XP Day 2012
More than XP. More than a day.
Hello,

XP Day is more than XP and it’s more than a day.

The first XP Day conference in London was 2001, making this the 12th annual event. This may also be the final installment of XP Day in London. Throughout the years it's helped nurture and grow a community that has now matured and gained widespread acceptance.

XP Day has changed over the years. Initially it was one of the only places in the UK that you could go find out about eXtreme Programming and Agile. Things have changed and now there are many opportunities to learn about Agile throughout the year. As a result, XP Day has changed. It is now an opportunity for the UK Agile Practitioners to come together to share their latest thinking in an “open space” format conference. As the attendees are now providing the content, we have subsidised the conference with sponsorship. This year we'd like to thank Camelot, QWAN, and EBay for their contributions.

XP day is organised by members of the eXtreme Tuesday Club (XTC). A group that has been meeting every week for over a decade. There is something special about XTC. It has become an established stopping off point for well known people in the Agile and Lean software community. But it remains a place were both experienced and novice local practitioners can come and enjoy a drink and good conversation.

A number of members of XTC speak at international conferences. They also write blogs, articles and tweet regularly. This magazine is a celebration of the quality of writing of XTC members, or people who are associated with XTC.

As you read through this magazine, pause a moment at each advert. The adverts are for the sponsors who have paid for us to have a good time at XP Day.

If you enjoy reading this magazine, you might want to join us at the eXtreme Tuesday Club one evening. We currently meet upstairs at the Bishop’s Finger opposite Smithfield’s market. It is important to check the wiki as we occasionally move venue or have special events. The wiki address is below however I find the easiest way to find XTC is to Google for “eXtreme Tuesday Club”.

Regards,

The XP Day Organising Committee
XTC Wiki: http://xpday-london.editme.com/eXtremeTuesdayClub
The Long Tail of Technical Debt
Michael Feathers, author of “Working Effectively with Legacy Code”

The other day, I was reviewing test execution times across several projects. The results were not surprising. A small number of tests consumed a lot of the execution time, but past a certain point the execution times tapered away to nothing over a very long range. They didn't look like traditional "long tail" distributions. Yes, the tails were long, but they were very flat. Here's a graph of one project. The data that you are looking at is a list of test execution times sorted in descending order:

This test set takes 19 minutes to run on a cheapo machine and the total number of tests is 741.

Actually, we can't see anything in that graph because of the largest test execution. It clocks in at 300 seconds. I think I can smell a whiff of framework and database, with a side of grand finale testing. Let's lop that one off and look at the remainder of the data.

It's sort of interesting to see this data, but what does it mean for us practically? Well, when we are aware of of these timings, they can help us make decisions.

Suppose we want to speed up our tests. There a a couple of ways that we can go. One is to attempt global optimizations. If we could, say, reduce the execution time of each of our tests by 50%, we have a 50% gain. Here is a graph of original test execution times (in blue) plotted against the same times reduced by 50%:

Another strategy that we can use is to concentrate on speeding up the slowest tests. To get a sense of how much we could gain in an ideal world, we can figure out the smallest number of tests we could delete to get a 50% gain in execution speed. In this case, the the answer is: pick 5.5% of the tests in decreasing order of execution time.

Because we can't just go around deleting tests (as tempting as it is), it makes more sense to pick a more realistic percentage, say 50% of execution speed for the slowest 10% of tests and follow the line backward.
This graph shows the execution times once we've cut %50 off the slowest. This might appear to be optimistic, but with tests it's often the case that you can get dramatic improvements, especially if people are touching the database unnecessarily.

If we re-sort the data, it looks like this:

The terrible thing is that distributions like these can lead you to be optimistic in the short term and pessimistic in the long term. They inspire optimism because it looks like there is a lot of low hanging fruit. The pessimism comes after you've plucked it. You are left with a long tail of timings you can speed up but with ever decreasing return on investment. In fact, the tail is often longer because long running tests are often sped up by breaking them into several tests. This isn't always the case, though. Quite a bit of test optimization involves deleting unnecessary execution paths.

If you are thinking that this scenario plays out in more than just test execution times, you may be right. I have a feeling that this sort of distribution shows up other areas of software. We bias toward something like short tests and execution time correlates pretty well with it, but there are times when we miss and those outliers cause trouble. Take the example of class size. We can tackle the worst god classes in our systems, but we will often end up increasing the number of classes and leaving less low hanging fruit for later.

One reassuring thing for this particular project is that there really are very few slow tests. In fact, 78% of total test execution time is spent by only 20% of the tests. The 50% payoff point is at 5.5% of the tests. 5.5% of 741 tests is about 40 tests. Speeding those up could be well worth the effort.

I don't think that I can definitely say that speeding up your slowest tests gives you better returns than trying global fixes like better hardware and parallelization, but I think there are some implications here. You should perform measures on your own test execution and see what you can get with optimization of slow tests. Remember, though, that eventually you'll have picked all of your low hanging fruit. Global optimizations are the path forward at that point.

This article was originally published on my blog at http://michaelfeathers.typepad.com where you can find details of my upcoming class “Reducing Technical Debt” on 17-18th January in London. I’m also doing a workshop hosted by Extreme Tuesday Club “Developing Project Guidance Through Code History Mining” on 16th January. Hope to see you there!

\[1\] http://michaelfeathers.typepad.com
Musings on Conway's Law and SOLID Design Principles
Rachel Davies

In 1967 Melvin Conway submitted a paper called "How Do Committees Invent?" to the Harvard Business Review. HBR rejected it on the grounds that he had not proved his thesis:

"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."

He got the paper published a year later in Datamotion magazine and if you’d like to read the original it’s available on his website here: http://www.melconway.com/Home/Committees_Paper.html

Fred Brooks subsequently coined the name "Conway's Law" when he wrote about this idea classic book "The Mythical Man-Month".

In his article Melvin asks "Is there any predictable relationship between the graph structure of a design organization and the graph structure of the system it designs? The answer is: Yes, the relationship is so simple that in some cases it is an identity," and he concludes "a design effort should be organized according to the need for communication." The basic point he makes is that the shape of software reflects the communication patterns of the people making it.

When I've heard conference speakers, such as Michael Feathers and Kevlin Henney, invite the audience to consider Conway's Law, their focus is usually spotting problem areas resulting in the code layer rather than the people layer. The division of an organisation into teams and departments can create barriers and interfaces in the software. The effects are usually even worse when multiple organisations are involved and those organisations are in different countries around the world with time-delays introducing inevitable delays into communications. Many traditional software development organisations use “teams” that are a bunch of people reporting to the same line manager but each person taking responsibility for adding specific functionality. These software developers usually have different coding styles and rationale for design decisions. The resulting code can be like a patchwork quilt and hard to maintain due to inconsistent coding conventions.

Agile approaches to software development can be seen as a way of improving communication between people and improve the software they create. When we adopt XP practices of Collective Code Ownership and Pair Programming, we improve knowledge sharing and code consistency. On an XP team, any developer can work on any part of the code, code is created and reviewed in pairs, this helps the team to produce code that can be understood by any team member. XP seems to be a set of organisational design patterns that works for a small team. Sharing knowledge between teams working on the same code is a challenge so maybe we need to exchange people between these teams in the same way as an XP team does pairing swaps.

As an agile coach, I work more with individuals and teams than on the code. I spent many years as a software developer and enjoy “debugging” people and communication problems. In working with people to simplify and untangle process, I

find myself applying concepts learned in software design, such as decoupling interactions and clarifying responsibilities. Information and work flows through a software, information and work also flows through a software development organisation. Are there any design principles that we regularly apply in software that could benefit how we design our organisations?

I thought it might be fun to revisit the SOLID design principles to see if any apply to the people domain. SOLID is a set of Object-Oriented design principles crafted by Robert Martin and the Object Mentor team in the early 00's to help teach OO design to C++ programmers who might be struggling to make the shift.

The principles are:

- **Single Responsibility Principle (SRP)**
- **Open/Closed Principle (OCP)**
- **Liskov Substitution Principle (LSP)**
- **Interface Segregation Principle (ISP)**
- **Dependency Inversion Principle (DIP)**

Let's look at each of these in turn to see whether they might apply in any way in the people sphere, at both the level of individuals and teams.

**Single Responsibility Principle**, an object should have only one reason to change. A class has a single responsibility; it does it all, does it well, and does it only. The purpose of this principle is to make it easy to understand the class and have a single place to make changes to this behaviour.

If we try to extend SRP to the people layer, this implies each person does a specialised job which is pretty much at odds with Agile software development concepts such as collective code ownership. We know that if we have a single specialist then they can become a knowledge bottleneck and we're in trouble if they get sick or go on holiday. This brings to the surface some constraints that we have to work with in the people layer - we have a limited number of people and they are not always available. In software we're used to a world with virtually no limit on the number of entities we can create with separate responsibilities.

However, we can go down the route of applying SRP for flow of information into the team by establishing a Product Owner to be the gatekeeper for new story requests and set priorities for the team. There is also sense in which SRP might apply at a team level, making one team responsible for all changes to a specific product. This is a core idea in Scrum, one team per product. We understand which team needs to make the changes and they have all the knowledge to make those changes.

On larger software systems, we end up with multiple teams and because of the various technologies involved component teams often proliferate. This may work where there's an even stream of work to keep all these component teams busy. But large agile projects often find that feature teams work better than component teams. So again in the P-layer we're concerned about levelling the workload and avoiding knowledge bottlenecks.

**Open/Closed Principle**, a class should be open for extension and closed for modification. Again this is all about isolating changes. We can't really extend or modify people so I'm not sure that there's a way to apply this in the people domain. At the team level we can add, remove or consolidate teams and having some consistent process across teams makes this easier.
**Liskov Substitution Principle**, instances of an interface should be wholly substitutable regardless of implementation. Implementations of an interface should behave the same way from the point of view of the caller. They should not require callers to have any knowledge of implementation variances to do the same work. Perhaps this applies to the the communication interface to teams. For instance, those processes used when we send the team new support requests and introduce new requirements. If there is a team that handles support requests, we use the same process irrespective of the individual who is picking up the requests, this is in some sense applying LSP. The team can rotate who is on support duty but the interface is the same.

**Interface Segregation Principle**, several interfaces with specific purposes are better than one larger interface. It’s better to have many small interfaces than fewer big ones as this allows them to change independently. I think that ISP does apply for software teams, we need to identify the parts of the organisation that interact with the team and agree a pattern for these interactions. I’d say that this applies to cadence of meetings, such as demos, and having a clear agenda for specific regular meetings with different groups. This can be a big improvement on general management meetings that aim to cover several different things and involve one or two of participants in turn during one long meeting.

**Dependency Inversion Principle**, collaborators should depend upon abstract interfaces, not concrete implementations. Objects should be minimally coupled and know little to nothing about how their collaborators are implemented. I’m running out of steam here but perhaps DIP can be seen to apply to how Scrum handles the separation of the stakeholders and the development team. The team decide how they do the work and owns the Sprint Backlog, commitments with stakeholders are made on the Product Backlog only.

I’ve written this article for XPDay magazine as a recent topic I’ve been pondering and hope to discuss this topic at the Open Space there. I’m also posting this article on my blog because I’m interested to hear your comments. See [http://agilecoach.typepad.com/](http://agilecoach.typepad.com/)
Guiding New Retrospective Facilitators
By Patrick Kua

Review by Rachel Davies

I’m now working as a coach for two XP development teams at Unruly Media. One of the issues that we identified to from the start is retrospectives had become a little stale and in need of freshening up. Although I have plenty of facilitation experience, I would rather see the team members grow their own skills than for me to facilitate every retrospective. So I put out an invitation to all team members interested in learning about retrospective facilitation. Anyone interested in taking a turn at facilitating a retrospective of the other team (than the one they’re currently in) gets personal coaching from me.

A week or so before the retrospective, I arrange to meet with each new facilitator to discuss their ideas for the design and preparation for facilitating the next session. We sketch out the design with some further reading and make a list what needs to be done (source sticky notes, book suitable room, find out what happened on actions and working agreements from previous retrospective, contact remote participants). This approach enables me to join in the retrospective as a participant and also to offer some feedback to the new person facilitating after the retrospective is over.

Each new facilitator brings a fresh approach to the meeting and being from another team has a neutral perspective to current issues being faced. In taking the plunge to lead a retrospective, they are able to build their confidence as meeting facilitators in a safe environment. I hope that as we grow our facilitation skills and become more familiar with the meeting mechanics that we will get deeper engagement in our retrospectives.

Naturally, a little patience is required from the team hosting a new facilitator - this is often the first time that they’ve run some retrospective activities and pacing these comes with practice. Facilitating is hard work - paying attention to the current conversation at the same time as watching the clock and keeping legible notes is not as easy as it looks! Another challenge is that one of our team working in a remote location and experimented with a number of different approaches to include her in our retrospectives.

Luckily Patrick Kua has just written an excellent new book on retrospectives, we bought it for our team right away. Patrick is an experienced coach and technical leader with many practical tips to share in The Retrospective Handbook: a guide for agile teams. This book is perfectly pitched for the new facilitator going over key steps in preparing and leading a retrospective (nicely complementing Agile Retrospectives which is more of a recipe book of activities). Patrick’s book also includes handy chapters on Distributed Retrospectives and Keeping Retrospectives Fresh so ideal for our team.

I’d be interested to hear questions or comments on this approach to sharing retrospective facilitation between teams.

3 http://www.thekua.com/atwork/about/
4 https://leanpub.com/the-retrospective-handbook
5 http://pragprog.com/book/dlret/agile-retrospectives
Thinking, Fast and Slow.
By Daniel Kahneman.

Review by Julian Kelsey

Making decisions is critical and in software engineering a great deal depends on the quality of our decision making. Kahneman's book gives an accessible account of what we know from psychology about how we make decisions. Prior to this bestseller he is most famous as a Nobel prize winner for applying evidence based models of human decision making to economics. Superficially it's not a book about software engineering, but for me it's been more enlightening than many computer texts I've read.

Kahneman tells us that his objective in writing Thinking, Fast and Slow is to improve the way we judge decision making, aiming to provide memorable language and easy models that will make the conversation around the proverbial water cooler deeper when it touches on why someone did or decided something. Beyond that he aims to equip us to make better decisions, a simple sounding proposition although one of the key messages of the book is that we are least well equipped to do anything differently precisely at the times when we are making decisions. The result of this, simple memorable vocabulary and ideas that can be dropped into conversation or remembered under pressure, is effective in the same way that simple aphorisms and mnemonics are the most enduring guides to coding practice, (think of DRY, loosely coupled tightly cohesive, YAGNI, and others).

His basic simplifying model is that there are two modes to our thinking, he calls them 'system 1' and 'system 2'; one makes decisions very quickly based on learned abilities and applies heuristics to fill the gaps, two follows more slowly and does the hard work of being deliberate. Both have weakness and are prone to distinct biases and errors. Understanding those biases and blind spots, and building systems that help us avoid misjudging because of them, is the substance of Thinking, Fast and Slow.

The first class of problems relate to system 1, why do we confidently make bad decisions quickly. Why can't we easily tell the bad decisions apart from the good decisions we make quickly? Many of the answers will obvious, but articulating the obvious can be powerful. For example 'priming' which is the influence of the present environment and recent experiences, or the related idea of 'cognitive ease' which describes ways that some thoughts are easier than others and how that ease leads us to treat those ideas more favourably. He particularly emphasises how we strongly favour what can be seen or what is known and have a corresponding disregard for even the potential that there may be other options.

One of the key ideas is 'substitution', a notion that recurs in various guise throughout the book. When asked a question which is hard to answer we substitute a related question which is easier to answer. Are you becoming a better programmer? You may answer yes, but perhaps the question you've answered is, are you studying a new programming language. It's hard to say what makes a good programmer, and much easier to point out that you've recently been to a code dojo.

A large part of the book is given over to experiences described in terms of forecasts and probabilities and the many ways that we make bad judgements around probabilities. This material is extensive and even with
training in the area I found myself easily, almost irresistibly falling into error in most of his examples. Soberingly in some cases, even after having the nature of the error clearly explained, my intuition rebelled at the choice that would actual deliver the result I really would want. These are fundamentally the problems we have when making business plans and estimates. There are some profound insights for our industry in this area, both in recognising why things go wrong but also in spelling out practical tools for guiding us toward better outcomes.

One thing that resonated very strongly with me were the short fallings of ‘narrow framing’. We make a virtue of the ability to break our problems down into small chunks and to stay focused on value as perceived in just those small parts. With limited human minds we must make simplifying moves like that. But the book powerfully illustrates the problems with narrow framing such as tendencies that mean we overspend on bad bets. The effect of tendencies like these can be exaggerated over many small choices. By contrast a decision process that takes into account many parts together can produce a better whole. The sections given over to these ideas were a sobering lesson in biases that are inherent in many of the more visible and widely practiced techniques current in Agile development, reminding us of the value of counter-balancing practices that help us keep an eye on the big picture.

Another profound idea is the ‘endowment effect’. Simply, we humanly value things by reference to what we already have or what we feel is rightfully ours, regardless of objective value. And from that reference point we are asymmetrical in our preferences: we feel our loses much more intensely than our gains. Many negotiated positions can be made more acceptable by framing them as foregone gains rather than losses, or simply recognising that what each part loses in a compromise will be felt more intensely than what they might gain. Next time you win a debate be aware that person who lost is feeling it even more strongly than you feel your victory, and you’re priming them to resist your ideas in future; put another way, winning arguments may be the worst way of convincing people to do the right thing.

Kahneman’s Thinking, Fast and Slow is accessible and full of gems of insight backed by solid research, the ideas in it have genuine potential to improve both our working lives and many much wider spheres where our decisions have an impact. I look forward to more discussions that draw on it, and to business processes and relationships that take it’s insights into account.
Every now and then, someone comes up with a new way of looking at the world that becomes the next fashionable thing to do. Every time I’ve seen this, there’s usually a space of time in which a lot of people say, “Meh, it’s irrelevant”, or “Meh, consultants”, or “Meh, they’re only in it for the money.” After a while, things settle down and everyone is used to that new model or concept, and it’s no longer seen as edgy or strange. I’ve seen it with Agile, and Lean Software Development, and BDD, and now it’s the turn of Cynefin and Complexity Thinking.

I’d like to shortcut some of that with Cynefin, because I think it’s kind of cool, I’ve found it useful, and it’s not actually that hard to get your head around once you make the small mindshift. I’m going to share a bit about what I know of it, then talk about how it might actually be a useful concept for a dev to have in their head.

Some of us went to the Calm Alpha meet up, created a list afterwards, and have been using it to discuss which kinds of software can be categorised by which domains. I think our need to categorise everything is part of the problem, but I haven’t worked out how to solve it yet. In this post, I’ll be categorising things according the Cynefin definition of the terms, so resist the urge to say, “No! What I’m doing is simple!” or “But it’s complicated!” because Dave Snowden, who created the Cynefin model, uses those terms in a slightly different way to our common usage.

So here’s Dave’s Cynefin model.

Dave Snowden, released under CC BY 3.0 – thank you!

The Cynefin model consists of four domains – simple, complicated, complex and chaotic – with disorder in the middle. The edges of the domains aren’t strict – they’re domains, not quadrants. There’s also a little fold beneath the “simple” to “chaotic” boundary, to show how easily simplicity can tip into chaos. If you’re going to use this for a lot of items, it might be worth using the domains as attractors, rather than categories. Put the items down on a table, then work out where the lines go afterwards. If you can get your head round that, you’re already on your way to understanding complexity thinking.

Here’s how I tell what kind of programming I’m doing.

---

6 http://en.wikipedia.org/wiki/Cynefin

7 http://creativecommons.org/licenses/by/3.0/deed.en
Simple

Everyone can work out how to do this. Like, everyone. The example the CE\textsuperscript{8} guys used was a bicycle chain falling off. It’s easy to work out how to get it back on again. If you’re programming, I imagine stuff like the turtle at the science museum, or Lego Mindstorms with its graphical drag-and-drop interface, might fall into this space. Children and non-programmers can do it.

In a simple environment, you sense, categorize and respond. You say, “Oh, it’s one of those problems.” No analysis is required.

Complicated

Complicated stuff is predictable, but requires expertise to understand. A watch is complicated. If you’re programming, complicated stuff will be well-understood, done before, not going to change as you develop it. Writing yet another CRUD form probably falls into this space.

In a complicated environment, you sense, analyze and respond. You say, “Let me have a look at this problem and I’ll tell you how to solve it, because I’m an expert at this.”

(I have a theory that most devs get really bored by doing the same predictable but complicated thing over and over again. We tend to turn it into open-source or automate it, reducing it to the far smaller but more complex problem of how to do the automation or use the open-source. Devs are drawn to complexity like moths to a bonfire. And we make it when we don’t have it…)

I reckon that if everything in software development was merely simple or complicated, Waterfall would work very well. You’d be able to set out to achieve some goal, work out how to do it, achieve the goal and say, “Job done.” Software doesn’t consist of only simple and complicated stuff, though. So let’s have a look at the other two domains.

Complex

My favourite way to understand complexity is that acting in the space causes the space to change, and cause and effect can only be understood in retrospect.

When you start writing tests, or having discussions, and the requirements begin changing underneath you because of what you discover as a result, that’s complex. You can look back at what you end up with and understand that it’s much better, but you can’t come up with it to start with, nor can you define what “better” will look like and try to reach it. It emerges as you work.

In a complex environment, you probe, sense and respond. You do something that can fail, safely, and it tells you things about the environment which you respond to, changing the environment. This is the land of high-feedback, risk and innovation: generally stuff you’ve never done before, anything that the business are unsure about, new technologies, etc. This is the domain in which Agile techniques really flourish. If you have a look at Cynefin’s pyramids, this is the one with very loose leadership and the people at the bottom all connected together, collaborating to solve a problem.

This is the most interesting domain for me. It’s the reason why we do things like BDD – using examples to discover more about what we’re doing – but it’s also the reason why, if we focus on trying to pin every small requirement down, we fail.

\textsuperscript{8} http://cognitive-edge.com/
Chaotic

Chaos is your house catching fire. Chaos is accident and emergency. Chaos is that bug you released to production that brought your site down on the day of release, and you need to drop *everything* and fix it now.

In chaos, you act. You get out of the house. You stem the bleeding. You do *something* to get the situation under better control. When Egor Homakov hacked Github this week, Github responded by suspending his account immediately. They acted on the threat. After that they analyzed his actions, considered what he had done and reinstated his account. Act, sense, respond.

So why should I care as a dev?

I’ve found this model really useful for understanding why certain ways of approaching software work best in certain situations, and fail at other times.

The biggest failures I’ve seen have come from treating complex problems as if they’re complicated. For instance, a common Agile practice is to divide a problem into small chunks that we call “stories”, then start working on them. That’s actually a good way to work when most of the problem is complicated, but if you’re doing something new then you may want to go down the “probe, sense, respond” route instead. Hack something out, and get feedback on it. There’s no point guessing how you’re going to reach the goal, because as soon as you get feedback, there’s a good chance the goal itself will change.

Also, sensemaking is itself a complex thing. By trying to get feedback, you might find out how easy or hard it is just to get feedback. That could change the way in which you engage with the business. It might affect how much time you put into preparing for a planning meeting, whether you make a feature work fully or just fire off a screenshot, etc. As you seek to get that feedback the business will change the way in which they respond, too, so it might become easier or harder, and you need to be watching for those signs so you can help to change the process to match. As a dev, you’ll be the one feeling the frustration or ease from the process. It’s no good just relying on your coach or Scrum Master to help you, because he or she needs your insight too.

Knowing this model helps me know when to do TDD or automate BDD – defining a well-understood outcome, and working creatively with the software to reach it – and when to just use examples as ways of discovering more about what we’re trying to do. If I can clearly articulate the outcome and everyone agrees on it, then probably it’s good for TDD and BDD automation. Otherwise, having conversations is more important than automation, whether it’s with a pair-programmer at a class level or a business spokesperson or tester at a system level. Being able to tell the difference can help ensure that the conversations are the most interesting and effective conversations we can have. If I spot the conversations becoming boring, with people yawning or drifting off, then I know we’re trying to apply a complex method to a complicated or simple space and I can say, “Okay, I think we understand this well enough,” and work on something different instead.

(Also, knowing that the act of sensemaking is complex itself makes me look out for situations in which we’re misapplying methods.)

When chaos erupts, I know that letting my PM order us about for a bit is OK. Dropping everything we’re doing is also OK – forget the whole “developers need to be left alone to do their work” idea that the Scrum timebox normally provides. We fix the bug,
hack the workaround, take the feature down, do whatever we need to do to get things under control. Chaos almost matches the “expedite” class of service in Kanban, except that I think you probably don’t even bother putting a card on the board until the need to expedite is over. After the emergency is over, we can look at whether it was just a one-off or whether there’s something we need to do to avoid happening again. For instance, being able to roll back a deployment cleanly and quickly gives us the ability to probe a release instead of crossing our fingers, moving something that might become chaotic into a complex space instead.

These are the kind of decisions that we make intuitively anyway, as developers, but human intuition is often flawed. We’re often uncomfortable with uncertainty, so we usually try to define outcomes regardless of whether they can be clearly defined or not. We’re definitely uncomfortable with chaos, and often make the mistake of treating a one-off chaotic incident as systemic, stamping controls over everything when what we really need is a way of probing, or trying things out safely. Having this model in my head has really helped me to become more comfortable with those situations, and to find approaches that fit them much better. Of course, this is massively useful when I’m coaching and acting as a change agent or helping a PM work out how to lead a team – but even as a dev, having this model makes a big difference to me and my day-to-day work.

I hope it will help you too.
Doing pair programming tests right
Steve Freeman

In her rant on the state of the industry\(^9\), Liz Keogh mentioned coding in the interview, which triggered several comments and a post from Rob Bowley\(^10\), who reminded us of Ivan Moore’s excellent post\(^11\). I think actually typing on a computer is essential which is why I’ve been doing it for ten years (enough with whiteboard coding\(^12\)), but I’ve also seen examples of cargo cult code interviews where the team didn’t quite get the point:

**It’s a senior responsibility**

Pair programming tests should be conducted by senior developers. First, this shows that the team thinks that actual coding is important enough that senior people have to get involved, it’s not just something they delegate. Second, now matter how smart, juniors will not have seen many different approaches, so they’re more likely to dismiss alternatives (technical and human) as bad style. They just don’t have the history. There are times when a tight group of young guns is just what you need, but not always.

**Do it together**

Be present for the work. Don’t just send the candidate off and tell them to submit a solution, the discussion is what’s important. Otherwise, it turns into a measure of how well someone can read a specification. It also suggests that you think that your time is too valuable to actually work with a candidate, which is not attractive. And, please, don’t play the “intentionally vague” specification game, which translates to “Can you guess what I’m thinking?” (unless you’re interviewing Derren Brown\(^14\))

**Be ready**

Have your exercise ready. Your candidate has probably taken a day off work, so the least you can do is not waste their time (and, by implication, yours). Picking the next item off the backlog is fine, as long as it doesn’t turn out to be a configuration bug or to have already been fixed. One alternative is a canned example, which has the benefit of being consistent across candidates. An example that is too simple, however, is a good primary filter but limits what you can learn about the candidate, such as large-scale design skills.

**Have a proper setup**

Your netbook is cute, portable, and looks great. That doesn’t make it suitable for pairing, not least because some candidates might have visibility issues and the keyboard will have keys in the wrong places. Use a proper workstation with a good monitor so you can both see, and talk about, the code.

---

\(^9\) http://lizkeogh.com/2012/02/18/calmalpha-the-first-request/

\(^10\) http://blog.robbowley.net/2010/07/15/experience-report-pair-programming-interviews/


\(^12\) http://blog.indextank.com/1030/interviewing-engineers-enough-with-the-whiteboard-coding/

\(^13\) http://en.wikipedia.org/wiki/Cargo_cult

\(^14\) http://derrenbrown.co.uk/
Allow enough time

Sometimes things take a while to settle. People need to relax in and you need time to get over your initial flash response to the candidate. Most of us do not need developers who can perform well under stress. I’ve seen great candidates that only opened up after 30 minutes. You also need to work on an example that’s interesting enough to have alternatives, which takes time. If you’re worried about wasting effort on obvious misfits, then stage the exercise so you can break early. You’re going to work with a successful candidate for some time, so it’s not worth skimping.

Give something back

This is something that Ivan mentioned. No matter how unsuitable, your candidate spent time and possibly money to come to see you, and deserves more than a cup of tea. Try to show them something new as a return. If you can’t do that then either you don’t know enough to be interviewing (remember, it should be a senior) or you messed up the selection criteria which means you’re not ready.

QWAN
Quality Without A Name

Learn
Code smells
Refactorings

Visualize
Refactoring scenarios

Improve
Team communication

qwan.eu/refactoring
or ask @mostalive at xpdays
Don’t Be Weedy
Seb Rose

During his excellent software craftsmanship session at Lean Agile Scotland, Sandro Mancuso made an analogy between software maintenance and gardening. A garden needs constant attention – lawns need cut; flower beds need weeded; old and diseased plants need replaced. As all you gardeners out there know, I could go on. The list is long and each season brings its own distinct tasks.

It got me to thinking about a bus journey I took many years ago from Inverness to Dundonnell. I got into conversation with a young woman (as one does) and she introduced me to a new term: permaculture. Permaculture borrowed widely from earlier practices, such as organic and no-dig, and synthesised a different way of thinking about horticulture. Rather than fighting against nature, work with it. Plant ground cover to suppress weeds. Use mulches to avoid the need to dig. Zone your holding so that the areas you visit most are closer to your house.

Maximise the utility of every design decision.

Adopting permaculture is not easy. It requires a change of attitude. There are new skills to acquire and master. Perennials require less input than annuals, so a re-examination of your cropping plan is necessary. And sometimes you may decide to keep doing things the old way for specific crops.

A lot of parallels could be drawn between permaculture and various movements within the software development community, but I'm going to restrain myself. The point I'd like to make is a simple one. Regular, tedious effort is not a given in gardening or software development – it is the result of choices made. Once you know there's a choice to be made, then you're free to make an informed decision. If you don't know there's a choice then you'll probably keep on weeding, digging and cutting the lawn, whether they make you happy or not.
The Moment of Pull

*Meditations on time and the movement of cards.*

Jabe Bloom

A tweet this morning from Liz Keogh, who is attending Lean Kanban Scotland, got me thinking:

lunivore: Using @olavmassen’s slide, @agilemanager says starting something = commitment. So less WIP = less commitment = more options. # lascot12

My twitter response captures the essence of what follows, but I thought it would be useful to explore the ideas in more depth.

cyetain: @lunivore WIP also creates a "observable" moment of commitment, prompting creation of more & evaluation of existing options @agilemanager

Much has been written about the signals we use on kanban boards. Specifically, many people have written about the value of having an information radiator¹⁵ that displays the current state of the system. Many have discussed the value of evaluating the flow of cards across the board in order to improve the process represented on the board. Others have written about the satisfaction of the kinesthetic experience of placing a card in done. I have joined others in the argument that simply visualizing work reveals the nature of that work in a way that makes it both discussable and observable (yes that is a circular argument).

I’d like to talk about what I think is one of the most important but often overlooked aspects of cards, boards and movement.

First let us define — abstractly — the relationship between, a card, an individual, a board, boundaries, WIP Limits and movement.

A Card (or traditionally, a kanban) is a symbolic token representing a unit of effort, either to be completed, or in the process of being completed. In manufacturing settings, these cards are used to indicate the relation between the work and a work station. In knowledge work, these cards hold (or stand in wait for) the relationship between the work, and the individuals involved in the work.

The Board is a system that represents the flow of work across states, moving through time.. At one level of abstraction, the board represents these states with boundaries, or columns. One of the simplest, abstract (and therefore common) set of columns reflects an inversion of our innate understanding of the flow of time. “Past, Present, Future” becomes “Future, Present, Past” and, therefore, “To Do, Doing, Done.” All boards, no matter how complex, share this innate structure: every column has a proceeding state, and a following state.

The individual is the human or system that is related to the work being done.

We can pause here to illustrate the “normal” understanding of information radiators. Individuals place cards on the board to represent the work they are completing. Information radiators, or more specifically Kanban-like boards, are most often used to display information about the state of the relationship of the individuals, work and state of work-in-process to other people, team members, managers and the like. This,
of course, can be very helpful in creating a shared understanding of the work.

We could branch off here to discuss ways of analyzing the flow of work through the board, the relationship of boundaries and how WIP limits can improve the process itself. However, I would like to take the path less-traveled.

Movement of cards across boundaries on a board is governed by explicit policies. Different teams can and do define unique constraints for the movement of cards, but almost all boards follow at least one common policy: Limit the Work In Progress. The WIP limit, as represented on the kanban board is a relationship between a set of cards and a column, where the number of cards in the set is limited per column. There are other potential ways to limit WIP, but this is by far the most common.

In my estimation, the most salient moments an individual has in relation to the board are those moments of pulling a card across a boundary. To be specific, I think there is great value in the moment of completion, wherein the individual can evaluate the work that has been completed against explicit policies before moving the work from one column to another.

However, in my mind, the most critical interaction with the board is the act of pulling a new card “into” DOING. This is the moment when the board comes alive. It is activated by the individual’s relationship to all the cards, columns and boundaries at once. This is when the board is most useful.

When “pulling,” the individual is engaged in evaluating all the possible options for his or her next possible task in relation to the other possible options, the current work in progress and the distal and proximal goals of him/herself and his/her team (“slow thinking”, combinatorial decision analysis). The cards in the previous column stand ready, representing future possibilities, while the cards in the later column offer insight on the disposition of the work. Individuals with more practice and independence will have an additional task at this moment: the generation of MORE options than are currently available – the moments just before selecting and pulling the card offer the opportunity to generativity act, to create possible futures. The language I find most useful to describe this moment is the language of Real Options (I recommend it to you if you are interested in a deeper understanding). The “moment of pull” is the moment of Commitment.

There is another aspect of the “moment of pull” that I would like to consider. When one considers feedback loops and their relationship to trust, one should arrive at the understanding that frequent, observable decisions and their related outcomes are critical to creating trust and individual agency within a system (of people and process).

Many people use the output of the process as the sole evaluation of outcome. This evaluation of the outcome of the work, the product, is notoriously complicated by the fact that MOST software production involves the unique instantiation of NOVEL

---

16 [http://www.infoq.com/articles/real-options-enhance-agility](http://www.infoq.com/articles/real-options-enhance-agility)
functionality (at least from the perspective of the individual). If what we are evaluating is nearly continuous novelty, the frequency of observable decisions and outcomes will tend toward 1, and be plagued by the samples of one or less problem.

I believe that, in flow based systems, there is a meta-level opportunity for learning DIRECTLY related to the “moment of pull.” The “moment of pull” creates a directly observable moment of decision, with a directly observable moment of outcome (moving the card forward). These two moments do not always allow us to evaluate the product outcome, but they do allow us to evaluate as individuals interesting qualities of the work we are doing.

To be clear… WIP limits, Boundaries and Cards make the decision to commit observable. In this way, the board isn’t just about making the work state, and process visible, it is about making the moments of decision observable. The decisions teams make about which boundaries to create, define and focus their learning.

One of the most critical qualities of work in a flow system is its size. How long did it take to complete? Making frequent observations about size can help us understand what I like to think of as the “hockey stick effect” of sizing. Below a certain size of task, the task will “remain in control” and be completed within a “predictable” period of time. As tasks get larger, they risk spiking in difficulty, extending in time, and being ensnared in complications. Effectively sizing task allows flow to be created and maintained.

Kanban boards are often seen to represent the State of a system (which they do), however they also model MOMENTS. These moments are defined as the crossing of boundaries. The crossing of boundaries can be thought of as the movement forward (often into a queue) from the current state upon completion, or at the moment of pull, where a card is moved forward (often from a queue) into an active state.

These moments of boundary-crossing allow teaching and learning moments at a meta-level about the “nature” of the work we are undertaking. In this article I have given an example of the size of tasks as one of the "natures" of work that we can learn about in these moments. As individuals learn more about the act of pulling, they notice when they are successful and when they fail. In this explicit, observable moment of decision making, individuals then gain the agency to make further decisions. As individuals gain agency, the board balances that (decision-making agency) against the team’s common goals and understanding of process.

The gift of the moment of pull is confidence in making commitments, increasing the flow of value, and minimizing the risk of instability of process.
The Language of Risk
Chris Matts

Last week I had an enjoyable chat with Janet Gregory about risk management. We were discussing how risk management is not one thing, rather it is an attitude or approach. The conversation reminded me why I started this blog.

Earlier this year the system I worked on was subject to a methodology audit. I spent several hours working with the auditor to explain the approach we were taking. At the end of the conversation the auditor said they wanted to use a kanban system to manage their audit process.

Why did the auditor like what I said? Because I explained everything we did in terms of risk. When they asked for a “process”, I explained the risk the process was meant to address. I then explained how our different process addressed the risk more effectively.

A couple of examples.

“Do you have stakeholder sign-off on your requirements to ensure they all agree on the priorities?”

reframed in risk terms as:

“How do we address the risk that we might be building the wrong thing?”

and we addressed as follows:

“Every week we present the status of the list of projects we are working on to the steering committee. The longest we can work on the wrong thing for is one week.”

“When is your functional spec?”

reframed in risk terms as:

“How do we avoid the risk of building the wrong functionality?”

“We have a two page functional spec. Any more and the stakeholders will not read it. We also have a whole bunch of examples that the stakeholders have verified as correct, and a mock up in Excel so they have an idea of what it will look like. The key is to get quality feedback from the stakeholders rather than get a signature that means nothing. A signed off spec. sort of transfers responsibility from IT to the business for getting the requirements right... but not really. IT will get the blame if it does not work, even if they have it all signed in stone.”

For the fifty or so process audit points we went through each one doing the same kind of thing. Step one, agree the risk the process step is meant to address. Step two, explain how our team addressed it.

The language is important because it helps you think about the problem in the right way.

From my experience, middle and senior IT management respond well to this way of thinking, as do the business investors.
Making bricks or building houses
Julian Kelsey

TDD has shown itself to be amazingly productive practice. With its tight feedback cycle it’s great for learning. Having very specific disciplines and methods mean some excellent tooling has been developed to support it. It’s a natural fit for the small stories and task we favour for iterative development.

It’s strengths are also its weakness, specifically, it drives an intense focus on the smallest units of development. And as time goes by it’s becoming coming coupled ever more tightly to other practices that also focus down at small units of development.

That plays to a very human bias, substituting a hard question with an easier one. We puzzle of over a tricky question: are we making good software? TDD leads us to answer yes by substituting different questions: Have we written good units? Or, have we worked hard with sophisticated tools?

Good units do not make a good program any more than good bricks make a good wall or good musical instruments make a good orchestra.

The general tide in TDD is to keep looking down at small units. This is reinforced by mocking libraries and test frameworks and a culture that favours unit tests over test that aggregate units, (no one questions having a unit test but it is frowned on to have higher levels tests without unit tests, and slow running aggregate tests may well get axed in favour of a faster running test suite).

TDD teaching methods in some quarters have had a similar bias, discouraging conversation and discouraging looking further ahead than the next test. Further reinforcement comes from common patterns of pairing, ping-ponging for example. Likewise practices that focus on stories with little context, either chunked into iterations or striving to have just a few upcoming and potentially unrelated stories in a stream. Add to that developers who take pride in starting what has been presented as the next story without further consideration.

All these things make it easier to focus on the finest grain of problem solving. But practiced without balances they take focus away from the system as a whole being delivered, hiding synergies and commonalities and large scale structure. Indeed it can cause great frustration when someone practiced with the tools and techniques of TDD plows on into chaos dragging in their wake a partner who wanted more time to reflect or consider their work in a context broader than the current test or story.

In all this the mainstream of TDD risks being a "greedy reductionist". Greedy reductionists are a parody in science that says everything can be described in terms of the lowest level realities, basic particles etc, and that higher order abstractions are needless. Dennett gives an example that illustrates the problem: imagine a calculator that returns 3 when you press 1 + 1, everything else it does just fine; you can describe the operation of this calculator in terms of electrons and semiconductors and completely fail to explain it's most interesting and distinctive behaviour. TDD when focused on the smallest granularity risks the same mistake.

Focusing on small problems is called narrow framing and it can a powerful tool. In many situations the human mind instinctively, perhaps unavoidably, resorts to narrow framing. But narrowly framing problems can be deeply misleading about what makes a good decision. Our tendencies to over spend
to guarantee a win or to avoid loss can be amplified in damaging ways if we don't take the big picture into account. Treating the parts of decomposed problems in isolation can exaggerate biases: TDD works against balancing our decisions in the context of the greater whole.

TDD helps us build good bricks, but it doesn't really say much about building good houses. Of course it's hard to build a good house from bad bricks, just there's a lot more that we fail to attend to if we only do TDD.

When the world was introduced to TDD it was in the context of system and practices that balanced those weaknesses. I personally have become very wary of programming with anyone people who don't actively practice techniques to balance the focus on the fine grain that is inherent in TDD.

There are many possible practices that can provide the balance; XP as a system has several built in:

The embedded customer and the system metaphor both help focus us on the whole system and it's values as a product, (sadly, they've often been the things left behind in emerging hybrid and customised development methodologies).

Code standards can be used to improve the focus on the whole and on code that works together as a coherent whole. One style of code standard identifies those areas of the code that embody the values at stake, (not just classes functions or files, but whole modules packages or pages that work as cohesive systems). Another style of coding standard uses guiding aphorisms, for example: business logic and events should be represented explicitly as objects; fan out should be limited; always wrap third party code; always wrap primitives and native library code.

That last is can be argued but ties into the DDD principle of ubiquitous language: write layers and abstractions so that you are implementing in terms that reflect your users language. The quest for an over arching language rooted in the users expression of the problem domain, and having that language pervade the implementation can be a powerful tool for creating a coherent whole.

Perhaps one of TDDs greatest strengths is that it's easier to do than the counterbalancing practices and we naturally give energy to the things we do well or that are at least easy to do. Real growth, however, requires giving attention to what we do poorly. So the next time you go to do some deliberate practice or try to formalise your team's process, have the boldness to ask yourself what you need to do to improve or to cover for your weaknesses rather than just shining a light on your strengths.

A Bad Citizen In Javaland
Darren Hobbs

You are born with no body parts. One by one, your organs are pushed into you. Should anyone attempt to interact with you before your organs have finished arriving you will die. Luckily this doesn’t happen. Once all your organs are settled you are ready to start a job (life moves fast in Javaland). You ask the universe for the abstract concept of vehicular transportation. The universe gives you a car. You look at the car and tell yourself it’s a motorcycle. You try to climb on the roof and are thrown off. Catching your error you land back on your feet. Finally you try telling yourself the car is a car and that seems to work. You open the bonnet and take out the engine. You reach into the engine and take out the starter motor. You turn the starter motor until the engine starts. You take the steering wheel and pedals out of the car and use them to drive yourself to work.

During the day you need to borrow some money from a colleague. You pick up your colleague, reach into their trousers and pull out their wallet. You open the wallet and pull out some cash. You keep their wallet in case you need more money later. They also keep their wallet, remaining unaware that you have taken out some money and that their wallet is now in two places at the same time. Later, you both attempt to spend the same money twice. The universe explodes and is restarted. An identical copy of you lives your life again. Subtle changes to the fabric of reality prevent the universe from exploding this time. Your day continues.

You leave work with your friends. One by one they forget who you are and leave. You are alone. A man wearing a refuse disposal engineer’s uniform appears from nowhere and assassinates you, placing your body into a recycling bin. Your organs are donated to the next generation. You die. The end.
We’re building something special.
Are you?

We’re building a team of Software Engineers to help build some of the country’s biggest brands. If you’re a talented, lateral thinker who knows how to cut clean code, we’d like to hear from you.

Job specs and current vacancies: [www.linkedin.com/company/camelot](http://www.linkedin.com/company/camelot)
Email: [recruitment@camelotgroup.co.uk](mailto:recruitment@camelotgroup.co.uk)
Call: 01923 425888
Follow: [@CherylRecruiter](https://twitter.com/CherylRecruiter)
Sign up to our linkedin group: [Coderetreat London](https://www.linkedin.com/)

[www.camelotgroup.co.uk](http://www.camelotgroup.co.uk)