent, it's sent. Once a bank payment is made, it's made. Once you've threatened to blow up an airport on a public forum, and so on and etc.

When actions can't be undone, the kindest thing we can do is warn users before they commit to them.

Another way we can protect users is by presenting them only with valid choices. How annoying is it when an ATM prompts you to withdraw £10, £30, and £50, and when you select one of those options you get a message saying "Only multiples of £20 available". Like it's your fault, somehow!

Interface design should clearly communicate what users can do, and whenever possible should not give them the opportunity to try to do things that they shouldn't. For example, a file that's in use can't be deleted. So disable that option in the File menu if a file that's in use is selected.

Similarly, when users input data, we should protect them from inputting data that would cause problems in the software. If the candidate's email address in a job application is going to be used throughout the application process, it had better be a valid email address. If you let them enter "wibble" in that text box, the process is going to fall over at some point.

Interfaces that protect the user from performing invalid actions or inputting invalid data are said to be *strict*. It may sound like a contradiction in terms to suggest that interfaces need to be strict AND forgiving, but it's all a question of context.

If, according to the rules of our software, there's no way of knowing that the user didn't intend to do that, then we need to be forgiving. If the rules say "that's not allowed in these circumstances", then we should be strict.

One final example, going back to this amazingly well-designed GUI with the rabbit Delete button. On the main toolbar, it was a rabbit. But there was also a Delete button on the individual File dialogue, which sported a picture of an exclamation mark. So having figured out once that "rabbit = delete", I had to figure it out again for "exclamation mark = delete". Boo! Hiss! Bad interface doggy - in your basket!

My point is this; in order for us to communicate effectively we must not just be clear, but also *consistent* in our use of language. When we're inconsistent (e.g., "rabbit = exclamation mark = delete"), we significantly increase the amount of learning the user has to do.

When designing interfaces, we should also remember Basic Principle #3 - Software Development Is A Learning Process. It's vanishingly rare to find teams who get it right first time. We should iterate our interface designs frequently, seeking meaningful feedback from end users and the customer and allowing the design to evolve to become as intuitive, responsive, forgiving, strict and consistent as it needs to be to allow users to get the best from our software.

There is, as I said, a whole lot more to interface design than this, but hopefully this gives you some flavour. In particular, we need to remember that good interface design is about effective communication.

AUTOMATE THE DONKEY WORK

Photo: AARDMAN

I don't know about you, but I'm not a big fan of mindless, repetitive tasks.

In software development, we find that there are some activities we end up repeating many times.

Take testing, for example. An averagely complicated piece of software might require us to perform thousands of tests to properly ensure that every line of code is doing what it's supposed to. That can spell weeks of clicking the same buttons, typing in the same data etc., over and over again.

If we only had to test the software once, then it wouldn't be such a problem. Yeah, it'll be a few dull weeks, but when it's over, the champagne corks are popping.

Chances are, though, that it won't be the only time we need to perform those tests. If we make any changes to the software, there's a real chance that features that we tested once and found to be working might have been broken. So when we make changes after the software's been tested once, it will need testing again. Now we're breaking a real sweat!

Some inexperienced teams (and, of course, those experienced teams who should know better) try to solve this problem by preventing changes after the software's been tested.

This is sheer folly, though. By preventing change, we prevent learning. And when we prevent learning, we usually end up preventing ourselves from solving the customer's problems, since software development is a learning process.

The other major drawback to relying on repeated manual testing is that it can take much longer to find out if a mistake has been made. The longer a mistake goes undetected, the more it costs to fix (by orders of magnitude).

A better solution to repeated testing is to write computer programs that execute those tests for us. These could be programs that click buttons and input data like a user would, or programs that call functions inside the software to check the internal logic is correct or that the communication between different pieces of the software is working as we'd expect.

How much testing you should automate depends on a range of factors.

Writing automated test programs that perform user actions tends to be expensive and time-consuming, so you may decide to automate some key user interface tests, and then rely more on automating internal ("unit") tests - which can be cheaper to write and often run much faster - to really put the program through its paces.

If time's tight, you may choose to write more automated tests for parts of the software that present the greatest risk, or have the greatest value to the customer.

Automating tests can require a big investment, but can pay significant dividends throughout the lifetime of the software. Testing that might take days by hand might only take a few minutes if done by a computer program. You could go from testing once every few weeks to testing several times an hour. This can be immensely valuable in a learning process that aims to catch mistakes as early as possible.

Basic Principle #7 states that software that can't be put to use has no value. Here's another obvious truism for you: while software's being tested, we can't be confident that it's fit for use.

Or, to use more colourful language, anyone who releases software before it's been adequately tested is bats**t crazy.

If it takes a long time to test your software, then there'll be long periods when you don't know if the software can be put to use, and if your customer asked you to release it, you'd either have to tell them to wait or you'd release it under protest. (Or just don't tell them it might not work and brace yourself for the fireworks - yep, it happens.)

If we want to put the customer in the driving seat on decisions about when to release the software - and we should - then we need to be able to test the software quickly and cheaply so we can do it very frequently.

Repeating tests isn't the only kind of donkey work we do. Modern software is pretty complicated. Even a "simple" web application can involve multiple parts, written in multiple programming languages, that must be installed in multiple technology environments that each has their own way of doing things.

Imagine, say, a Java web application. To put it into use, we might have to compile a bunch of Java program source files, package up the executable files created by compilation into an archive (like a ZIP file) for deploying to a Java-enabled web server like the Apache Foundation's Tomcat. Along with the machine-ready (well, Java Virtual Machine-ready) executable files, a bunch of other source files need to be deployed, such as HTML templates for web pages, and files that contain important configuration information that the web application needs. It's quite likely that the application will store data in some kind of structured database, too. Making our application ready for use might involve running scripts to set up this database, and if necessary to migrate old data to a new database structure.

This typical set-up would involve a whole sequence of steps when doing it by hand. We'd need to get the latest tested (i.e. working) version of the source files from the team's source code repository. We'd need to compile the code. Then package up all the executable and supporting files and copy them across to the web server (which we might need to stop and restart afterwards.) Then run the database scripts. And then, just to be sure, run some smoke tests - a handful of simple tests just to "kick the tyres", so to speak - to make sure that what we've just deployed actually works.

And if it doesn't work, we need to be able to put everything back just the way it was (and smoke test again to be certain) as quickly as possible.

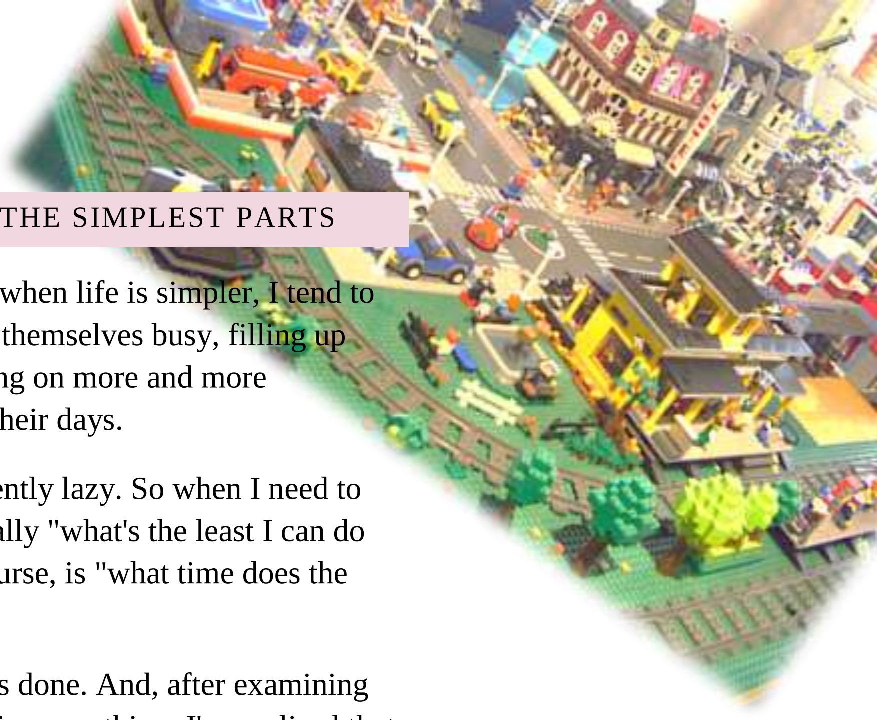
When we're working in teams, with each developer working on different pieces of the software simultaneously, we would also follow a similar procedure (but without releasing the software to the end users) every time we integrated our work into the shared source code repository, so we could be sure that all the individual pieces work correctly together and that any changes we've made haven't inadvertently impacted on changes someone else has been making.

So we could be repeating this sequence of steps many, many times. This is therefore another great candidate for automation. Experienced teams write what we call "build scripts" and "deployment scripts" to do all this laborious and repetitive work for us.

There are many other examples of boring, repetitive and potentially time-consuming tasks that developers should think about automating - like writing programs that automatically generate the repetitive "plumbing" code that we often have to write in many kinds of applications these days (for example, code that reads and writes data to databases can often end up looking pretty similar, and can usually be inferred automatically from the data structures involved).

We need to be vigilant for repetition and duplication in our work as software developers, and shrewdly weigh up the pros and cons of automating the work to save us time and money in the future.

GROW COMPLEX SOFTWARE USING THE SIMPLEST PARTS



One thing I learned years ago is that when life is simpler, I tend to get more done. Other people make themselves busy, filling up their diaries, filling out forms, taking on more and more responsibilities and generally cluttering up their days.

Like a lot of software developers, I'm inherently lazy. So when I need to get something done, my first thought is usually "what's the least I can do to achieve this?" (My second thought, of course, is "what time does the pub open?")

Somehow, though, I do manage to get things done. And, after examining why someone as lazy as me manages to achieve anything, I've realised that it's because I'm an ideal combination of lazy and focused. I tend to know exactly what it is I'm setting out to achieve, and I have a knack for finding the lowest energy route to getting there.

When life gets more complicated, we not only open ourselves up to a lot of unnecessary effort, but we also end up in a situation where there are a lot more things that can go wrong.

Although I'm lazy, I actually have to work quite hard to keep my life simple. But it's worth it. By keeping things simple and uncluttered, it leaves much potential to actually do things. In particular, it leaves time to seize opportunities and deal with problems that suddenly come up.

Keeping things simple reduces the risk of disasters, and increases my capacity to adapt to changing circumstances. I've got time to learn and adapt. Busy people don't.

And waddayaknow? It turns out that much of the joy and fulfillment that life has to offer comes through learning and adapting, not through doggedly sticking to plans.

Software is similar. When we make our programs more complicated than they need to be, we increase the risk of the program being wrong - simply because there's more that can go wrong.

And the more complex a program is, the harder it is to understand, and therefore the harder it can be to change without breaking it. Teams who overcomplicate their software can often be so busy fixing bugs and wrestling to get their heads around the code that they have little time for adding new features and adapting the software to changing circumstances.

When we write code, we need to be lazy and focused. We need to work hard at writing the simplest code possible that will satisfy the customer's requirements.

And hard work it is. Simplicity doesn't come easy. We need to be constantly vigilant to unnecessary complexity, always asking ourselves "what's the least we can do here?"

And we need to be continually reshaping and "cleaning" the code to maintain that simplicity. Uncluttered code will no more stay magically uncluttered as it grows than an uncluttered house will magically stay uncluttered with no tidying.

But doesn't software necessarily get complicated? Is it possible to write a "simple" Computer-Aided Design program, or a "simple" Digital Audio Workstation, or a "simple" Nuclear Missile Defence System?

While we must strive for the simplest software, many problems are just darn complicated. There's no avoiding it.

Cities are also necessarily very complicated. But my house isn't. I don't need to understand how a city works to deal with living in my house. I just need to know how my house works and how it interacts with the parts of the city it's connected to (through the street, through the sewers, through the fibre-optic cable that brings the Interweb and TV and telephone, etc.)

Cities are inescapably complex - beyond the capability of any person to completely grasp - but living and working in a big city is something millions do every day quite happily. We can build fantastically complicated cities out of amazingly simple parts.

The overall design of a city emerges through the interactions of all the different parts. We cannot hope to plan how a city grows in detail at the level of the entire city. It simply won't fit inside our heads.

But we can apply some simple organising principles to the parts - houses, streets, communities, roads, waterways, power supplies and all the rest - and in particular to how the parts interact, so that what emerges is a working city.

And we can gently influence the overall shape by applying external constraints (e.g., you can't build here, but build affordable houses over there and we'll give you a generous tax break.)

When it comes to organising software in the large, a great deal of the action needs to happen in the small. We can allow complicated software to grow by wiring together lots of very simple pieces, and applying a few basic organising principles to how those individual pieces are designed and how they interact with each other.

We can focus on getting it right at that atomic level of functions, modules and their interactions, working to maintain the ultimate simplicity.

And then we can constrain the overall design by applying the customer's tests from the outside. So, regardless of what internal design emerges, as a whole it must do what the customer requires it to do, while remaining simple enough in its component parts to accommodate change.

TO LEARN, WE MUST BE OPEN TO CHANGE

If there's one thing we can be certain of in this crazy, mixed up world, it's that we can be certain of nothing.

In previous posts, I've alluded often to change, and how important it is in software development.

This final post - putting aside my feeble joke - seeks to reify change to a first-order component of successful software development. It deserves its own principle.

As software development is a learning process, and since we learn by incorporating feedback in an iterative sort of fashion, it stands to reason that our software must remain open to the necessary changes this feedback demands.

If we're not able to accommodate change, then we're unable to learn, and therefore less likely to succeed at solving the customer's problems.

But, although we call it "*soft*ware" (and, admittedly it is easier to change than things made out of, say, concrete) changes to software don't come at no cost.

In fact, changing software can be quite expensive; more expensive than writing it in the first place, if we're not careful.

What happens when software is too expensive to change? Well, what happens when anything becomes too expensive? That's right - nobody's willing to pay for it; except fools and madmen, of course.

Software becomes too expensive to change when the cost of changing it outweighs the benefits of making those changes.

A surprisingly large number of software products out there have reached this sorry point. All over the world, there are businesses who rely on software they can't afford to change, and therefore can't change the way their business works.

When a business can't change the way they work, they struggle to adapt to changing circumstances, and become less competitive.

Lily Strugnell (108), World's Oldest Facebook User

The same goes for the software we use in our daily lives. We may see many improvements that could be made that would add a lot of value, and we may have come to rely on the version of the software we're using. But if the people who make that software are unable to incorporate our feedback, we end up stuck with a duff product, and they end up stuck with a duff business.

Meanwhile, competitors can take those same lessons and come up with a much better product. There are thousands of new businesses out there ready, willing and able to learn from your mistakes.

To accommodate change in our software, we need to minimise those factors that can be barriers to change.

Some of these factors have already been touched upon in previous principles. For example, if we strive to keep our software simple and readable, that can make a big difference. It will make our code easier to understand, and understanding code makes up the lion's share of the work in changing it, as studies have revealed.

If we automate our repeated tests, this can also make a big difference. One of the risks of making a change to a piece of working software is that we might break it. The earlier we can find out if we've broken the software, the cheaper it might be to fix it.

Automating builds and release/deployment of software can also help us to accommodate change. Teams that frequently integrate their individual work find that they minimise the impact of integration problems. And teams that can quickly and cheaply release or deploy their software (and safely undo that deployment if something goes wrong) are in a much better position to release software updates more frequently, so they can learn more and learn faster.

There are other important factors in our ability to accommodate change, but I'm going to end by considering two more.

As well as making our code simple and easy to understand, we also need to be vigilant for duplication and dependencies in our code.

Duplicate code has a nasty tendency to duplicate the effort required to make changes to the common logic in that duplicated code. We also risk duplicating errors in the code.

We must also be careful to minimise the "ripple effect" when we make changes in our software. Ask any experienced developer, and they'll be able to tell you about times when they made what they thought would be a tiny change to one part of the software, but found that small change broke several other parts that were depending on it. And when they fixed those dependent parts, they broke even more parts of the software that were in turn depending on them. And so on.

When the dependencies in our software aren't carefully managed, we risk the equivalent of "forest fires" spreading throughout it. A seemingly small change can end up turning into a major piece of work, costing far more than that change is worth to our customer.

Finally, in order to accommodate change, we must be open to change. The way we work, the way we communicate with each other, the way we plan what we do, all has to make change - and therefore learning - easier.

Too many professional software developers have a fear of change, and too many teams organise themselves around the principle of avoiding it if they can.

For example, many teams do everything humanly possible to avoid customer or end user feedback. They can become very defensive when someone points out a flaw in their software or makes suggestions for improvements. This is often because they fear they cannot act on that feedback, so they employ the coping mechanism of hiding, or getting angry with the person offering the feedback.

Many teams employ complex, bureaucratic procedures for "change management" (which is software-speak for "discouraging change") which can only be designed to put customers off asking for new things.

The language of software development has evolved to be anti-change: commonly used terms like "code freeze" and "scope creep" are aimed at encouraging a culture where change is bad, and no change is good.

When we approach software development as a learning process, and accept that much of the real value in what we create will come from feedback and not from what we originally planned, then we must not just tolerate or allow change, but actively *embrace* it.

ABOUT CODEMANSHIP

We provide training and coaching to software teams in the key technical disciplines that are critical for sustaining the pace of innovation in your business.

Formed in 2009 by industry veteran Jason Gorman, our clients include the BBC, Electronic Arts, Channel 4, Sky, Caplin Systems, Rabobank, Capital Group, Treyport, Higher Education Statistics Agency, AXA Swiftcover, XLN, Red Gate Software, 7digital, Pinesoft, Canal Digital, Collinson Latitude, City Index and Siemens Industry Software.

We are proud organisers of the original international Software Craftsmanship conference.

You can find out more by visiting www.codemanship.com