Principles of Software Development

By Allan Kelly, allan@allankelly.net, http://www.allankelly.net

> "principle: 1. a fundamental truth or proposition that serves as the foundation for a system of belief or behaviour or for a chain of reasoning. 2. a general scientific theorem or law that has numerous special applications across a wide field. 3. a fundamental source or basis of something" Oxford Dictionary of English, 2003

Underlying all my writing are a set of principles which deserve spelling out in their own right. The first set of these I believe underly all software development work however it is conducted. The second set require one to accept the ideas behind Agile software development. The third and final set I characterise as "Kelly's Laws". Again I believe these to be universally applicable however I don't believe they will meet with universal acceptance, at least immediately.

As it stands I don't believe this list is exhaustive and there might be some even deeper underlying truth below some of these principles I have yet to expose. There are also plenty of other laws and principles offered by others some of which I agree with lots, some I agree with a little, some I even disagree with. The principles and laws here are the ones I keep coming back to again and again, me these are the important ones.

The degree to which Agile is applicable in a domain other then software development largely depends upon the degree to which these principles hold in that domain. If the principles hold then much of Agile will be applicable. If the principles don't hold then far less of Agile will be applicable.

Software Development Principles:

Principle 1: Software Development exhibits Diseconomies of Scale

I, like many, if not most, readers have been brought up with the idea that if as things get bigger they get cheaper: Buying 2 litres of milk is cheaper than buying 1 litre; building 1,000,000 identical cars is cheaper than building 10 different cars 100,000 each. Blame Henry Ford if you like but somehow Economies of Scale thinking infects much of our thinking.

In software this isn't true. Bigger teams are more difficult to manage, more expensive and less productive per head than smaller teams. This effect is so pronounced that really large teams might be less productive in total than small teams.

For example: a team of five will, per head, be more productive than a team of 25. Still the team of 25 will achieve more in total than the team of 5. However a team of 50 might be less productive than a team of 10 in total.

And its not just teams. One large software releases can be more expensive than many small releases. Producing software to satisfy 100 users is more expensive than producing software to satisfy 10, or 1.

The effect appears again and again. Its why Lean folk like to emphasis small batch sizes. Unfortunately post-industrial society has internalised the concept of mass production and economies of scale. You, me, everyone, needs to purge themselves of economies of scale thinking and embrace dis-economies of scale if you are going to be be successful in this world.

Dis-ecnoomies of scale may well apply to more industries than is commonly recognised. As a software guy, I can only talk with authority about software so that I'll stick to there.

Principle 2: Quality is essential - quality makes all things possible

The quality I talking here is: bugs, internal quality if you like. Poor quality means rework is required, high quality means no rework (i.e. bug fixing) is required.

I want to see bug-free software. This does not mean I want to gold plating of code or requirements, I don't. Nor do I have time for reusable code. I want code which is fit for purpose and free of bug.

Philip Crossby said it best: "Quality is Free". Neils Malotaux puts it more accurately if less dramatically "Quality is cheaper" while Tom Demarco says: "Quality is free, but only to those who are willing to pay heavily for it" (DeMarco and Lister 1987). Capers Jones has extensive matrices to back up his assertion that:

> "Fixing software bugs is the most expensive software activity in history... High Quality leads to high productivity and short schedules." (Jones 2008)

The basic message is the same: pay attention to quality, rid yourself of rework and it will work out better.

There are those people, mainly managers, who believe there is a dial on the wall marked "Quality". When the dial is turned down work is done faster, when the quality dial is turned up it slows down. That might be true in some industries but in software development the evidence points in the opposite direction:

1. **Low quality makes for rework, rework is time consuming and unpredictable.**

2. **Rework increases the time needed, increased time leads to increased costs.**

3. **Increased the time to delivery increases the chance that requirements change.**

4. **More requirements changes lead to more work and later delivery, thus creating a vicious circle.**

Specifically in the case of Agile work: if you quality it low the iterations cannot end cleanly, rework will disrupt future iterations, iterations cannot be satisfactory closed thus you cannot achieve true Agile.

Principle 3: Software Development is not a production line

Various people, at various times, have likened software development to a production line in a factory. While this might be an attractive metaphor it doesn't hold. Factory production lines are designed to product many identical products. Differences between products are seen as defects that need to be fixed.

In IT the "identical copies" problem is solve: Control-C, Control-V does it on most computers. The ability to make many identical copies of a digital product as negligible cost has destroyed the music industry as we knew it. It has changed other industries too and is - at the time of writing - changing the publishing industry.

Software development is not a manufacturing industry, rather is is a product design industry. Hence while we may borrow from the production line metaphor it should not be used with due care.

Importantly given Agile's links to Lean, while the much of the writing on Lean Manufacturing is interesting and can provide insights it is limited in its applicability. Rather Lean Product Development is where software people should look. Unfortunately Lean Product Development is less well understood and documented then Lean Manufacturing.

Agile Software Principles

Agile Principle 1: Highly adaptability over highly adapted

Strive to make you processes, practices, team, code and environment adaptable rather than adapted. The future is uncertain and shows no signs of getting more predictable. Rather than try and predict the future and be adapted to it, strive to be adaptable to what ever may come.

Code design and architecture is a good example of this. Rather than spending a lot of time designing a system for the requirements you think the business wants accept that things will change. Even if the business swear on oath that this is what they want and sign in blood things will change.

Adaptability comes not from exhaustive contingency measures but from simplicity, reflection, learning and change and, above all else: good engineering.

Plan, design, for the short term but put in place mechanisms to roll with change. Spend hours not weeks designing, be prepared to refactor, put in test frameworks to allow change. Sometimes there are architectural things you can do to allow for change - for example the plug-ins pattern - but these are fewer than you think. Above all else don't try to second guess what might happen tomorrow, next month or next year.

Don't implement this or any other architecture until you actually see the need for it. Don't build it because you have a hunch.

Agile Principle 2: Piecemeal Growth - Start small, get something that works, grow

> "By piecemeal growth we mean growth that goes forward in small steps, where each project spreads out and adapts itself to the twists and turns of function and site" [@Alexander1975]

In keeping with the adaptability principle above and dis-economies of scale: always start as small as you can, if things work then grow. For example: Start with the minimal viable product, if customers like you product add to it. Start with the smallest team you can, if the team delivers good software people use then grow the team.

This principle applies not just to the code or the product under development but the development process, the size of the team and much else besides.

When considering Agile itself start with very light weight processes and a minimum of practices. Implement something small and add as needed. To put it another way: start small and tailor the approach by adding.

This is the reverse of traditional larger methods (e.g. PRINCE 2 and SSADM) which advised practitioners to start large and tailor down by removing parts.

Agile Principle 3: Need to think

Agile is not for those who want to follow a flow chart or for those who want to leave their brains at home and follow procedures. Indeed, if you are such a person then working in IT is probably a bad move in the first place.

Not every Agile method or practice will work where you work. Agile experts and methods disagree. You need to think about this.

You also need to think about what you are actually doing, what you have been asked for, and how you can work most effectively. Unfortunately economies of scale thinking has created a lot of companies whose modus operandi seems to be to stop staff from thinking.

Agile Principle 4: Need to unlearn

> "I can't understand why people are frightened of new ideas. I'm frightened of the old ones" John Cage, composer

Doing Agile practices is the easy bit. The hard bit is unlearning: the things you have to stop doing. If you have burn down charts you don't need Gantt charts; if you have an automated test suite you don't need a regression test process; if you have supplies you trust you need not waste time on penalty clauses; etc. etc.

Things that have made us successful in the past are included here. Teams advance, companies change, to much baggage from the past is carried forward making life in the new world expensive. Remember the words of John Maynard Keynes:

> "The difficulty lies, not in the new ideas, but in escaping from the old ones, which ramify, for those brought up as most of us have been, into every corner of our minds." The General Theory of Employment, Interest and Money (Keynes 1936)

Agile Principle 5: Feedback: getting and using

Agile doesn't work out of the box, as the two points above say: you have to find what works for you. Some of this might be obvious at first but the more you get into this the more you are dependent - in all sorts of ways - on feedback.

Feedback about the thing you are building. Feedback about the way you are working - as team and as individuals. Feedback about how your customers are responding. etc. etc. etc.

Agile Principle 6: People closest to the work make decisions

The people doing the work are usually in the best position to make decisions about the work itself. They have the most knowledge about the work and they have the most immediate need of a decision.

Every time a decision traverse up a decision tree information is lost and delay is introduced.

Agile Principle 7: Know your schedule, fit work to the schedule not schedule to work

> "Hofstadter's law: It always takes longer than you expect, even when you take into account Hofstadter's Law." (Hofstadter 1980)

Easy really. Know when something is needed by and fit the work to that schedule.

Part of the problem with estimates is that humans are not very good at estimating, the so called "planning fallacy" (Kahneman and Tversky 1979). A second problem is that people tend to leave work until shortly before it is done so even with enough time they don't start on it until near the deadline. People are however good completing work by a deadline (Buehler, Griffin, and Ross 1994).

Agile Principle 8: Some Agile practices will fix you, others will help you see and help you fix yourself

Some Agile Practices - like planning meetings - will administer a fix and you will get a little bit better immediately. Other practices - like retrospectives - will not fix anything but will help you see what is happening and find your own solutions.

Most Agile Practices are somewhere in the middle. Iterations and stand-up meetings are good examples. They may fix some problems immediately - they will improve focus and communication for sure. More importantly they will help people see what is happening and will, in some cases, force you to get better at what you do.

In the case to iterations they help because the introduce deadlines - as per #7. More importantly they put in place a routine which is initially hard to keep to. Getting good at keeping to iterations forces teams to fix other problems. Iterations, and other practices, "Make a rod for the teams own back".

Kelly's Laws

Kelly's Law of Advanced Code Complexity

> *Any code that is sufficiently advanced will appear as unmaintainable to a novice* (Adapted from Arthur C. Clarke)

This law originally came from looking at C++ meta-programming and some of my own experiences handing over C++ projects ("High-Church C++" Kelly 2000). The law seems to hold not just for C++ but elsewhere too.

I still hear from developers who have taken over a code base and find that the original developers used some whacky coding techniques. In some cases the code was simply bad but in a fair few cases it is really advanced stuff. C++ is still the worth offender but Java generics and Aspect Oriented programming also feature.

Kelly's First Law of Project Complexity

> *Project scope will always increase in proportion to resources*

I am even more convinced of this law than I was 10 years ago when I first proposed it. The more people, time and money put into a work effort the higher the expectations. In part this is the result of applying economies of scale thinking in a domain with dis-economies of scale. Companies and teams add more in an effect to exploit economies of scale but in fact the reverse happens.

This law may also be seen as an application of Parkinson's Law: "Work expands so as to fill the time available for its completion.".

Funnily enough, there is another Kelly's Law, coined by another Kelly, a Mike Kelly (Kelly is a very common name). Mike Kelly's Law is: "Junk will accumulate in the space available." Substitute jump with scope, and space for resources and you have the same thing.

Kelly's Second Law of Project Complexity

> *Inside every large project there is a small one struggling to get out*

This law is really consequence of Kelly's First Law. If you have a project with lots of resources the original project will get lost inside of it. All too often the real mission in turning around a failing development effort is liberating the small inner project.

Kelly's Law of Software Subtlety

> *Subtlety is bad - if it is different make it obvious - Write it BIG!*

This law is really about communication. Its saying "If something is worth saying then say it properly". In many ways the visual tracking, Kanban boards, cards and what not in Agile is an example of this law.

Kelly's Law of Documentation

> The bigger a document is the less likely it is to be read
>
> The bigger a document is, if it is read, the less that will be remembered

While this has obvious applicability to requirements documents it is generally applicable to just about all the documentation we write.

The Iron Triangle

The above figure shows a variation on what project managers often refer to as "The Iron Triangle." I say variation because although the Iron Triangle should be fixed it does appear in different version from place to place - perhaps not so iron them.

I first came across the triangle in Jim McCarthy's Dynamics of Software Development (McCarthy 1995) - a book which I think has never received the attention it deserved - but it is a standard part of project manager training and mythology.

The constrains represented by this triangle are not unique to Agile work. The same constraints and rational apply to all software work. The triangle isn't so much a principle in itself but is the logically of considering the fundamental constraints under which work happens. Oddly, despite being such standard faire opinions differ on what the three sides are labeled.

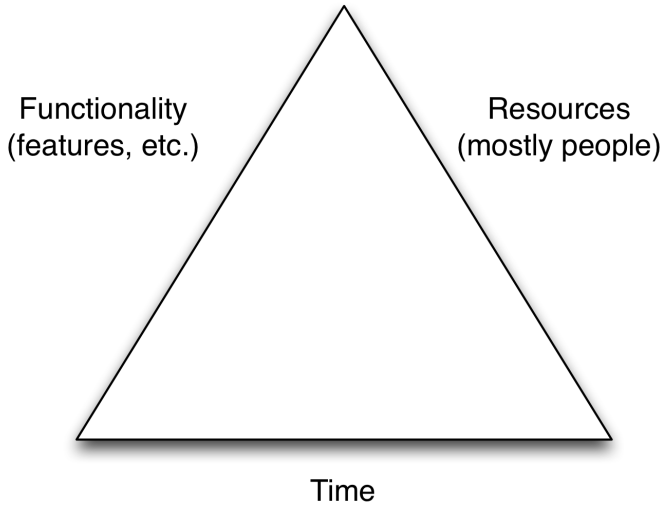

Figure 1: The Iron Triangle for Software

On many version of the triangle one of the sides is labelled "Quality" however I prefer to leave this off the triangle. As already noted - and demonstrated by the work of Capers Jones (Jones 2008) low quality in software development work leads to increased schedules. In keeping with Philip Crosby's "Quality if Free" mantra quality must be kept high, i.e. every effort must made to avoid bugs. Or to use Jones prefer terminology: defect potential needs to be minimised and defect removal efficiency maximised.

If this does not happen, for example developers and testers are discouraged from quality practices, then time and/or resources will increase. If anyone believes work can be accelerated by reducing quality the opposite will happen.

Unfortunately many people have come to believe - even been taught - that reducing quality increases speed and reduced schedule length. This might be true in some other industries but it is not true in software development.

It is as if there is a dial on the wall marked quality. When schedule dates look likely to be missed there are those - often project managers - who turn the dial down and request quality be reduced to reclaim time. In software development the dial is wired the opposite way. If you want to increase speed increase quality.

Secondly, some versions of the triangle label one side "Cost." However in software development the overwhelming source of costs are people and the duration for which they are employed, i.e. how many people you have and how long you have them for. True these people need machines and software but the costs of these resources is a small percentage of the salary costs. While there may be some additional costs not related to people or time these are usually negligible or fixed.

Cost is merely a function of these two parameters:

Cost = Time x Resources

Now lets look at the three constraint parameters: Time, Resources (i.e. people) and Features.

Over the short run time is fixed, a two week sprint lasts 14 days. Although another sprint will quickly follow the first there is always a question of: What will we put in the next sprint?

Even if a team is releasing quarterly (every six sprints say) the same question arises: given the time we have, what do we put in it? Time is fixed, two weeks, 10 working days, or six sprints, 12 weeks, 120 working days. And the costs over these periods are equally fixed.

As discussed elsewhere, working to a deadline is more effective than working to an estimate. Thus it is always a question of: what do we do next sprint?

In the same way that time is fixed over the short run so are resources. Brooks Law (Brooks 1975) states: "Adding more people to a late project makes it later." A more general form of this law may be: "Adding people to a work effort slows it work."

New people know little about the system under development and perhaps little about the application domain. The first thing they do is ask questions of those who already know the system thus reducing the current incumbents productivity.

Many of the companies I speak to - mainly in the UK and specifically London - find that hiring a new developer in anything under three months to be impossible. Once hired a new developer needs to ask questions and learn the system.

Coplien and Harrison (Coplien and Harrison 2004) suggest it takes 12 months for a new developer to "come up to speed." While I am sure they are right in many

case my personal experience suggests developers can make a net contribution in about three months if the conditions are favourable, but seldom more quickly.

Put these numbers together and it seems increasing resources takes at least six months and perhaps a lot longer. Very occasion a team might be able to higher quickly, or higher a super-quick learner, or more likely poach someone internally who has some knowledge but these are the exceptions - although this usually displaces the problem somewhere else.

Few teams are able to increase the size of the team in the next two weeks or even the next quarter. Therefore: the resources side of the triangle must also be regarded as fixed.

(Of course resources may be reduced far more quickly and unpredictably. Even if companies do not reduce the number of staff on a piece of work employees may resign, retire or get sick. Ideally each team would aim recruit new people at a pace commensurate with natural reduction just to stand still.)

This leaves one dimension in which a project, or other work effect, may be controlled: Features and Functionality, i.e. what the software does.

Functionality negotiation should be an on going, rolling process. Work should, ideally, focus on doing the most valuable thing as soon as possible, i.e. the next sprint. Given that few teams have excess capacity this is always going to involve hard decisions about what is scheduled next and what waits for a future sprint.

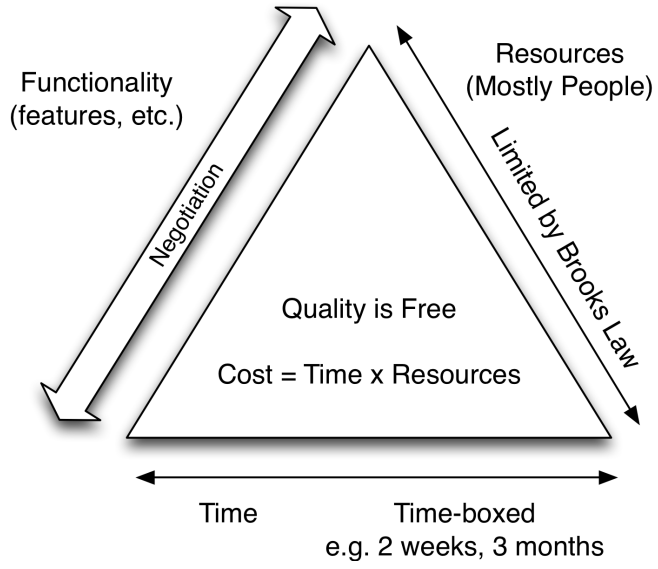

Figure 2: The Iron Triangle Constrained

That these discussions are on-going, changing and evolving demonstrates the

need to ensure somebody is assigned to this role at all times. Often this person goes by the title of Product Owner in Scrum, Customer in XP, or Business Analyst in corporate environments and Product Manager in ISVs. Whatever the title the key point is: this is an on-going activity not a process that once completed can be considered done.

Brooks Law is always in effect and while the time scales maybe longer - and more flexible - there comes a point when a team must - Agile or not - deliver. Providing the team with more time simply postpones much of the negotiation.

This negotiation may be the key different between traditional - so-called Waterfall - work and Agile work. In traditional work the negotiation was agreed at the start of the work, perhaps months or even years before it was likely to be done. As work progressed time might be extended and more resources assigned to the work. But if a team persistently fails to deliver then sooner or later these extensions will cease - or the whole effort will be cancelled.

Only when these two options were no longer available (e.g. because nobody would pay for the increased costs) would negotiations on the "what" be reopened. However I would suggest that every software effort that has ever shipped has eventually renegotiated the "what."

Conversely on in Agile work time scales are fixed - perhaps artificially - and negotiation on the what is to be built happens on day-1. It then happens at the start of every iteration thereafter, i.e. approximately every two weeks. Negotiation on what to build is an on going and evolving conversation.

Conway's Law

Another principle software developers and their managers need to be aware of is Conway's Law named after Melvin Conway. There are times when I wonder if there is any point to the discipline called "Software Architecture." Sure design can make a difference in the small but when it comes to the big then, for my money, Conway's Law is a far more powerful force than the plans, designs and mandates of an enterprise architect.

Conway's Law predates Agile and Waterfall but it applies to both:

> "Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure." (Conway 1968)[1]

[1] The original piece was published in Datamation, April 1968. It is currently available from Conway's own website: http://www.melconway.com/Home/Home.html.

While Conway's Law can be seen at the macro level, the company level, it is also observed in in the small, at the micro level. The law if often restated in the small as:

> "If you have four developers writing a compiler you will get a four pass compiler"

Or

> "If you have three developers writing a UI you will get three ways of doing everything (mouse click, menu item, short-cut key)"

For example, if a development team is set up with a SQL database specialist, a JavaScript/CSS developer and a C# developer you they will produce a system with three tiers: a database with stored procedures, a business middle tier and a UI tier. This design will be applied whether it is the best or not. Indeed, the system might not even need a relational database or a graphical interface - a flat file might be quite adequate for data storage and a command line interface perfectly acceptable.

It is a little like a children's cartoon where the superheroes need to rescue somebody: Superman says "I can use my super strength to lift the ice boulders out of the way", while Fireman says "I can use my fire breath to melt the ice" and Drillerman says "I use my drill hands to make a rescue tunnel". Once added to the picture each specialist finds a way to utilise his or her special powers. For a software team this can lead to poor architecture, over complicated designs and competing modules.

In setting up a team architectural decisions are being made even if no architecture diagrams are being drawn. Just deciding to staff the team with four developers rather than three will influence the architecture because work now needs to be found for the fourth team member.

These decisions are made at the time when the least information is known - right at the beginning. They may made by planners - project managers - who have little knowledge of the technologies or by people who will not actually be involved in the development effort, e.g. enterprise architects.

To avoid this my advice is to start a team as small as possible. Try to avoid making architectural decisions in staffing the team by understaffing it. Keeping the team hungry will reduce the possibility of building more than is needed or over architecting it.

There are those who would quickly point out the risk of under architecting a system, not looking to the future and not building a system that can change and grow. But the risks of over architecting are if anything worse. Too much architecture can equally prevent a system changing and growing, and too much

architecture leads to more time consuming and expensive code to cut. Then there is the risk of not shipping at all, too long spent producing the "right" design may result in a system too late to be viable.

Second staff the team with people who are more generalist than specialist, people who have more than one skill. Yes have a Java developer on the team but have one who knows a bit of SQL and isn't scared of a little UI work. In time you might need to add database and UI specialists but delay this until it is clear they are needed.

These suggestions echo Conway's own conclusion at the end of his paper:

"Ways must be found to reward design managers for keeping their organizations lean and flexible. There is need for a philosophy of system design management which is not based on the assumption that adding manpower simply adds to productivity."

It is worth remembering that Conway was writing over 20 years before Womack and Jones coined the term Lean to describe Toyota.

Reverse Conway's Law & Homomorphic force

Since Conway wrote this a lot more systems have been developed. Many of these systems are now referred to as "Legacy Systems." In some cases these systems force certain structure on the team who maintain the systems and even on the wider company.

For example, take the 3-tier database, business, GUI system from above. Years go by, the original developers leave the company and new staff are brought in. Perhaps some of these have been and gone in the years. The net effect is a system so complex in all three tiers that each requires a specialist. The database is so rich in stored procedures, triggers and constraints that only a SQL expert can understand it. The GUI is crammed full of JavaScript, CSS and not HTML 5 that only someone dedicated to interfaces can keep it all working. And the middle tier isn't any different either.

Given this situation the company has no option but to staff the team with three experts. Conway's Law is now working in reverse: the system is imposing structure on the organization.

Again this happens not just at the micro-level but at the macro-level. Entire companies are constrained by the systems they have. Economists might call this path-dependency: you are where you are because of how you got here not because of any current, rational, forces.

Underlying both Conway's Law and Reverse Conway's Law is the Homomorphism, or as I prefer to think of it: The Homorphic Force.

This force acts to preserve the structure of system even as the system itself moves from one technology to another, from one platform to another.

Conway's Dilemma

Both forms of Conway's Law and the Homomorphic Force pose a dilemma for any organization. Should they work with the force or try to break it?

I can't answer this question generically, there is too much context to consider. However I tend towards saying: Work with Conway's Law, not against it - like woodworking, work with the grain not across it. Be aware of Conway's Law and Learn to play it to your advantage.

Conway's Law does contain a get out clause: the system that will be created will be a copy of an existing system, if you can create a new system, a system not pre-loaded with assumptions and well-trodden communication paths then maybe you can create something new and of a better design.

Thus I sometimes view myself as an architect. I architect not by coding or passing comment on code but by (trying) to bring about a good organizational architecture which will in turn bring about a better system architecture via Conway's Law.

References

Brooks, F. 1975. *The mythical man month: essays on software engineering.* Addison-Wesley.

Buehler, R., D. Griffin, and M. Ross. 1994. "Exploring the 'Planning Fallacy:' Why People Underestimate Their Task Completion Times." *Journal of Personalty and Social Psychology* 67 (3): 366–381.

Conway, M. E. 1968. "How do committees invent?." *Datamation* (April 1968). http://www.melconway.com/research/committees.html.

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

DeMarco, T., and T. Lister. 1987. *Peopleware.* New York: Dorset House.

Hofstadter, Douglas R. 1980. *Godel Escher Bach: An eternal golden braid.* Harmondsworth: Penguin Books.

Jones, C. 2008. *Applied Software Measurement.* McGraw Hill.

Kahneman, and Tversky. 1979. "Intuitive Prediction: Biases and Corrective Procedures." *TIMS Studies in Management Science* (12): 313–327.

Kelly, A. 2000. "High Church C++." http://www.allankelly.net/writing/WebOnly/HighChurch.htm.

Keynes, John Maynard. 1936. *The general theory of employment, interest and money.* [S.l.]: Macmillan.

McCarthy, J. 1995. *Dynamics of Software Development.* Microsoft Press.

This essay is a work in progress. The author welcomes comments and feedback at the address above. April 2013

About the author

Allan Kelly has held just about every job in the software world, from system admin to development manager. Today he works as consultant, trainer and writer helping teams adopt and deepen Agile practices, and helping companies benefit from developing software. He specialises in working with software product companies and aligning products and processes with company strategy.

He is the author of two books "Business Patterns for Software Developers" and "Changing Software Development: Learning to be Agile", the originator of Retrospective Dialogue Sheets (http://www.dialoguesheets.com), a regular conference speakers and frequent contributor to journals.

Allan lives in London and holds BSc and MBA degrees. More about Allan at http://www.allankelly.net and on Twitter as @allankellynet (http://twitter.com/allankellynet).