FEEDBACK

QUALITY

This publication forms part of Value, Flow, Quality® Education. Details of this and other Emergn courses can be obtained from Emergn, Fitzwilliam Hall, Fitzwilliam Place, Dublin 2, Ireland or Emergn, One International Place, Suite 1400, Boston, MA 02110, USA.

Alternatively, you may visit the Emergn website at http://www.emergn.com/education where you can learn more about the range of courses on offer.

To purchase Emergn's Value, Flow, Quality® courseware visit http://www.valueflowquality.com, or contact us for a brochure - tel. +44 (0)20 8444 4503; email valueflowquality@emergn.com

Contents

INTRODUCTION

Feedback has been discussed throughout this course. It is one of the fundamental concepts that links Value, Flow and Quality – and by its nature feedback impels change.

We tend to think of feedback in a very narrow fashion: 'let me give you some feedback on how that presentation went'; 'have you got the feedback from the first user test?' Feedback in its purest sense is far more fundamental than this – within a system, feedback affects the input, causing a change in the process, whether the system is conscious of it or not. Consider the human body which uses feedback to control temperature – sweating when hot, shivering when cold – or a mechanical version, the thermostat – turning the heating on when the room drops below a certain temperature, switching off when it reaches the desired warmth. For those who prefer to think visually, a diagram of such systems shows neatly that feedback works as a loop. The feedback input causes a recurring reaction.

Figure 1.   Melting snow

Feedback loops can interconnect and nest in unexpected ways. It's one reason why making predictions based on feedback is not always easy. Take a complex system like world climate. A simple feedback loop is: as snow melts, more black rock is exposed, which absorbs more heat, causing more snow to melt. The link to increased temperatures globally might seem obvious – but the local temperature depends on many other factors. A vast inrush of melted snow water might actually change ocean currents. One climate prediction for Britain, for example, suggests it might get significantly colder as temperatures rise globally – because the warm North Atlantic Drift might be diverted away from British shores.

We mention this in our introduction, not because we want to start a debate on climate change, but because we want to make it clear that while feedback is so fundamental to all systems, it does not necessarily make predictions simple. Sometimes – especially in software development – agile practitioners talk as if feedback will solve everything. Whether they're thinking about test-driven development (technical, quantitative feedback) or asking a customer whether they like the colour (qualitative feedback) – they assume that feedback will fix the problem.

It doesn't do everything, but we do need it – especially in IT. Why? Well, let's just start by looking at three things we'd like to be true:

In IT fantasy-land:

1. The customer knows what they want
2. The developer knows how to build it
3. Nothing changes along the way

Anderson and Jeffries exaggerate, but put it entertainingly: 'If your customer knows now just exactly what she wants, and if by the time you're done she's still going to want the same thing, it may be the first time in software history that this has happened.'

By the end of this session, students will be able to:

1. Understand the importance of feedback and how it helps us.

2. Appreciate natural problems we have in using feedback.

3. Understand the structure of feedback loops within the software development cycle.

4. Set up nested feedback loops within each part of the development process to improve responsiveness, customer satisfaction and quality as well as reducing risk.

5. Evaluate the cost of feedback.

6. Realise how to separate valuable feedback signals from environmental 'noise'.

7. Apply feedback cycles to team and process, as well as the project, using Retrospectives.

8. Identify key reasons why feedback can be misused or ignored.

1 WHY WE NEED FEEDBACK

In IT fantasy-land:

1. The customer knows what they want
2. The developer knows how to build it
3. Nothing changes along the way

1. The customer knows what they want

Henry Ford is supposed to have said, 'if I'd asked people what they wanted, they'd have said faster horses'. People do not always know what they want. Sometimes they don't know there's a need, until it's pointed out to them. Sometimes what people claim they want and what they actually want can be pretty divergent as well. 'I want to eat healthily', they might announce. But if you cater for this by offering them a quinoa and soya bean salad, you might find to your annoyance that the same person continues to spend their money in the fried chicken shop next door.

All these problems are prevalent in IT projects. Business customers may struggle to define their requirements or they may change them half way through – either because something in the market has shifted, or because through watching the project develop, their ideas have moved on. Schwaber and Sutherland calculate that over 35% of requirements change in a typical software project – and those are just the ones that are known about at the beginning. If you included all the requirements that are added to a project, the percentage would rise.

For some developers this uncertainty is a form of torture: 'the wretched customer's changed his mind again' can still be heard (even if fiercely shushed). For a company, investing heavily in developing something which turns out not to please the customer, this can mean the difference between fabulous profit and going bust. For the customer it can also be frustrating: 'I know I asked for a nested menu, but now that I see it, it looks too complicated. We need to start again.'

2. The developer knows how to build it

A popular element of the total quality and zero-defects culture imported from Japanese manufacturing was 'do it right first time'. Which would be perfect – if only we knew how to do it right. As Andres and Beck point out in their book, *Extreme Programming Explained*, 'if we are solving a novel problem there may be several solutions that might work or there may be no clear solution at all'. The way we plan software projects – insisting that developers provide an estimate for each element of a project in advance, can assist an attitude of false confidence. We believe that everything is possible, and that because we thought it would take about a month it will take a month. We may not yet know which technology to use, or indeed how to do things effectively with the new technology.

Programming is not always easy. Even if you manage to make your code work, that doesn't mean that you have made it work in the cleanest way, keeping every section independent to keep it maintainable, reusable and scalable. If the developer is held to an estimate, the code is more likely to end up being 'adequate first time' than 'right first time'.

3. Nothing changes along the way

As we've discussed throughout this course, people often respond to risk in a counter-productive way – by increasing the detail of their plans. Schwaber and Sutherland sum it up neatly: 'Project planning could take as long as the actual development of the software. Massive amounts of work went into gathering requirements, defining architecture, and detailing work plans. But all of that work was useful only if the plan was based on accurate information that did not change over time.' Indeed, as time passes it becomes more likely that something will change – so the more time you spend planning, paradoxically, the more certain you make it that your plan will be out of touch.

Activity 1 – *I like it, but...*

That customers will change their mind on receipt of partial delivery is an inevitability, and, as just described, is often seen as an inconvenience. But it is also a useful form of feedback, and in order to demonstrate this, this activity simulates the reaction to partial delivery from both client and provider.

This activity requires 2 players and takes place over a series of 3 feedback sessions, spread over one week, each of which should take 5 minutes.

To minimise disruption to your week, we recommend that you plan the feedback sessions in advance. An example of how this might look in your diary is below. The entries in purple are those that relate to this activity.

Monday	Tuesday	Wednesday	Thursday	Friday
10am – Review meeting 11am – meet Jim for exercise round 1 (rules and writing spec) 11.30am – phone conference	11.30am – planning meeting (5 mins needed to do initial drawings for Jim)	Out all day	9am – 11am interviews (drop off second designs to Jim, pick up his new design to review)	8.45am – re-design with Jim's 2nd feedback – drop off for him to look at 9am – 11am interviews
1pm – lunch with Sue	1pm – lunch and round 2 feedback with Jim	(find 5 mins to respond to Jim's and make adjustments to design)	1.30pm – Lunch with Dom	
		Out all day	2.45pm – Jim for second feedback session 3pm – call Paris office	2.30pm – Jim for final feedback session

The work preparation for each round can be done in your own time, but you need to make sure you get this to the other player in time for them to look at and consider their feedback.

Each player takes on 2 roles – that of client and that of designer.

Player One will be the designer of a new foyer for a bank, and also the client who is commissioning Player Two to design a new foyer in his cinema.

Player Two is the client who has recruited Player One to design the foyer of his bank, and is also the designer for the cinema foyer.

Play:

Round 1

This should take only 2 minutes

First, both players take the role of the client and write the specifications that the designer will need in order to make the initial drawings.

This should be no more than 20 words. It is a specification, not a prototype. Think how much coding knowledge most of your clients have when they give you requirements, and match that level.

Do this together, then take away your partner's specifications so you can prepare the first design.

Round 2

Spend only 5 minutes on this round.

You now need to swap job titles and become designer, not client.

Looking at what your client wants (or doesn't want) to experience in the space, create a physical design that you think best fits their needs. Make a floor plan and/or rough sketches to show your ideas.

For example, if the specification mentions a problem with waiting, you could interpret and solve this by putting plenty of sofas around, or by adding water coolers, or by creating more sales points to reduce queuing times.

Feedback for Round 2

Put your client hat back on and meet with your partner to look at the work your designer has done.

Spend 5 minutes giving and receiving feedback.

Each client looks at the initial design and gives comments based on the following 3 criteria:

- The response to your specification.

- Changes you would like made.

- Anything you weren't expecting that you like, and want kept in the design.

Make sure you keep a copy of this prototype so that you can re-examine it at the end of the exercise.

Round 3

Now as designer, and with the feedback that your client has given you, go back to your design and see what needs modifying, what needs completely re-thinking. Don't forget to consider what the client liked.

This process is repeated until 3 feedback sessions have been completed. (Consider the diary as an example of how this can be scheduled into one week.)

Rules:

- For Round 1, the initial specification is all you have – you cannot ask questions.

- You have 5 minutes to draw your initial plans. You can talk about them to the client, and take questions, only after you present them.

- Your interpretation of how to realise what the client wants can include lighting, colours, furniture, as well as architectural design.

- The process of showing, talking and taking questions can only last 5 minutes.

- As the client, you can change your mind at any point.

Commentary:

While we may pride ourselves on knowing what we want, when presented with an actuality as opposed to a concept, we often find that we've changed our mind.

To the designer, this can feel annoying, but in fact, modifications on partial delivery should be seen as a useful, indeed necessary tool, to deliver the best product.

Debrief:

Describe an occasion when partial delivery prompted a client to change their mind about requirements, leading to what you would consider an improvement in the product.

Could this have happened without partial delivery?

1.1.   How feedback helps

We said that feedback is fundamental to all living systems – this is because feedback permits learning and thus adaptation. This is not always conscious, voluntary or even beneficial. We may not want to feel hungry, but our body has evolved an excellent feedback method to inform our brains when we need more nutrients. It does this even if feeling hungry distracts us from more important issues – like actually finding some food. The feedback encourages change – the first change may or may not be for the better, but the next piece of feedback should enable us to make another change, correcting what happened if the first change was wrong. In other words, feedback helps us quickly adapt to unpredictability.

Single-loop versus double-loop learning

While this works most obviously with humans who learn, remember and build on each piece of experience, you can see the process in action with something that can't build on learning. This is known as single-loop learning – a system in which the activity is repeated endlessly, even if the goal is never reached. The feedback may modify the behaviour, but it never changes the goal.

Have you ever watched one of the robot vacuum cleaners - a Roomba, for example? When the Roomba meets a wall its bumper retracts, activating object sensors. The machine then backs up, rotates and moves forward. It repeats this sequence until it has found a clear path. This is feedback in action – in fact, the Roomba can adapt to new input up to 67 times per second. A Roomba combines this with what you might call the advance plan – an infrared sensor with which it tries to estimate the size and layout of a room before it begins. But the plan would be meaningless without the capacity to adapt to change – because few humans keep their rooms so spotlessly tidy that the layout will not change – perhaps even during cleaning. We should stress that only the very entertainment-challenged should create a Roomba obstacle course in order to prove this.
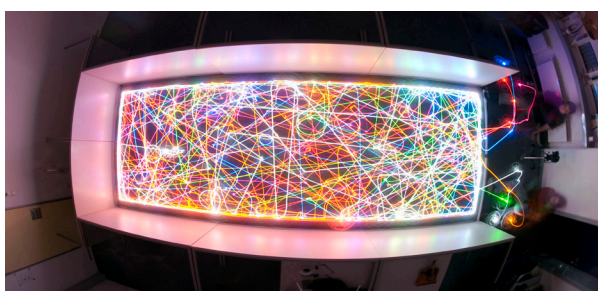
Figure 2.   A pattern generated by a Roomba's movement

Because the metaphorical layout of our environment changes so much and so frequently, our software projects need plenty of feedback to help us navigate obstacles successfully. Indeed, normally we need double-loop learning – a system in which our feedback assists us to actually change the goal if need be.

Faced with a filthy room, a Roomba incorporating double-loop feedback, might decide not to bother cleaning any more since the room always got dirty again. Instead, it might change the goal to 'stay peacefully on recharging point".

Activity 2 – Is yours a single or a double?

This activity will take 5 minutes to be followed by work-place observation.

The situations in the table below have either a single or double-loop response. Often the single-loop answer will solve the immediate problem for us, but the double-loop has the advantage in that it explores the situation at a deeper level, and therefore may come up with a more long-term solution.

Situation	Single-Loop Response	Double-Loop Response
Staff want to keep the temperature above a certain level	1. Put in a thermostat	
Customers don't want to get out of the car to fill it up with petrol		
A supplier is delivering late		

In the list below you will find a selection of either single or double-loop responses – choose which to fill in the gaps in the box.

1. ~~Put in a thermostat, to fire up boiler when temperature drops.~~

2. Hire a forecourt attendant to fill cars for customer.

3. Terminate contract with supplier if timeframes are not met.

4. Redesign pumps and put in place a money back incentive for self service.

5. Review acceptable temperatures. Put on an extra jumper!

6. Check reasons for late delivery and if needed adjust timeframes.

Observation:

For one week, list on a large sheet of paper (or whiteboard if you have one) the feedback you receive for a project you are working on.

As you or your colleagues respond to each piece of feedback, stick one colour Post-it for a double-loop response, and a second colour Post-it to signify a single-loop response along side each piece. Schedule 15 minutes where you and your team can discuss these, and see if you can implement more double-loop reactions the following week.

In IT fantasy-land:

1. The customer knows what they want
2. The developer knows how to build it
3. Nothing changes along the way

1. Helping the customer know what they want

People work best with a tangible expression of ideas. You could describe what a new system should look like and then write a specification down on paper. Or you could build something that gives an idea of what you mean. This can range from a paper prototype (a cardboard box to represent the computer screen) to a demonstration of a piece of working software.

As Alastair Cockburn points out in his book *Agile Software Development*, 'people generally work better by starting with something concrete and tangible, such as examples, by altering rather than creating from scratch, by watching, and by getting feedback.'

The more often and the earlier this is done, the more customers can influence the development process, learning themselves as well as informing and guiding. After all, everyone involved in a project needs to learn as it progresses – and that includes customers and stakeholders as well as the development team.

Sometimes people get worried – will the process ever stop? Will we ever build anything? Gaining feedback does not preclude acting on it, indeed it demands it. Nor does feedback stop you launching and then releasing an improved version later.

Activity 3 – Can you feel the force?

Die-hard *Star Wars* fans may argue, but actually, no, you can't feel the force. Anything abstract is hard to describe, because it's intangible. If, on the other hand, I lifted a spaceship out of a swamp, the effect is concrete, and then you'd be impressed by the force. In this exercise we will explore the difficulties associated with describing, understanding and appreciating the abstract.

For this activity you need paper, pens and 3 players. It will take 30 minutes and makes a good warm up for a team meeting.

Play:

Describe the following list of words to the group, without using the words themselves, "sounds like" or "rhymes with", and without acting out or miming. If possible, do it from behind a screen or with your back to the players.

Do this using no more than 10 words. Be very strict with the word count, give them a few seconds to write down their answer, then move on to the next word.

List of words:

Table	Inspiration
Football pitch	Party
Jealousy	An idea
Nervousness	Coffee cup
Playground	Saturn
Love	Feedback

When you have finished the list check over the answers.

Were more correct answers given to the red or the orange words in your list? Which did you find the hardest to describe?

Can you now think of some recent examples of when being able to provide a physical object or some working software would have helped you demonstrate something to your client?

With the group, discuss where a physical demo might help a project you are working on at the moment. Draw up a plan for implementing this.

Commentary:

"I know that you believe you understand what you think I said, but I'm not sure you realize that what you heard is not what I meant". (Our thanks to Robert McCloskey for demonstrating the elusiveness of language here!)

The answer? Wherever possible, demonstrate, and so take away some of the opportunity for misunderstanding. The model need not be perfect, just as long as it describes the idea better than words alone.

We do not simply progress towards the same goal that we began with. Like the double-loop learning system described above, when we discover new information, we may change our goal. We may take an entirely new direction. Eric Ries in his book *The Lean Startup* calls this the pivot, the moment at which a company realises something fundamental about the business proposition and takes it in a new direction.

A small piece of information can spark a dramatic change. When Coca-Cola discovered that some people were concerned about the sugar in Coca-Cola they created a new sugar-free version called Tab. The product only really took off when the company renamed it Diet Coke. But then a further asymmetric payoff occurred when the company noticed that men weren't drinking Diet Coke. Could it be the name that was putting them off? Coca-Cola launched Coke Zero – an extremely similar product (slightly different flavour profile) but with masculine-styled packaging and advertising.

Figure 3.   Diet Coke and Coke Zero

2. Helping the development team know if they can build it

For developers, feedback works at a variety of levels. At its most basic, a test tells you if your code is working, at the next level up various integration tests tell us whether our code works not just on its own, but with all the other building blocks.

Most importantly, of course, feedback tells us if we are building something useful to our customers, a goal which can occasionally be ignored by developers. A few – not you of course – act as if the real task is to churn out several hundred lines of code, check they do something, and then head home to build another obstacle course for the Roomba. In reality a developer needs to care about whether the piece of functionality does what the customer wanted and expected – otherwise we risk investing time and money in the technical solution, not the customer's problem. An urban legend recounts how NASA spent time and money to invent an anti-gravity pen that would work in space. The Russians took pencils. The story may be apocryphal, but the point is freshly sharpened for software developers not to over-engineer.

Figure 4.   An anti-gravity pen

In *Agile Modelling*, Scott Ambler points out the benefits of early feedback to building software in the most cost-effective manner:

> *"If the only feedback you receive is the errors detected late in the life-cycle of your project, during testing in the large, or after the application has been released, they are likely to be very expensive to fix. However, if you receive feedback quickly, just after misunderstanding what you were originally told, it will be much less expensive to address the misunderstanding."*

The cost of fixing a problem increases with time – this is true whether you're finding a bug in code you wrote before lunch versus a bug in code someone else wrote two years ago, or whether your customer says they wanted an Android not an iPhone app after one day of work versus when the application is finished. Steve McConnell goes into some detail as to the actual order of magnitude of costs:

> *"Studies have found that reworking defective requirements, design, and code typically consumes 40 to 50 percent of the total cost of software development (Jones 1986b; Boehm 1987a). As a rule of thumb, every hour you spend on defect prevention will reduce your repair time 3 to 10 hours (Jones 1994). In the worst case, reworking a software-requirements problem once the software is in operation typically costs 50 to 200 times what it would take to rework the problem in the requirements stage (Boehm and Papaccio 1988). Given that about 60 percent of all defects usually exist at design time (Gilb 1988), you can save enormous amounts of time by detecting defects earlier than system testing."*

The real point is not to get hung up on the exact savings, but to realise that every feedback loop we put in place helps us increase the quality of our product. It helps make our systems more resilient and decreases the amount of technical debt the project carries. A short overall feedback loop of code, test, integrate, deploy is designed to increase a product's quality cheaply. The word 'overall' is important. Spending months on perfecting validation delays the most important 'validation' of all – sales and usage.

3. Helping us adapt to a changing environment

Even if we have built exactly what our customers want, a change in the environment can force us to change our goals. Many things change in the marketplace, in technology, even in our internal workplace. Some organisations make the mistake of trying to foresee these possible changes and create contingency plans, rather than being able to adapt to changes as they happen. If a product no longer seems relevant to the marketplace, we don't simply blindly continue working on it and then launch according to our original plans. We might stop altogether, or we might change our goals and plans.

Let's imagine we have been developing a mobile phone. Unexpectedly, our competitor launches a very similar proposition but for a lower price than the one we had built our business case on. What do we do? We might choose to match their price and try to cut our development or marketing costs. We might choose to launch with a price premium and develop a particular feature in order to justify this.

In either case, some kind of adaptation to the changed competitive environment is essential.

In general, feedback promotes learning and enables better decisions for the team, the product and the customer. Pryce and Freeman summarise it in their book *Growing Object-Oriented Software* as, 'We think that the best approach a team can take is to use empirical feedback to learn about the system and its use, and then apply that learning back to the system.'

How small can these short loops be? That depends upon our needs, but in essence, new information can often be used highly effectively. Once share prices were listed in newspapers and therefore appeared on a daily basis. In 1863, Edward Calahan invented the ticker-tape machine which allowed a continuous rolling update of share prices. All at once trading became far more time sensitive with new opportunities for traders to make (or lose) money based on near real-time events. Every time you go to a gym wearing a heart-rate monitor to check your training intensity, you are gaining the benefit of instantaneous feedback. We'll go on to see later how such 'micro-loops' are especially powerful in software development.

Figure 5.   A ticker tape machine

1.2.   What's wrong with feedback?

We began by pointing out that feedback has its limitations. One of the most important is the way we, as human beings, react to it. Rather than being delighted by a new input that will enable us to improve the project, we see feedback as leading to rework. How many times have you heard someone on your team groan, 'I've got to do it all over again!'? If you're honest, how many times have you felt that way yourself? Searching out feedback, especially unwelcome feedback, requires a significant culture shift for most organisations and individuals.

1.3.   Assumptions

How much learning you can gain from feedback depends upon the quality of what you put in. Normally in software development every input contains a number of hypotheses and assumptions, from the business objective (assuming people are prepared to pay for online content) to the technical (assuming that we can figure out a good enough firewall). We have to check these assumptions, deliberately creating ways of testing them and then evaluating the hypothesis based on the evidence. This only works if we are transparent about our assumptions, intelligent in setting up tests and objective in seeking and analysing the feedback.

Some of the tools we'll go on to discuss are essentially about recording our assumptions explicitly so that we are forced to consider whether feedback should change the original hypothesis.

Case study – Grockit

Grockit was an internet start-up, the brain-child of an ex-teacher, Farbood Nivi, who had begun by setting up a website helping students studying for various tests (GMAT, LSATs and SATs). The VFQ team have a particular soft spot for this company, because it was one of the first to realise and enable the different ways people learn.

Anyway... one of Grockit's particular features was to permit something called 'lazy registration'. The team was convinced that asking people to sign up, go through the hassle of registering and providing credit card details, would be deeply off-putting.

This hypothesis was not only held by the team, it was even enshrined in the industry as 'best practice'. The most advanced companies and web practitioners all shared the same belief. To deal with this, a technique called lazy registration had been developed.

With lazy registration, visitors to the Grockit site could get stuck in straight away, see how good the product was and then decide to register or pay when sent some marketing messages urging them to sign up. Such a feature came at a cost, however. The site was much more complicated to run and maintain because it was managing three classes of user (unregistered, registered and premium) – all of whom needed targeting with different messages at different times.

In fact, the feature was proving so difficult to manage that the company decided to test the hypothesis and see exactly how valuable lazy registration was. They set up a direct A/B test designed to show this. As new users arrived at the site they were randomly directed to one of two possible experiences. One group were asked to register immediately, being given no more information than existed on the marketing pages. The second group continued to interact with the site via the 'lazy registration' method.

According to the assumption that initial registration was a barrier, Grockit expected to see a much higher eventual sign up from those who used the lazy registration method. The only question ought to have been 'how much higher?'.

But the result was not what the company had expected. The test group signed up in the same proportions as those who got to experience the product via lazy registration. Longer-term measures like subscription and retention rates were the same as well. In short, the effort of lazy registration was simply waste.

Instead of supporting this costly feature, the team could invest their time and effort in something else – perhaps the marketing, perhaps features the product provided... testing each change as they went, of course.

1.4.   Confirmation bias

Humans look for feedback that confirms the assumptions we already have. It's a characteristic known as confirmation bias and is just as prevalent in organisations as individuals. The tools of logic and empiricism are not in themselves sufficient to undermine this tendency. Racist scientists throughout the 18th and 19th century found all manner of false evidence to support their theories – measuring skulls to prove that white brains were supposedly larger than black or Asian. The scientist and historian Stephen Jay Gould wrote in his book *The Mismeasure of Man* that some scientists unconsciously over-packed skulls in order to confirm what they expected to find. Interestingly enough, a later critic re-checked the original measurements, decided they were accurate and accused Gould himself of showing a confirmation bias.

The Wikipedia entry on confirmation bias, helpfully points out, 'The effect is stronger for emotionally charged issues and for deeply entrenched beliefs.' You don't need to be arguing about racism to uncover an emotionally charged issue. One's own idea, one's own work, can feel like a precious child that ought to be beyond criticism. Exposing this to the harsh judgement of others can be extremely painful – so painful that we just don't do it, or dismiss unwelcome feedback with a shrug. 'What do they know anyway, unimaginative herd of dinosaurs?'
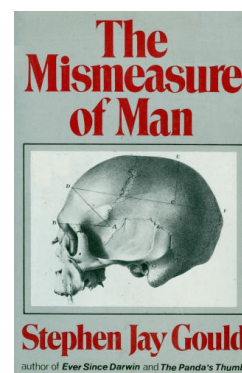
The temptation is to stack the test – find people that you know like the same things that you do, or deliberately include all the great things that might work and not the problems that still exist with the product.

There will always be plenty of stories to remind you of great and lonely geniuses who ignored all the doubters and the nay-sayers and went on to launch their brilliant innovation to eventual acclaim. Just remember, for every lonely genius, there are thousands of stubborn idiots producing rubbish products that no-one buys.

Warden and Shore put it rather more gently in the book *The Art of Agile Development*, 'feedback from real customers is always informative, even if you choose to ignore it'.

The American mathematician and cryptographer, Claude Shannon, is most famous for a paper that considered how to separate out a communication from background noise – rather reminiscent perhaps of trying to find high-quality feedback. The paper led to what is now known as information theory, which suggests, amongst many other things, that the ideal level at which to maximise learning is to aim for a 50% probability of success. Shannon was also a big fan of pointless inventions. He created a motorised pogo stick, an automated juggling clown and a box with a switch on the side which when 'on' caused a mechanical hand to emerge from the box and switch itself 'off'.

Figure 6. An example of the 'Ultimate Machine' by Claude Shannon

Apart from proving that cryptographers really like prototyping with Meccano, the point of discussing the difficulties of feedback is to show that it is also the best solution. Feedback does not provide creativity, certainty or tell us what to do. Nor is all feedback equal - what your customer thinks is more important than what your Mum thinks; what a 16-year old girl in your target market thinks is probably more relevant than what your 50-year-old CFO thinks (although possibly not more influential at project approval meetings). But it is still the best tool we have.

Eventually, most products live or die by real feedback in the sense of whether they generate revenue or not. This is the over-arching feedback loop of software development – of all development come to that. But just as hoping we evolve to cope with environmental stress is an expensive, risky and slow form of adaptation to climate change, so waiting for the market to judge is a risky way to innovate. Instead we need to create our own much shorter and cheaper cycles or feedback loops to help us test our ideas and adapt in the light of new information. Fortunately, software abounds in short cycles that can easily be exploited to do exactly this.

2 FEEDBACK CYCLES IN SOFTWARE

The over-arching feedback loop in software development is concept to cash. It is in some ways the only real form of feedback, but it is slow. One way to try and speed up this feedback loop is to deliver value early. Eric Ries stresses, 'The goal of a startup is to figure out the right thing to build—the thing customers want and will pay for—as quickly as possible.' This goal is valid for all companies – the mature as well as the start-up.

The very nature of most projects, however, means that developing and building something will require so much investment of time and money that we need to maximise its chances of success and minimise the risk of failure. To do this, we need to build in shorter, smaller feedback loops.
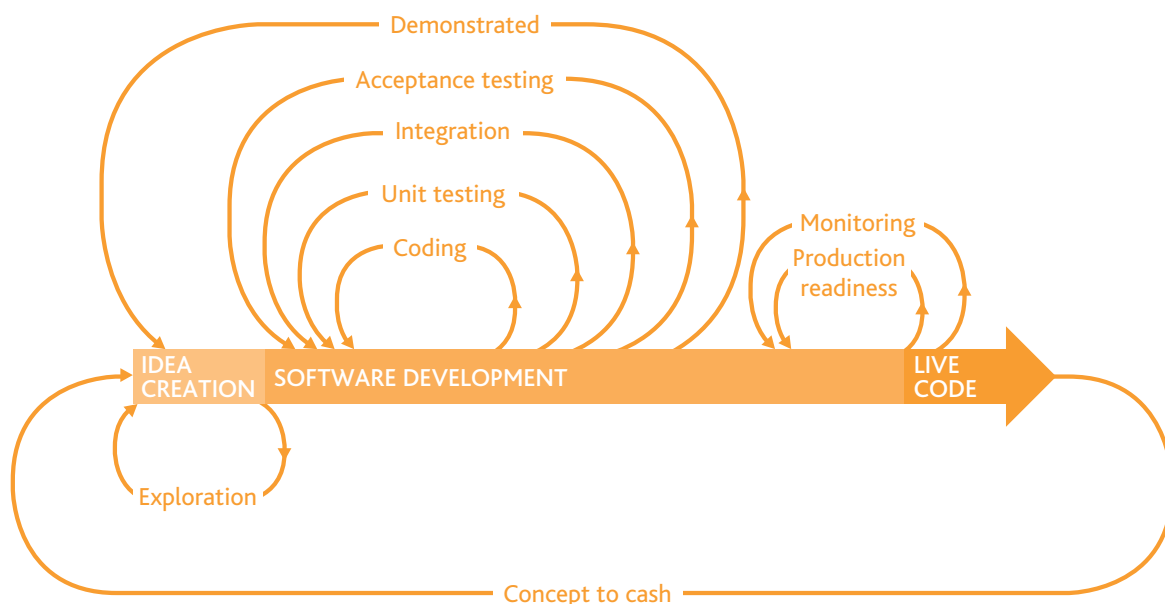
Figure 7.   Feedback loops in software development

Consider the diagram of feedback loops in software development. As well as the overall 'concept to cash' loop, you will see many smaller, nested loops. These are not simply iterative, they reinforce once another. As Pryce and Freeman state in *Growing Object-Oriented Software*, 'if a discrepancy slips through an inner loop, there is a good chance an outer loop will catch it.'

This is at odds with the approach that we should save up checks and changes right to the end. It's a view that often feels like 'common sense'. What's the point of checking the quality, you might think, when they're going to change half the coding anyway? Imagine you were checking this course – would you bother proof-reading and spell-checking before all the comments and changes had been collated? Why not just wait until they've locked down the actual content before worrying about the style?

Waiting until the end to check quality

This approach, which feels more efficient, actually takes more time in the long run. More to the point, it allows errors and defects through. While a spelling-error in this course might not worry you unduly, errors in code can be more serious, directly impacting on customer satisfaction, revenue, and in some cases, even safety.

Activity 4 – Catching errors

This activity will take 2 minutes to run and then 5 minutes in discussion. Use it as a quick warm-up at the start of a team meeting.

Here is a piece of text. Hand a copy to each team member and ask them to proof read it, correcting any errors. The time limit needs to be strictly kept – use a stopwatch!

At the end of the 2 minutes, ask every team member to read out how many errors they found. When everyone has given their figure, compare which errors were found – often they are different. Did you find all the errors? The answer is available overleaf.

Test

Noone would have beleived in the last years of the nineteenth century that this world was being watched keenly and closely by intelligences' greater than mans and yet as mortal as his own; that as men busied themselves about their various conserns they were scrutinised and studied, perhaps almost as narrowly as a man with a microoscope might scrutinise the transient creatures that swarm and multiply in a droop of water. With infinite complacency men went to and from over this globe about their little affairs, serene in their assurance of their their empire over matter. It is possible that the infusoria under the microscope do the same. No one give a thought to the older worlds of space as sauces of human danger, or thought of them only to dismiss the idea of life upon them as impossible or improbable . It is curious to recall some of the mental habits off those departed days. At most terrestrial men fancied there might be other men upon mars, perhaps inferior to themselves and ready to welcome a missionary enterprise. Yet across the gulf of space, minds that our to our minds as ours are to those of the beasts that perish, intellects vast and cool and unsympathetic, reguarded this earth with envious eyes, and slowly and surely drew their plans against us? And early in the 20th century came the great disillusionmeant.

Commentary:

Time pressure is a normal factor in software development and it will affect your capacity to find errors. Different testers find different errors in software; it's also often easier to find someone else's errors, rather than your own.

Answer (20 errors)

No one would have believed in the last years of the nineteenth century that this world was being watched keenly and closely by intelligences greater than man's and yet as mortal as his own; that as men busied themselves about their various concerns they were scrutinised and studied, perhaps almost as narrowly as a man with a microscope might scrutinise the transient creatures that swarm and multiply in a drop of water. With infinite complacency men went to and fro over this globe about their little affairs, serene in their assurance of their remove extra 'their' empire over matter. It is possible that the infusoria under the microscope do the same. No one gave a thought to the older worlds of space as sources of human danger, or thought of them only to dismiss the idea of life upon them as impossible or improbable. It is curious to recall some of the mental habits of those departed days. At most, terrestrial men fancied there might be other men upon Mars, perhaps inferior to themselves and ready to welcome a missionary enterprise. Yet across the gulf of space, minds that are to our minds as ours are to those of the beasts that perish, intellects vast and cool and unsympathetic, regarded this earth with envious eyes, and slowly and surely drew their plans against us. And early in the twentieth century came the great disillusionment.

Studies at NASA's software engineering laboratory showed that it didn't especially matter which specific test to discover defects was applied. Instead, what mattered was using a combination of tests and different inspectors. Different people spot different errors – only 20% of the defects found in inspections were found by more than one inspector.

Automation is one of the best-recognised means to make these feedback loops painless and we'll be mentioning it many times in the feedback loops we discuss in software development. The spelling checker in Microsoft Word is intended to act as an automated proof-reading loop – a red wavy line appears underneath a misspelled word as soon as you have finished typing it.

Automation is one of the best recognised means to make these feedback loops painless and we'll be mentioning it many times in the feedback loops we discuss in software development. The spellcheck function in Microsoft Word is intended to act as an automated proof-reading loop – a red wavy line appears underneath a misspelled word as soon as you have finished typing it. Of course, this can't correct a typo which is still a correct word: writing 'form' instead of 'from', for example. A human editor still needs to catch such slips although it may be that programs will get better at recognising context, doing even more of such work for us. Just a reminder though - setting up the spell check probably took several developers a long time. It was worth it because the spell check adds a lot of value to the user. Automation can take up a great deal of set up time – it's worth it if the feedback loop is valuable to you. Quark, for example, a package used by designers more than writers did not have a spell check function for many versions, and it is still not automatic.

Figure 8. Microsoft Word's spelling checker in action

Of course, this can't correct a typo that is still a correct word: writing 'form' instead of 'from', for example. A human editor still needs to catch such slips although it may be that programs will get better at recognising context, doing even more of such work for us. Just a reminder though – setting up the spell-check function probably took several developers a long time. It was worth it because the spelling checker adds a lot of value to the user. Automation can be very costly in set up time – it's worth it if the feedback loop is valuable to you. Quark, for example, a package used by designers more than writers, did not have a spell-check function for many versions, and it is still not automatic.

Whether automation is a worthwhile investment or not, a series of nested loops is necessary. We'll now go on to examine each set in more detail.

2.1. Exploration

It is tempting – very tempting – to just dive in and start building. But as soon as we begin to code, we limit our options. As soon as a developer starts typing, we might find we have inadvertently selected a programming language and deployment platform, or committed to an architecture, frameworks and design approach...

The purpose of feedback at an exploratory stage is to help guide us in our idea. In the Understanding Your Customer session, we mentioned Aardvark, a start-up which hoped to be able to help answer the types of question traditional search engines struggled with: where should I go for dinner tonight; given that climbing Mount Kinabalu nearly killed me, would I be able to cope with walking the Pennine Way? Instead of diving into preparing complex algorithms, the company employed

humans to work as a 'virtual' algorithm. A customer would fire off a question and a team of people would work out the best answer. The customer would then rate the answer. It was absurdly expensive – no-one would actually run a business that way, but it told the company a great deal about the kind of questions people might ask and the kind of service they expected in return.

Cheating ?

The point is not to spend ages examining every technology or architectural choice until we're certain we've selected the very best – that would be slow and thus counter-productive. Rather, we need to quickly create a model that provides real feedback to tell us we're heading in the right direction. The model is a tool to help us and then be discarded, not a fetishised 'mini product' for the team to fall in love with. It is a short-cut – something that technical teams can sometimes find difficult to accept. Seymour Cray, often referred to as 'father of the super-computer', began design work in the 1950s, making a series of calculations using his radial slide-rule. A more experienced engineer pointed out that by making a prototype, testing it to see where it was wrong, and then adjusting, he could save days of analysis.

The model itself must be just good enough to generate some useful feedback. There is no point creating a beautiful sprayed up Styrofoam model when a cardboard box with marker pen drawings will provide the same answer. Scott Ambler in *Agile Modeling* refers to this as 'the sweet spot, where you have modelled enough to explore and document your system effectively, but not so much that it becomes a burden that slows the project down.' Tim Brown in *Change by Design* mentions the same issue: 'prototypes should command only as much time, effort, and investment as is necessary to generate useful feedback and drive an idea forward...'

We are, in short, trying to answer a question as quickly, easily and cheaply as we can. That means the feedback does not have to be formal or technical in nature. It might involve user trials with customers playing with a model or a prototype, but it could also simply mean the internal team and a big stack of post-it notes. Ambler mentions, 'By working with other people on a model, particularly when you are

This whole paragraph is very relevant to usability testing

working with a shared-modelling technology such as a whiteboard, CRC cards, or essential modelling materials such as Post-it notes, you are obtaining near-instant feedback on your ideas.' Just to return to Seymour Cray again for a moment – when asked what computer-aided design tools he employed, he quite seriously mentioned a paper pad and a number 3 pencil.

Figure 9.   Seymour Cray's
             preferred CAD tool

When the process works well, modelling should help you see whether your approach will solve a problem or provide the benefit you had in mind. It combines the knowledge and ideas of both technical and non-technical people, because it is often as much about business propositions, or user perception and requirements as it is about how the product might work or how it could be built. Toyota are well-known for ensuring factory engineers are included in the design phase to ensure that their concerns on assembly and tooling etc. will be included before the design has been set in stone.

Activity 5 – Early feedback loops

Which of the two examples below demonstrates the earlier feedback loop?

1. A radar showing your speed as you drive past a school?

2. The arrival of a speeding ticket through your letterbox?

Now look at a piece of work that you have recently completed. Think back to the exploration stage of this piece of work, and list the reactions or comments that you had (and reacted to) in this stage. Of these, note down the earliest piece of feedback you received.

Can you describe what the results of NOT reacting to this piece of feedback might have been?

Did this piece of feedback, and your reaction to it, add value to the product?

Now, with the power of hindsight, can you think of 3 things you wish you had known which would have added value.

What questions could you have asked that would have elicited this information?

Commentary:

It has been shown in many different areas that early feedback loops work – in California, road-side radar displaying drivers' current speed reduced average speeds by 14%, whereas a visible police presence had no effect.

However, most organisations today miss opportunities for feedback at this ideas stage, and work is allowed to continue without the checks in place to ensure it is the right work.

Case study – Using paper prototypes to design a website

When Monash University decided to re-design their 'Course Finder' system for the website, they needed to make some basic design decisions.

There were over 700 courses on offer at undergraduate and postgraduate level, spread across three separate countries. The designers needed to make the search easy to do and provide the kind of useful information that the user wanted – not too broad, not too narrow. The development team were pretty sure they knew what to do – three simple dropdown menu boxes which allowed users to select an area of interest and then filter by location and course level.

The problem was that there was some feedback from an earlier project which suggested users weren't so keen on this approach, and that they struggled to know they could select more than one area of interest.

So the team decided to run a quick test using paper prototypes.

They drew out two different designs:

• Design A: three dropdown menus, each acting as a filter condition

• Design B: a list of tick boxes, followed by two dropdown menus for filtering
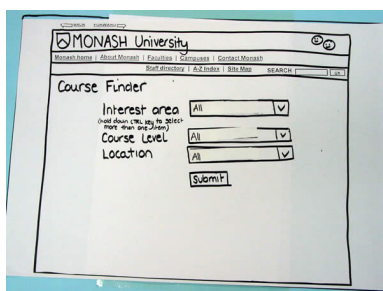
Figure 10.   Design A: three drop-down menus
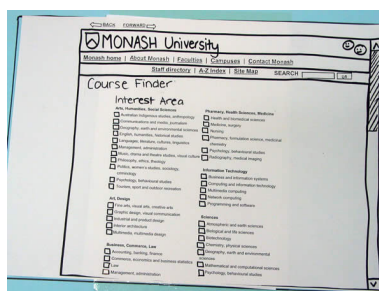
Figure 11.   Design B: A list of tick boxes

The team asked 21 representative users to come in and help them test – all prospective postgraduate and undergraduate students.

Half were shown Design A first, half were shown Design B first.

They were given two simple tasks and asked to talk aloud through their thinking as they interacted with the prototype.

Task 1: "You are interested in studying [say what you are interested in] at university. Go ahead and see if you can find some information about a course that might interest you." This was designed to exploit the differences between the two designs and reveal preferences.

Task 2: "You are interested in a career in management. Go ahead and see if you can find a course that would prepare you for a career in management". In paper prototyping it would be impossible to prepare ALL the screens that would be necessary to enable participants to access information for over 700 courses. With

this task, the user could follow an idea to its conclusion helping the team judge if the design of any of the information screens or content could be improved.

One of the team acted as a human 'computer', responding to the user's actions. So if the user selected a specific course, the computer would place a piece of paper with the specific course information on the table. If the user clicked on the 'more information' button, the human computer brought out a further piece of paper.

A facilitator then asked the user a few questions about the experience – which option she preferred, what more information she was looking for or expecting and how easy it had been to fulfil the tasks.

62% of the users preferred Design B. This was almost certainly because most (71%) users wanted to make multiple selections of the initial area of interest, and they didn't understand, or didn't read the instruction on using the CTRL key to make multiple selections in Design A. The evidence was overwhelming in terms of which design should go forwards.

The second task also revealed important gaps in what users expected. The most serious was that links at the bottom of the course overview screen were insufficiently clear which meant that several users (29%) stopped short of completing the task, not understanding where they would find the extra information they were looking for. Even at the end of the test when shown both screens, they said they couldn't see any way of moving from one to the other.

Quick and cheap, the paper prototyping revealed significant information to the development team helping them make informed decisions which they would not have got right if they had followed their gut instinct.

2.2. Coding

Just as letters are an abstract means of representing communication, so code is an abstract representation of what we're instructing the machine to do. Some people are better at quickly translating these abstracts in their heads than others. It's why when children are small their parents spell words out to ensure they don't understand: 'does she deserve an I-C-E-C-R-E-A-M?' As children get older they have become so familiar with this kind of linguistic abstraction that they can follow it instantly. Some developers are better than others at seeing abstract code as a working program in their minds, but even the very best can make mistakes or find the process too complicated.

We need to know that the code we write is going to make something happen. In the very early days of computers this meant writing something, handing it over to a special typist who would key it in onto a punch card. Two weeks elapsed before the card arrived to be put in the computer.

This is a feedback loop, but it is also a long one. If the card didn't work the programmer would clutch his hair
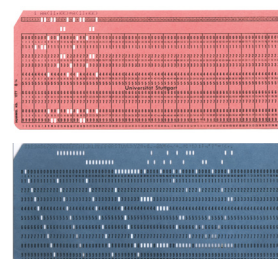

Figure 12.   Punch cards examples

and stare at the card and back at his lines of code to work out what had gone wrong. It might be human error, a little mistake in how the instructions were typed, or manufacturing error, if the hole had been punched poorly with a 'furry' edge instead of a clean cut. Either could cause a malfunction.

Twenty years ago, writing a little programme ended when you typed 'RUN' and watched to see if it worked. Nowadays, it happens almost instantly, with editors and processors checking as we type. We have tools to help us – an integrated development environment (IDE) that checks syntax as we write (some even compile your code in the background). This makes searching for any error, or making sure we comply with development standards, easier. Jeffries, Anderson and Hendrickson comment, 'XP values feedback, and there is no more important feedback than early information on how well the program works. If it were possible to have the computer beep at the programmer one second after she made a mistake, there'd be fewer mistakes in the world today.'

Actually we have become so much better at shortening the feedback loop in coding that the beeping computer is not far from reality. Admittedly, such tools may also allow humans to become slightly sloppier as a result. Those original coders used to hand-check their code to make sure it would compile, because the cost of an error was so high. Today, code inspections can still prove costly and laborious operations and we can enhance the quality of our code through practices such as pair programming.

Pair programming

Interestingly, this is not a new idea. Alan G. Carter recalls how wise programmers 'developed the habit of sitting down together and examining each other's coding sheets in minute detail before they were sent off for punching'. The point of modern pair programming is not to check the code will compile, but to provide fast feedback on the design and functionality – the crucial elements of coding that an IDE editing tool can't help with. Pair programming is expensive. It uses two programmers to do the work of one. Or at least that's what critics and worried budget controllers tend to think.

Figure 13.   Two programmers working together

Laurie Williams of the University of Utah suggested that pair programming was only 15% slower than two independent programmers, while pairs produced fewer bugs (error free code rose from 70 to 85%). Thus a slight loss in productivity was recovered by gains in quality. Deciding whether this works for your particular programming needs depends on the type of task and the experience of your developers. Pair programming is ideal where the task is challenging and calls for creativity. It encourages learning and knowledge transfer, and tends to ensure processes are followed better. Laurie Williams noted, "Two people working in a pair treat their shared time as more valuable. They tend to cut phone calls short; they don't check e-mail messages or

favourite web pages; they don't waste each other's time."

In *The Art of Agile Development*, Shore and Warden advise, "Paired programmers produce code through conversation. As you drive or navigate, think out loud. Take small, frequent design steps—test-driven development works best—and talk about your assumptions, short-term goals, general direction, and any relevant history of the feature or project. If you're confused about something, ask questions. The discussion may enlighten your partner as much as it does you."

It follows that if a pair are not talking or questioning one another, and if one seems to be watching or disengaged much of the time, that it is not working. Where tasks are simple and offer few challenges, productivity actually decreases in a pair – an effect most of us surely remember from paired work at school. Not everyone wishes to work this way, and not all pairs will get on and work creatively together.

One of the most well-known areas in which 'pair working' occurs is in commercial aircraft where there is a pilot and a co-pilot (and sometimes even a third 'flight engineer' who is also a qualified pilot). The extra cost is justified because of the importance of safety – the need for two people to both be alert, advising one another and sharing tasks, even though one is the Captain with overall responsibility. There have been times when the system has failed, precisely because the pilots stop offering feedback to one another.

In 2009, Air France flight 447 crashed into the Atlantic with the total loss of all crew and passengers. The blackbox recorder revealed that the most junior pilot was in control. He had asked for advice without receiving any definitive answer from his more senior co-pilot. At the same time the 'feedback' both pilots were receiving from their instruments was confused, leaving them unsure which pieces of data to trust. The side-stick controls, which the pilot pulled back on (sending the nose up and stalling the plane), did not provide visual or sensory feedback. Indeed, because the aircraft systems judged the data 'invalid' pulling back on the throttle stopped the stall alarm, while pushing forward (the correct move) started it. Thus the mechanical feedback appeared to be in conflict with the actual events.

Figure 14.   The cockpit of an AirBus A380 with the side-sticks

As the airline investigator and safety expert C. B. Sullenberger wrote, 'We need to look at it from a systems approach, a human/technology system that has to work together. This involves aircraft design and certification, training and human factors. If you look at the human factors alone, then you're missing half or two-thirds of the total system failure...'

Pair programming works side by side with mechanical and technical tests to help you improve quality. The best results occur when you use them to reinforce one another.

2.3.   Unit testing

As soon as we know that our code is correct (that is, actually working), we can start to pay attention to questions about whether the code we're writing is behaving as intended. This used to be done by a developer repeatedly running a piece of code, checking the outcomes were correct and debugging on encountering any issues. The process was: write all the code; test the full programme.
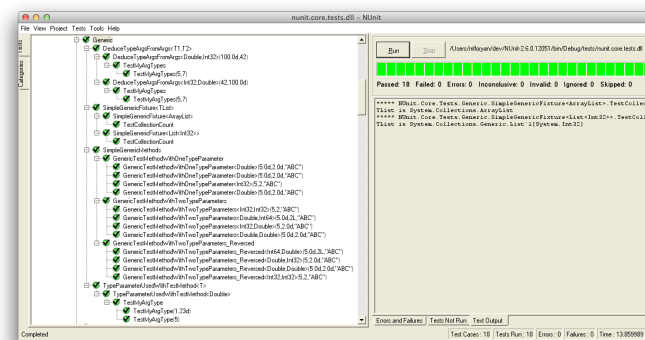
Figure 15.   A suite of Unit Tests for the NUnit framework

Today many developers frequently use frameworks such as JUnit, NUnit or xUnit. The theory behind Unit Testing is simple – rather than waiting until the end, let's test each small and independent unit of code and feed back the results. All functions and methods should have their own test case so as to trace any issues easily. Mike Cohn writes feelingly on the improvement in *Succeeding with Agile*, 'Automated unit tests are wonderful because they give specific data to a programmer—there is a bug and it's on line 47. Programmers have learned that the bug may really be on line 51 or 42, but it's much nicer to have an automated unit test narrow it down than it is to have a tester say, "There's a bug in how you're retrieving member records from the database," which might represent 1,000 or more lines of code.'

A unit test is a way of automating tests, but developers can go further, automating the execution of unit testing. There are even tools that run your tests in the background every time you change your code.

In recent years it has become increasingly common to turn to what Kent Beck describes as Test Driven Development (TDD). In this, the developer writes the test before the code. It may seem back-to-front, but it actually helps clarify thinking. On looking at the task, you ask yourself: what is the test that will prove this is true; or

perhaps, what is the test that will prove this is not true?

Sometimes known as Red-Green-Refactor, the process begins with writing a failing unit test (red) and then adding just enough code to allow a pass (green), before going on to make the code cleaner and perhaps a better fit to the architecture or structure (refactor). Once again a short feedback cycle increases quality by making issues easy to spot and fix. Since the tests run every few minutes (and sometimes every 30 seconds), the code's execution is always being tested – the developer only needs to check a few lines of code to find any problem.

We discussed the need for feedback on the broader level to make assumptions explicit and transparently test against them. TDD makes that concrete at the smallest level because by writing the test first, we explicitly state the assumptions – this helps throw a light on the design and structure of the code as a whole. Mike Cohn comments, 'It is appropriate to think of TDD as being as much a design practice as a programming practice. After all, the tests a programmer writes and the order in which they are written guide the design and development of a feature. A programmer doesn't create a list of 50 small unit tests and then randomly choose which to implement first. Instead each test is selected and sequenced so that the uncertainties of the feature are addressed early.'

Recently, Dan North has devised a further step known as Behaviour Driven Development (BDD), in which he seeks to link the idea of unit testing back into the non-technical initial feature that we are working on. Often what tends to happen in software is that we start off with an idea very close to the customer, 'I want the lights to come on when I press a button', and developers get quite quickly down to the equivalent of drawing circuit diagrams with voltages and logic gates. When designing, we now work hard to try and express features in ways that are meaningful to customers, 'I want to check my bank balance', so the idea is to express the technical steps that would make the feature possible in natural language. When this can be done even at the unit test level, then non-developers can have a full insight into code, and the tests can be automated. Dan North writes, 'BDD is as much about the interactions between the various people in the project as it is about the outputs of the development process.'

2.4. Integration

We have written more fully on integration in the Integration session, but here we want to stress that integration is also a feedback loop, and one that can be made smaller and more helpful.

Code does not exist in isolation – the units have to come together and work with other parts of the system that we have built. Pryce and Freeman in *Growing Object-Oriented Software* comment that, 'Writing unit tests gives us a lot of feedback about the quality of our code, and running them tells us that we haven't broken any classes—but, again, unit tests don't give us enough confidence that the system as a whole works. Integration tests fall somewhere in the middle.'

Psuedo-Psuedo code almost

System Testing

Most developers will have worked on a project where integration was left until the end and proved to be a major headache. More and more teams are switching to making the feedback loop faster – perhaps by loading all code up once a week, sometimes every day, sometimes continuously. In continuous integration, every piece of completed code is automatically added – if the build breaks, it should be easier to trace who has just uploaded a piece of code and then fix it. This is important because in large teams or on Open-Source projects you could end up with a lot of engineers looking

Figure 16.   A build breaker!

through their own code to check that they didn't commit a bug. In small teams it is less of a worrying issue, but it is an excuse for computer companies to amuse themselves with firing foam missiles, inventing horrible hats, and anything else they fancy to show who 'broke the build'. It's important that this works as a way of making visible that someone is working to fix the problem, NOT as a means of punishing or shaming them. That means it works best with teams that are functioning well – for new, nervous or distributed teams it can be a dangerous method. Fixing the build is an opportunity to learn and understand the project better – something that works especially well when pair programming is also in place because then two people are sharing the experience and can build upon one another's experience.

In a SOA service oriented architecture it is critical that Integration happens early and often. Each SOA module/ component/system is basically a unit when compared to all the others.

Case study – Continuous Integration feedback: much more than simply builds

In 2011, Energized Work, a software engineering lab, began building an MVP for a new start-up BuyaPowa.com. The concept was to create a social commerce website and back office that invited consumers to combine their spending power in order to achieve bulk discounts. If 50 people wanted a digital camera, for example, they should be able to buy it at a wholesale price. In theory, the more people who wanted a product then the bigger the potential discount.

The team practised continual integration from day one. Every piece of code checked in was executed with unit tests and integration tests by the CI server and then deployed to an internal reference environment. At the end of week one, the CI server was deploying running tested features to a secure public-facing demonstration environment on a daily basis. This meant that it could be demonstrated to investors – their feedback improved the concept, and a new version would be ready to show in a very short time for the next pitch.

The concept of BuyaPowa meant that products were being offered with a limited stock. Therefore, the system had to ensure that only the number of units on sale would be sold by reducing the stock count only after payment at PayPal had completed successfully. Tests checked there were no concurrency issues and by creating mocked-up versions of the external systems, in particular PayPal, the CI server executed high volume concurrency tests through the deployed end-to-end system. This meant that any changes that might introduce concurrency problems were automatically detected. For example, a few times subtle changes to the domain model broke expected behaviours and the CI server provided instant feedback about the problem.

Given the different integrations with external systems, it was important the team made sure the feedback was as meaningful and useful as it could be – not just a 'build is broken' alarm! When feedback revealed an external dependency had broken, for example because Facebook had changed their API, the team knew to stop and address the integration problem before moving on to develop new features. They went a step further and made sure that any break automatically generated a complete information picture of that exact point in time – error messages, log files, configuration set-ups. This meant the pair fixing the break could easily identify whether the fault was something relating to connectivity to external systems, the external systems themselves, or whether there was actually an error in the code.

For this team, continuous integration was not something simply to tick off (great, the builds didn't break this time), but an integral part of their feedback on development and customer interaction with the system.

2.5.   Acceptance testing

When you have finished building a feature, checked your code works, that it passes the tests and hasn't crashed the rest of the system you might feel like you've done plenty of work, but you still need to be sure that your code meets the requirements and acceptance criteria for the feature.

Figure 17.   Ronseal - 'Does exactly what it says on the tin'

Actually we would like to call this the Ronseal feedback loop (after a famous UK advert). Does it do what it says on the tin? If somebody asked you for a complete search function and it only works for one term, they will be disappointed. Working out the customer team's expectations is often part of the developer's job. If you are working from a specification document then it may be your responsibility to work out which scenario would signify completeness. If you are working from user stories then it is more likely that the team will have already considered explicit acceptance criteria that you can check functionality against.

Part of the function and benefit of user stories is the fact that these acceptance criteria have been communicated earlier to the development team. It avoids a situation where the development team feel that they have delivered, but the customer team had different expectations and feel disappointed. Or at the other extreme, a situation where the developers create more functionality than is strictly necessary. Besides the wasted effort, it leads to overly complicated code which is expensive to maintain.

In his book *User Stories Applied*, Mike Cohn takes as an example a job search website. The customer knows that not every company will supply data for every different field. Lots of companies, for example, like to leave the 'salary' field blank and only discuss this with the applicant. The customer may feel strongly that such a field should only appear on a job description when the information exists, but this assumption is not one that a developer is likely to make on his own. An acceptance test would make that assumption explicit – New job can be posted when salary information is missing; blank fields are not shown to the job seekers on the final posting.

Indeed one of the major benefits of setting out acceptance criteria is that they ensure we explicitly record our assumptions of what we think we're going to get. The acceptance criteria becomes a test defined by the customer pre-coding, which of course keeps it in non-technical language. This is not the same as what we want to get (a requirement), nor does it need to be perfect or detailed – because as soon as you get customer feedback in the next feedback loop (demonstration) you'll know whether you've succeeded or not.
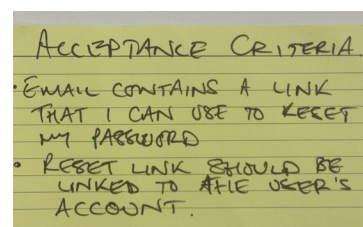

Figure 18.   Acceptance criteria written on the back of a user story card

Test vs Requirement :-)

HOW. They seem to vague to automate against

These tests can be automated against the acceptance criteria previously described. This helps keep coding to a minimum – as soon as the test passes, enough work has been done. That doesn't mean you can get rid of manual testing altogether – you almost certainly want to ensure lots of 'user-like' testers check out your work – but the process is designed to help minimise the acceptance testing phase of design – helping you deploy your code and your product into the market as soon as possible.

Agile methodologies and Scrum in particular often have a team-agreed definition of 'done'. As well as the feature meeting its acceptance criteria the team would ask a series of questions to check that the feature was genuinely ready to release. Has the code been checked, tested, integrated and is it deployment-ready? The team and project's definition may change over time or according to their specific needs, but the development team would take responsibility for a much more complete concept of 'done' than is often the case in IT.

Activity 6 – *No telepathy in the toolbox*

Part One (5 minutes)

The following experiment is for acceptance test sceptics – if you are already on board, you may want to use it to persuade others.

Imagine you are the examiner setting an arithmetic exam, and you know the answer you want to a certain question is 24.

The number 24 in this case is like the broad requirements or user story, and there are various ways you could arrive at it, and when this number is given, and you can say that the test is 'done'.

Tell a colleague that you want them to play a game which involves them guessing the number that you are thinking of. If they do not guess correctly, start reading the following list of questions out to them, and see at which stage they come up with the right answer. (They must make an attempt at the answer at each stage).

1. The answer I am looking for is a whole number

2. It is an even number

3. It is less than 50

4. It is divisible by 4

5. It is greater than 15

6. When doubled it is less than 50

7. It is divisible by 12

Did you need all the questions?

Did anyone get the answer from the first question?

How could you reduce the number of questions, but still ensure the correct result?

Part Two (20 minutes)

Take a project that you have recently worked on, and identify any problems that you think could have been avoided if acceptance testing had been in place.

Write one of these tests.

Now move on to a project that you are working on now, preferably in the early stages, and identify areas which you think could benefit from up front acceptance testing.

Write 2 of these tests, and discuss their possible value with colleagues.

Commentary:

Moving testing up front not only helps the design process, but it improves communications. Test specifications early on focus the minds of both customer and designer on what it is we are really trying to achieve.

2.6.   Demonstrated

> *"The demo is an opportunity to solicit regular feedback from the customers. Nothing speaks more clearly to stakeholders than working, usable software. Demonstrating your project makes it and your progress immediately visible and concrete. It gives stakeholders an opportunity to understand what they're getting and to change direction if they need to."*
>
> *The Art of Agile Development.* Shore, J., Warden, S.

This is probably the most famous step in all agile methodologies – the loop in which we gain feedback from our customer or customer representative. In Scrum, the demonstration is called the 'Sprint Review' and is an informal but crucial meeting in which stakeholders consider the increment of software produced during the preceding iteration and gain feedback on how it is working and whether it meets expectations or not. The product owner (or perhaps the customer herself) can then discuss with the team whether to accept the piece of work as 'done' or not.

The benefits have been discussed at length already. It enables stakeholders to know whether the project is moving along the right lines and to make decisions about continuing funding or not. The team should gain concrete feedback in order to build on for the future, and the customers should be able to ensure that they are getting exactly the functionality that they want. The more uncertain the project, whether because it is high risk or requirements are vague, the more frequent such demonstrations should be. As Ambler says in his book *Disciplined Agile Delivery*, 'Deliver working solutions frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.'

Figure 19.   Dilbert cartoon
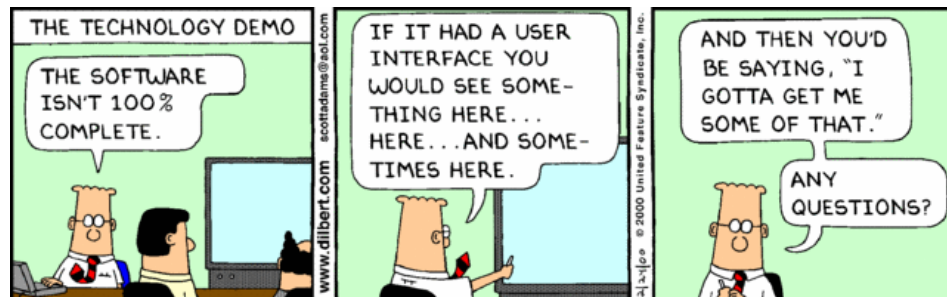
Demonstrations need to be approached in the right spirit in order to get the most from them, however. This is not a sales pitch in which the development team show off successes and try to hide failures. It is a working meeting designed to foster collaboration and in which the team needs to be actively looking for feedback that may lead to changes in order to better fulfil customers' needs.

2.7. Production readiness

How many times have you sat nervously while changes to the website go live, dreading to hear that it has crashed or that orders have stopped processing, or any one of the myriad things that could go wrong, has gone wrong?

An artificial development environment is very different to the live environment. There are many systems that work perfectly on the developer's computer, but suddenly collapse when loaded up to the live server. Maybe you hadn't taken proper account of the security firewalls; perhaps the architecture is insufficiently robust to deal with actual user numbers... whatever the issue, deployment can be a nerve-wracking time in case a launch turns into a fiasco.

A fiasco is of course a feedback loop. But just as zero sales is an expensive way to judge customer interest, finding out your software falls over when deployed to the server is an expensive and painful way of finding out that something is wrong.

Companies often make the decision to take the live environment down for a period while the new software is deployed. Banks, for example, often take down their online services overnight – which is why insomniacs who fancy a spot of online banking often find it's impossible to access accounts at 3am. Outages, as they're known, are expensive and unpopular – there's always the risk that something might go wrong which keeps the service down for much longer than anticipated. It's a bit like planned engineering works on a motorway which take place at 3am on a Sunday morning but which actually carry on until rush hour on a Monday – causing traffic misery. Problems that occur are not always technical; liaison with other departments is an essential but occasionally overlooked part of deployment – this can be as simple as ensuring that the service and system-delivery departments don't have maintenance planned on the day you wanted to launch, or it might involve contacting key customers to warn of potential problems and ensuring this is not a critical period for them.

To avoid inconvenience to our customers, lost sales or embarrassment, we put a great deal of time into trying to ensure that everything will deploy without issues. For many companies this comes in the form of a checklist. Usually the checklist captures learning from previous deployments as well as general best practice. Perhaps at our last deployment we discovered that one of our five servers did not have the most up-to-date operating system, so that our software only worked with four out of five servers. Our checklist would include an instruction to update the system on all servers the day prior to deployment.

Such checklists are important, but because dependencies in software can be very complex, we often do better if we can embed a feedback loop rather than capturing past learning. In other words, we do best if we can test our software in a realistic environment without actually risking upsetting our current system.

To do this the team tries to check issues in advance. We might try creating a virtual environment, using a simulation of the real server and database, for example, in which we model all the differing aspects as accurately as possible to check that the process of moving the code, data and setups all works correctly.

Many projects duplicate the real environment – buying two databases and servers rather than one, and keeping them in sync to ensure that our test environment exactly models the real. To what extent this is necessary depends upon the size and sensitivity of the project or how heavily regulated the environment is.

There may be aspects which cannot be modelled, however. Perhaps we will be integrating with hardware which we can't replicate or with data which only exists in a live format. No matter how sophisticated your models or simulations, only the live reality may be a sufficient test.

Deployment tests at least flag up the elements which can't be tested, allowing stakeholders to evaluate the risks and take appropriate measures. A planned outage might be deemed the only responsible way to manage deployment, or perhaps the whole development team might remain on standby overnight in case something goes wrong. By testing all the elements which can be modelled in advance, the team should be able to reduce the risk to acceptable levels.

We talked in our Integration session about a 'rule' in IT: when you find something painful, do it more often. Continuous integration is much talked about, but some teams are also moving towards continuous deployment. While requiring an increase in set up costs, the idea is to make each transaction cost much lower. A single developer can load code as a 'canary', and switch traffic over, monitoring the results. If there are no problems, then the new version becomes permanent and the old is deleted. If there are problems, the developer switches the traffic back and works out a fix, then tries again. The new and old code must run concurrently, there is no stop or start, just a streaming of traffic between them. We describe the operation in more detail in our Delivering Early and Often session.

2.8.   Monitoring

As soon as a product deploys, real data begins to provide answers to many of the questions we may have had pre-launch. Many of these, of course, are related to fundamental business questions (of which more below), but several are technical in scope, and should be dealt with almost as if they were still part of the implementation. Is the system working as we expected it to? Is it slower to respond than we thought it would be? How does it hold up with 50,000 users? Is the architecture sufficiently robust when a spike in demand occurs?
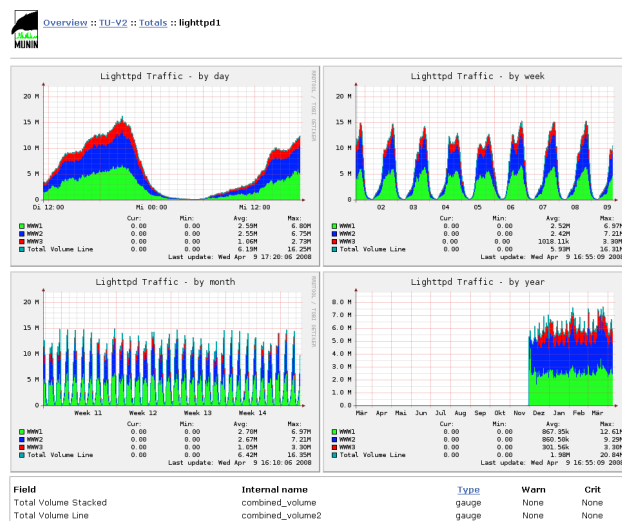
Figure 20.   Server monitoring software

This information gives us important health signals to which we need to respond. It may be that we need more disc-space or another server. Indeed, often organisations wait to see if a feature will be adopted by users and then have a plan ready to roll out that provides the extra capacity. Some of these tests can be automated – with automatic alerts when the system is in difficulty. Just as many cars are now fitted with systems that alert drivers if they have a door open, a seatbelt not on properly or have gone over the speed limit, so you may wish to set alerts to inform you when the system is nearing various boundaries.

Of course you will also use this real feedback to inform future plans. Knowing how the system responds under different stresses tells you how you might implement changes even better in the future. Consider how a diabetic takes a finger-prick blood test every day to check her blood sugar level and then medicate or change behaviour accordingly.

But primarily, monitoring as we mean it here is about a very short feedback loop – providing system operation information that the team can use to check the system is functioning as they intended it to. Whether the system is working as well as it can, or as a customer wants it to is another question.

2.9.   Concept to cash

We began by talking about the most important feedback loop of all – the actual results that you get from the marketplace. This is the highest quality feedback, because it does not come from an artificial environment (the lab, a questionnaire or a focus group). It is objective and it deals with actual behaviour, not what someone claims they might do.

For many organisations, although deployment is an anxious time, as soon as the code has been in place for a few days, the team breathes a sigh of relief and goes on to the next project. Bugs and problems may get flagged up – at least initially - but anything else is seen as belonging to a different department – maintenance or operations perhaps. The team celebrates if they hear sales are up, but that's about as far as it goes. Indeed, in many cases, a development team may cease to be a team at all once the project is live, being reassigned to new projects. This is an enormous waste. Feedback from the real environment is the most valuable feedback we have, allowing us to check all those assumptions and hypotheses against reality.

This works in every aspect of the plan – from which features have turned out to be the most popular to what capacity we thought was necessary for the system to operate at. Examining real results against assumptions teaches us to plan better for the future – have we over-engineered our system? How about that extra feature we spent so much time on – has it been used after all? When we decided to use existing storage did we make a false economy? We should also be using actual results to evaluate our initial business plan – how accurate were our assumptions on revenue, cost, time, etc.?

As well as learning for the future, we can make instant changes. A start-up may be the most obvious example, because they have to tweak their proposition continuously, but more mature organisations should be doing the same. Companies who are watching the real data closely can pull success from failure. When ExpertCity launched an interactive technical support system, their sales were pretty awful, but they observed that customers were using an unintentional piece of functionality to access their PC remotely. The company re-launched a simplified product as 'gotomypc.com' – it was a huge sales success.

Despite the importance of this loop, it is under-used by organisations. The team may have dispersed or organisations may not want to expose inaccuracies in a business case, preferring these to remain opaque. Such an attitude risks discarding the essential information we could be gathering from customers about what they use and like. This is especially ironic in a company that may pride itself on gathering feedback from customers about the quality of its customer service, for example, but ignores feedback on how customers use the product itself.

'In my end is my beginning', wrote T. S. Eliot in his poem *Four Quartets*. Just as we began and ended feedback loops in IT with concept to cash, so we need to reiterate the point we made at the beginning: feedback does not provide you with the answer, even if it may suggest a direction. That being so the shorter you make this loop – gaining real customer feedback to improve or discard your previous solution – the more you reduce your risk of failure and improve your chance of success.

2.10.   Clearing the signal

This is about how to set up the feedback you receive in the concept to cash loop so that you can use it effectively. We want to hear the useful feedback signal loud and clear from among various 'noises' that might be obscuring it. If somebody takes 20 minutes to go through your registration process then it might be that you've made it really complicated. Or they might have answered their phone half way through. How can you know?

Some feedback is really loud: if no-one buys your product then you know they don't like it. But hidden amongst that huge shout is the information you need to decipher in order to take action. Did we make it too complicated to complete a purchase? Did they hate something easy to change – like the packaging – or did they hate the very idea of what it does?

Valuable feedback requires a scientific approach. Just as we want to see how our customers actually behave rather than how they say they might behave, so we want our interpretation of what their behaviour means to be objective rather than subjective. But we don't always know what behaviour is in reaction to. If we make many small changes, how do we know which one has had a positive or negative effect? We walk a difficult line between trying to avoid the artificial but controlled environment of the laboratory, and ensuring that we don't lose the signal amongst all the noise of the marketplace .

2.10.   Clearing the signal

Case study – *Coca-Cola New*

For big manufacturers the expense and risk of a major launch means that they obsessively test each element of their product before launching it – blind taste testing, focus groups on packaging and name, questionnaires and user trials. Of course, these lab tests don't always translate as expected. Coca-Cola believed that they were losing sales to Pepsi. Pepsi, at the time, was shouting loudly that customers should 'take the Pepsi challenge' insisting that in blind taste tests, customers preferred the taste of Pepsi. So Coca-Cola set about creating a new improved flavour. They blind-tested numerous tweaked formulas with thousands of individuals and focus groups. Eventually they had a flavour that they knew consistently outperformed their own drink and also their competitors.

They launched Coca-Cola New. It bombed. It bombed so badly that some journalists claimed the whole thing was a stunt to boost sales of 'Coca-Cola Classic' (the original flavour).

What went wrong? The real problem was that none of the feedback the team had diligently gathered tested their assumption. An assumption so basic that they completely over-looked that it was an assumption at all – that taste was the most important factor. In the market-place, what became obvious was that branding mattered more. Pepsi was gaining market-share, not because their taste was better, but because they were injecting their branding with something that Coca-Cola wasn't.

It was a lesson the company learned from with admirable good sense. We mentioned earlier in the session how they took a small piece of information and launched the highly successful Coke Zero. What we didn't say was that the company also launched a second product. Aimed at the same market, it was a low sugar version. This time around the company spent less on the launch and instead waited to see which product consumers would prefer. Then they cut the dud, and poured marketing into the winner – a winner based on objective, actual sales – not on focus groups.

Advances in technology allowed the company to go even further. Freestyle vending machines were deployed to theme parks, restaurants and even universities in the USA. The machines allow flavourings to be added to the soda at the point of dispensing, meaning the machines can create up to 128 different variations. The machines instantly send data back,

Figure 21. Coca-Cola freestyle vending machines

including product sold, time of day and service information. It makes the ideal way for the company to test a new flavour and gauge demand based on actual purchase, not a score on a form.

For decades marketing managers have been fascinated by the opportunities offered by direct marketing. Because letters and brochures were sent to different individuals, it was possible to test what each person saw. Did your customers respond better to a 20% discount or to a free gift? Did you sell more if you grouped products by 'theme' (breakfast things) or by 'format' (mugs)?

You'll already guess where this is heading – the wonderful thing in the Internet age is that we can make such tests far more quickly and easily than direct mail marketeers, or even than a Freestyle vending machine. We can control what each individual entering the website sees and then observe how they act. Do they click-through this ad or that? If a pop-up with a voucher appears how much more likely are they to buy? If we change the order things appear on the page then are they more likely to register or less?

Testing works because we are able to separate out the feedback signal from the general noise through single variable testing. By changing just one thing – like greeting a returning customer with their name in huge flashing pink letters or in plain text – and splitting the returning customers to see one or the other, you can decide whether it's an improvement or not. This is known as A/B testing.

Of course you have to be slightly careful about how you split the customers – maybe women prefer flashing pink, maybe young people do, maybe only Roger from Brighton likes it... You have to be sure that you're getting a random, representative sample and in some cases it may be worth splitting by geography as well. Data analysts can help set up the tests – but it's important that developers and IT departments understand the concept and appreciate the power it hands the team to make a difference in improving the product.

This is how to use feedback to combine the rigour of scientific method with the value of real-market feedback.

Case study – *Barack Obama's 2007 campaign*

When Barack Obama began his campaign for President in 2007, the team needed to raise grass roots support in volunteers and donations. Without a huge war chest or big donor funding, their key tool was the website. The landing page was pretty important – and the key result the team wanted was to get people to register. Once they captured names and emails then follow up contact could begin to solicit donations or provide information. The current website had a static image and a sign up button and roughly 8.26% of people who visited the page registered.

So the team designed a simple test. They had four 'call to action' buttons.

- Sign Up
- Learn More
- Join Us Now
- Sign Up Now

They also had six different visual treatments for most of the page, three images, and three videos.

Figure 22.   The three images used from the six different visual treatments

Before the test, the team had a clear favourite. They ran the test, presenting each combination of button and visual treatment and measuring how many people registered. Every visitor to the site was randomly shown one of the 24 combinations. Over 300,000 people visited during the test, so about 13,000 saw each combination and the team measured the results.

The team were all convinced that a particular video was the answer. In it, Barack Obama makes a moving appeal to see America not as liberal or conservative, black or white, but as a 'United States of America'. It's a good video. But the test showed that it did worse than the current still image. Had the team changed without testing, the results would have been terrible.

The winning combination was the central image above with a 'Learn More' button. Of those seeing this combination, 11.6% registered. That's an improvement of over 40% in sign-up rate. Over the whole of the campaign about 10 million people saw the splash page. The increased likelihood of registration meant that a further 2.88 million email addresses were captured. Of all the emails sent out to those, a further 10% would be converted to volunteer – that's 288,000 volunteers extra. Every email address captured ended up donating an average of $21 throughout the campaign – that's $60 million that the Obama campaign can genuinely claim to have raised as the result of a simple A/B testing. It also helps highlight why you need to test early – making changes at the start of the campaign meant reaping the benefits immediately.

2.11.   Be greedy: you can do better

The power of the internet is allowing us to get even more sophisticated in our use of testing. Imagine that we have three different home pages – each of which is intended to lead to an action – subscribe. With every user that we direct to each page we gain an immediate piece of feedback – whether they subscribe or not. Rather than waiting for a set amount of time to make our decision about which is the highest performing, we can utilise this feedback immediately. If page A is performing better, why wait? Instead an algorithm, the 'greedy algorithm' as it is known, automatically switches an appropriate percentage of new users to the page. If the designers have new ideas based on this success, they can insert a new test page, which now has to compete for user traffic with the others. It is a highly-effective way of doing multi-variant testing, enabling you to make changes dynamically and then garner the rewards of higher performance immediately. As low-performing ideas fall down the scale, they will eventually have so few users assigned to view them that they can be deleted from the test.

2.12.   Summary

Feedback is great. No question. But amongst all the discussion regarding how and when to embed feedback loops there is sometimes a strange silence around an important aspect of feedback. It costs.

Before we are drowned by howls of protest we should be quite clear that we don't think this means you shouldn't get feedback. It's a bit like buying a house – a surveyor's report costs you money, but it doesn't cost as much as buying the house without one and then finding you've got subsidence, dry rot, rising damp and live woodworm. Nor, of course, is feedback just about avoiding disaster – it's also about discovering what customers love, improving quality and increasing value.

But the fact that feedback costs does mean that you need to pay attention to the type of feedback you seek and how frequently you seek it. As we stressed above, feedback is often about knowing when you have enough – failing early and cheaply is often a better strategy than investing a great deal in ensuring that you have a winning formula.

It is always worth considering whether the feedback loop we have embedded provides a justified economic benefit. To go back to our survey example, it may be that you're desperately in love with this adorable thatched cottage and you know it has woodworm, but you don't care. If you're going to buy it anyway, then why bother with a survey? Similarly, if you're buying at auction where you only expect one bid in every twenty to succeed, you may decide the cost of 19 surveys which turn out to be useless is not worth the risk of the one house you do succeed in buying have major problems. In both cases a particular type of feedback is not justified economically – you rely on other types of feedback (the way you feel about the cottage, the calculation of costs versus average risk). Our point is that while the general rule – gain feedback early and often – remains valid, you have to use your individual judgement to answer what type of feedback you want and how often.

Activity 7 – *Mapping your loops*

This will take 1 hour both to prepare and discuss, but is an activity that you can come back to, in order to check on feedback loops in future projects.

Look back at the feedback loops diagram.

Draw the value stream of your product development process, using the format of our Feedback Loops diagram. Make sure it includes such main areas as Idea Definition, Funding, Development, Release.

Using the feedback diagram as a guide, add the feedback loops present in your own process to your drawing.

Now, in a different colour, draw any extra loops that you think could be useful and add value.

Write in a 'key' that identifies for each loop:

• Who is involved

• What information comes back or might come back as feedback

• How much value will it add (with 0 as no value, 10 as maximum)

• What are the costs involved in getting this feedback (with 0 as no cost, 10 as maximum cost)

Discuss these with your team, and see if you can agree to implement at least some of them on future, or current, projects.

Commentary:

Visualising the feedback processes helps you see where you could usefully embed more feedback loops, or perhaps lose a few if you have too heavy a concentration in one particular area.

As Jurgen Appelo sensibly writes in *Management 3.0*, 'there is a point at which it makes no sense anymore to further reduce the length of the feedback cycle. It may be great to reduce the Scrum sprint length from four weeks to one. But reducing it to one day might not be worth the trouble. At some point performance improvement levels off and does not outweigh the added costs of increased overhead and measurements'.

This holds true throughout the many feedback loops we have discussed. We mentioned that in an exploratory phase investing too much in prototypes, etc. can cost more than the feedback is worth. Just so the set up of some unit tests or acceptance criteria can prove more costly to maintain than beneficial. System-level tests can slow down the whole process and in such cases, it can actually be cheaper to fix a problem later in the process. There is no substitute for the individual or team's intelligent judgement in such matters.

It is worth adding that although it is true that frequent feedback costs more, it does not necessarily cost more by the same order of magnitude. Don Reinertsen writes in *The Principles of Product Development Flow*: "A feedback loop that has long time-constant can only smooth variation that has equally long time-constant. This means that we often need to embed fast time-constant feedback loops inside the slow ones. These fast-time constant feedback loops do not need a large dynamic range because cumulative variation is roughly proportional to the square root of time. Thus, it takes less resource margin to maintain control for fast feedback loop."

Which is a slightly involved way of reassuring you that creating lots of little feedback loops does not actually mean we use up all our time chasing feedback rather than getting any work done. Because constant variation exists in our environment, our ability to respond swiftly to micro-changes brings benefits that outweigh any efficiency gains you might expect to see from saving all the changes up until the end.

3 MAKING FEEDBACK WORK: IT'S ABOUT PEOPLE

Most of the feedback loops we have been talking about are known as 'balancing loops'. These are where feedback acts as a check – tests, yes or no, more or less, this or that. It moves us towards a target and we remain in control of the input.

A reinforcing loop, by contrast, is where the feedback actually becomes part of the input, reinforcing a change and thus amplifying or exaggerating the effect. This was the loop which Darwin noted and discussed in sexual selection. One of the most famous examples is how the African widowbird could have evolved its long streamer-like tail. Natural selection should work against it: the long tail acts as a drag during flight and thus comes with a calorie cost. The answer is that female swallows select their mates based on tail-length. Once an initial preference for tail-length occurs, taste and tail-length become linked in a runaway loop of exponential evolution.

Figure 23.   A long-tailed widowbird in flight

The same thing can happen in our own business processes. We might use the rather more popular phrases a vicious circle or a virtuous circle – both of which are examples of a reinforcing feedback loop. You probably know that if things are going horribly within the team and you then receive poor sales results morale goes down further, meaning the quality of work decreases, which inevitably leads to even worse business results, and so on. Perhaps you've also been lucky enough to work in a team where the opposite is true and success seems to breed success.

Obviously we all want to be part of those successful, virtuous circles. Is it possible to create them? We believe so – but doing so means something very important – paying attention to the frequency of feedback required to change behaviour.

Every parent knows that you have to tell a child off immediately if they've done something wrong. You can't wait until you get home from the party and then tell little Johnny that he's not allowed to mash cake into the carpet. As we become adults our short-term memory improves (well, other than for where we left our glasses, our keys or what we've changed our screensaver password to), but some

elements don't change – in order to alter our behaviour we need immediate feedback. And by behaviour we don't simply mean being nice to people or presenting our ideas effectively, we mean acquiring skills and abilities that will improve the overall performance of the team and the system as a whole.

There are some things that we only do rarely – like buying a house or organising a wedding party. Because the feedback is infrequent, it makes it hard to learn from these. So we tend to be slow or make heavy weather of the task. A professional property developer or a wedding party planner would snap through decisions much more quickly. Schoemaker in his book *Brilliant Mistakes* calls this a 'lack of sufficient feedback' and points out that this makes it 'hard to generalize from small samples and easy to over-interpret an isolated success or error.' If the house we bought went up in value we might decide that we are brilliant property investors to whom everyone should listen, but in fact we just got lucky.

We can't create a shortcut to overall experience, but as we know, we can break projects down in order to deliberately insert feedback loops. This works with behaviour just as effectively as it does with products.

3.1.   Retrospectives

Agile insists on regular feedback cycles devoted to the process itself – to seeing how the team has functioned, rather than how the product is working or the development process is progressing. This is because agile is trying to embed a behavioural feedback loop that will lead to a positive, virtuous performance improvement cycle. Admittedly a retrospective once a month does not fulfil the need for immediate feedback in order to best change behaviour, but it is a considerable improvement upon the kind of reflection which happens in traditional project management. Think about a review held at the end of the project when it is far too late to make any changes. Even worse, consider an individual's annual performance review. Is it really sensible to save up pointing out to Dan the time you noticed him shouting down a colleague, just so you have something to talk about in the review? And if you did point it out to Dan and he hasn't done anything similar since, then shouldn't you be praising him for improvement rather than raking up an old grievance?

Figure 24.   A Retrospective in action

Team issues are often harder to deal with than a tricky coding problem, and they are also harder to raise. Having a regular 'inspect and adapt' meeting devoted to the way we are working is an excellent way to foster behaviours and learning that lead to team success. This does not mean that any issues should be saved up for the monthly retrospective, but the formal meeting tends to mean that people are more comfortable raising concerns at other times as well.

The retrospective leads to two reinforcing feedback loops. The first is that the more people see they are allowed to change the system the more they will do so until they take full responsibility for it. The second is often known as the 'Pygmalion effect': those who have high expectations set for them tend to perform better, while those who have low or negative expectations perform worse. The effect is seen in education, but also in employment. Thus, the very emphasis a retrospective places upon the team's performance tends to lead to it performing better. On the other hand, of course, if a team has a few miss-hits, keeping them positive and prepared to change things can be very difficult. In acknowledging the fear, you are more likely to help the team than by forcing a fake cheer (yay, we're still the best) that nobody quite believes in. By being open about the problem, you are also more likely to cut off the unconscious negative behaviours that others around the team may also display (don't bother giving a great project to that team, they're hopeless).

We have mentioned that feedback needs to happen quickly in order to change behaviour. This is sometimes characterised as a 'local feedback loop' – local being used in the sense of close and meaning close in time as well as geography.

> *'Local feedback loops are inherently faster than global feedback loops. It simply takes longer for a control signal to propagate through a larger system… What is the simplest thing we can do to create fast feedback in a group of humans? Increase face-to-face communications. Face-to-face communication is inherently real-time and combines both verbal and nonverbal information'*
> Don Reinertsen

As Reinertsen makes clear, humans are just as, if not more, affected by the non-verbal element of feedback. You've probably experienced a partner saying they're 'fine' or 'nothing's wrong' when every aspect of their body language and voice, tells you they're not fine, that something is very wrong indeed and you better work it out and apologise pretty darn soon unless you want to move into the doghouse. While colleagues shouldn't go in for such extremes of behaviour, it's certainly true that one of the reasons for insisting on in-person demonstrations is that you can pick up nuances of a customer's reaction. They may be saying that the feature is fine, but do they look slightly disappointed or bored? Do people seem to have lost confidence in the product? Questions like these are hard or even impossible to gauge over telephone or email, but they may be essential to the product's success.

Face-to-face feedback doesn't have to happen in a formal meeting. Sitting near colleagues who work on the same team allows for myriad informal interactions – from checking you understand exactly what the user story means to sparking ideas over coffee. As Scott Ambler puts it, 'Close proximity enables teams to reduce the overall feedback cycle, providing opportunities to address problems quickly and

cheaply and thereby increase quality.' Teams that are far distant from one another or with long gaps between opportunities for face-to-face feedback tend to have more miscommunications and misunderstandings. There is simply more time and space for 'noise' to get in the way of the communication signal.

Of course distributed teams are an inevitable part of global business, but by knowing the problems that are likely to occur, we can try to counter them. A face-to-face meeting at project kick-off, mid way point and review, for example, or video conferencing rather than sending email reports. We should not just see these contacts as 'costs'. Instead, we need to recognise that they are shortening and creating feedback loops, thus increasing the product's quality and likelihood of success.

Case study – *The postfire critique*

Retrospectives are not only a function of software or of Agile. The New York Fire Department insists on a postfire critique after every incident. Battalion Chief Frank Montagna of the New York Fire Department describes the process and objective. 'Our goal should be to learn from each mistake and to try not to repeat it. We should also teach others not to make the same mistake we made.'

The critique is explicitly about learning, not blame. While discussion is expected to be lively, it should never be hostile, the officer is responsible for directing the discussion to ensure the team stay focused on the goal – identifying the reason why the mistake happened and ensuring that anything that can be changed is changed so as not to repeat the same mistakes again.

The team begin by running over the events of the fire in turn – normally beginning with the most junior member. Each fireman discusses publicly anything that went wrong or which could have been done differently and also describes events as they struck the individual. Montagna stresses that 'it is important that the officer include himself in the exercise. Once the firefighters know the officer will admit his mistakes, they more readily will admit theirs in future critiques.'

Crucially, the postfire critique needs to be carried out straight away – either at the fire scene or back in quarters. This means incidents are still fresh in the memory. Often the team will also include the 'engine company' assigned to the firetruck in the critique in order to give each company a better understanding of the functions and needs of the other.

Finally, the 'critique' is held after perfect and successful fires as well. As Montagna says, 'Less experienced firefighters still can learn from the more experienced ones. It also provides a chance for your firefighters to brag about a job well done. Let them. They earned it.'

4   HOW WE FAIL WITH FEEDBACK

4.1.   Feedback overload

There are times when feedback can be overwhelming – of course this tends to be the case if we have not instituted sufficiently short feedback loops. Perhaps instead of monthly meetings with the customer, feedback has been gathered half way through the project. Instead of small pieces of information, a mass of feedback lands in one go. The team must consider how to address the over-arching problem that has led to the overload – normally a question of seeking feedback more frequently. This is often a difficult idea to accept. The team feels they are drowning in rework – their instinct is to hide away from the customer – but this is a mistake and will end up leading to greater changes.

That doesn't help with the problem of having a constant flow of information, which is more than the team can absorb. In *Extreme Programming Explained* Beck and Andres insist that 'If the team is ignoring important feedback; it needs to slow down, frustrating as that may be, until it can respond to the feedback. Then the team can address the underlying issues that caused the excess of feedback.'

Although a valid point in many ways, this ignores the fact that not all feedback is of the same quality. Web analytics, for example, can present mounds of data on usage. Sifting through this to decide what is useful and what is not can be overwhelming and sometimes simply take too long if external deadlines exist. The team should set up in advance what they believe the key measures to be by deciding what questions they are looking to answer. This doesn't mean that in future the team can't return to mine other information, merely that at this stage the key question is whether people are searching by 'author' more frequently than 'book title', for example. The team might also need to create a hierarchy of whose feedback will be attended to first – refusing to listen to the comments of the Sales Director's teenage daughter is easier if the team has a clear hierarchy and can show that her comments clash with those of the customer.

4.2.   Not using feedback

This is astonishingly common in individuals and organisations. We go to enormous amounts of effort to gather feedback and then we fail to use it. Sometimes this is a case of only listening to the feedback we want to hear (your product is wonderful) and ignoring the unwelcome (the search function doesn't work very well), but just as often we simply fail to implement changes.

Consider a very common example – discovering which features are used and which are not. Organisations often analyse this and decide they will learn from the information and develop more of the popular features. Do they delete the unpopular feature that nobody is using but which is adding to complexity and maintenance? No!

Activity 8 – Identifying missing feedback loops

This activity will take 30 minutes for preparation and discussion.

Without looking at the box below, see if you can list a number of feedback loops which you think are currently being missed, or not used properly. An example of this would be when you spend time and your not-inconsiderable talents in writing a feature which you release only to find no-one is using. Would earlier feedback have helped?

You may want to refer to your map of feedback loops to identify potential areas where loops are missed. Also, cast a critical eye over those loops you have in place now, which you may not be using to their full potential.

Look at the headings below. Can you match them with specific problems in your workplace?

Discuss these with your colleagues and see what suggestions you can agree on to improve this situation.

Not soliciting feedback

Getting 'too much, too late'

Not reacting to feedback

Feedback not communicated to all

Not responding to/acknowledging feedback

It is not uncommon for a company to gather huge quantities of market data and conduct a careful business review – but then make a purchasing decision based on gut instinct. Roger Martin in his article *Changing the mind of the Corporation* vividly remembers being convinced that he had persuaded a CEO not to purchase a particular company using 'bulletproof logic', only to watch amazed when the CEO bought it anyway because the price had dropped. As he put it, 'Obviously, something other than pure strategic reasoning had been poised to assert itself all along – something powerful but unacknowledged beneath the surface of our conversation, something my client was inevitably going to fall back on as soon as the right conditions presented themselves.' We discuss the way we use gut instinct in the Why Change session. Its use is very common when we aren't sure on what to listen to, on how to find the one clear signal amongst the noise. At such times, gut instinct is usually calling loudest.

Feedback then, is only useful if we are listening for it, wish to accept it and intend to use it. Otherwise, it's just more waste.

5 CONCLUSION

Feedback is designed to reduce risk, improve quality and increase the chances of success by ensuring that you know whether you are building something of value to your customer that works as they wish it to. To manage this best, wise development teams embed feedback loops within their process, discovering customer views early and often, testing their own assumptions and checking product performance at regular intervals.

You need to invest the minimum time, effort and money to answer your questions. Where possible, you should seek feedback from the marketplace rather than internal opinions or artificial results created from a simulated environment. Just as a poet could continue to polish and tweak forever, developers can find it hard to stop fiddling with code to make it more elegant, robust or perfect in some way. Running tests and gathering internal feedback should never be used as an excuse to delay getting the product out in the marketplace to gather direct customer feedback.

Feedback helps us to learn and to act on the learning. Teams that can do this are able to out-perform more brilliant, talented and experienced teams who do not keep changing and evolving speedily through the use of feedback.

Learning outcomes

Understand the importance of feedback and how it helps us

- Seek feedback early and often to ensure that the product you are building is something your customer wants; that you can build, and that continues to be appropriate in the changing environment

- Feedback enables us to manage risk when there is a great deal of change, when tasks are non-standard, and when there is uncertainty in requirements or solutions

Appreciate natural problems we have in using feedback

- Associating feedback with rework

- Confirmation bias that seeks out only feedback we want to hear

- Failure to make our assumptions and hypotheses explicit and transparent

Understand the structure of feedback loops within the software development cycle

- The over-arching concept to cash is the most important loop, but both slow and risky

- Nested feedback loops lower risk and catch more defects

- Every feedback loop should test an assumption as well as a technical function

- Automation can assist in lowering the transaction cost of each feedback loop and make them faster

Set up nested feedback loops within each part of the development process to improve responsiveness, customer satisfaction and quality as well as reducing risk by answering questions at each stage

- Exploration – is our idea valid and valuable?

- Coding – does our code compile and comply with agreed standards/rules?

- Unit testing – does our code work as we intended it to?

- Integration – will our code work with what else we and others have built?

- Acceptance – does our code do what we and our customers expected it to do?

- Demonstrated – what does our customer think of it?

- Production Readiness – are we ready to deploy our code?

- Monitoring – what happens to the system as it is used?

- Concept to cash – has our product met our and our customers' expectations? how can we make it better?

Evaluate the cost of feedback

- Feedback does have an associated cost

- Asking for feedback twice is not twice as costly as asking once

Realise how to separate valuable feedback signals from environmental 'noise'

- Use single variant or A/B testing to trial changes in order to continually improve the product and add value

- Use the 'greedy algorithm' for multi-variant testing

Apply feedback cycles to team and process as well as project using retrospectives

- Harness the power of reinforcing feedback to build a virtuous circle

- Retrospectives encourage the team to apply feedback cycle benefits to its own processes in order to learn and develop faster than direct experience alone can achieve

Identify key reasons why feedback can be misused or ignored

- We can be overwhelmed with feedback and fail to separate out the most important communication

- We can ignore feedback and rely on our own subjective, gut instinct or we fail to act on it

BIBLIOGRAPHY

Alexander, D., 2004. *Using paper prototyping as a user-centred method of informing web design decisions: a case study.* [online] Available at: <http://ausweb.scu.edu.au/aw04/papers/refereed/alexander/paper.html>. [Accessed 10 July 2012].

Ambler, S., 2002. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process.* Wiley.

Ambler, S., Lines, M., 2012. *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise.* IBM Press.

Appelo, J., 2010. *Management 3.0: Leading Agile Developers, Developing Agile Leaders.* Addison Wesley.

Argyris, C., 1999. *On Organizational Learning.* Blackwell Publishing.

Beck, K., Andres, C., 2004. *Extreme Programming Explained: Embrace Change.* 2nd Edition. Addison Wesley.

Brown, S., 2012. *Software Architecture For Developers.* Leanpub.

Brown, T., 2009. *Change by Design.* Collins Business.

Cockburn, A., 2006. *Agile Software Development: The Cooperative Game.* 2nd Edition. Addison Wesley.

Cohn, M., 2004. *User Stories Applied: For Agile Software Development.* Addison Wesley.

Cohn, M., 2005. *Agile Estimating and Planning.* Prentice Hall.

Cohn, M., 2009. *Succeeding with Agile: Software Development Using Scrum.* Addison Wesley.

Cooper, B., Vlaskovits, P. 2010. *The Entrepreneur's Guide to Customer Development: A cheat sheet to The Four Steps to the Epiphany.* Cooper-Vlaskovits.

Crispin, L., Gregory, J., 2008. *Agile Testing: A Practical Guide for Testers and Agile Teams.* Addison Wesley.

Derby, E., Larsen, D., 2006. *Agile Retrospectives: Making Good Teams Great.* Pragmatic Bookshelf.

Freeman, S., Pryce, N., 2009. *Growing Object-Oriented Software, Guided by Tests.* Addison Wesley.

Gould, S., 1982. *The Mismeasure Of Man.* W. W. Norton & Co.

Harvard Business Review, 1993. *Changing The Mind Of The Corporation.* [online] Available at: <http://hbr.org/1993/11/changing-the-mind-of-the-corporation/ar/1>. [Accessed 27 April 2012].

Harvard Business Review, 2002. *Customers As Innovators: A New Way To Create Value.* [online] Available at: <http://hbr.org/2002/04/customers-as-innovators-a-new-way-to-create-value/ar/1>. [Accessed 23 April 2012].

Humble, J., Farley, D., 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, And Deployment Automation.* Addison Wesley.

Jeffries, R., Anderson, A., Hendrickson, C., 2000. *Extreme Programming Installed.* Addison-Wesley Professional.

Kahneman, D., 2011. *Thinking, Fast And Slow.* Penguin.

Marcus, G., 2009. *Kluge: The Haphazard Evolution of the Human Mind.* Faber and Faber.

Martin, R., 1993. *Changing The Mind Of The Corporation.* [online] Available at: <http://hbr.org/1993/11/changing-the-mind-of-the-corporation/ar/1>. [Accessed 18 September 2012].

Martin, R., 2011, *The Clean Coder: A Code of Conduct for Professional Programmers.* Prentice Hall.

McConnell, S., 2004. *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press.

McConnell, S., 1996. *Rapid Development: Taming Wild Software Schedules.* Microsoft Press.

North, D. *What's In A Story?.* [online] Available at <http://dannorth.net/whats-in-a-story/>. [Accessed at 4 May 2012].

Reinertsen, D., 1998. *Managing the Design Factory: A Product Developer's Toolkit.* Simon & Schuster Ltd.

Reinertsen, D., 2009. *The Principles of Product Development Flow: Second Generation Lean Product Development.* Celeritas.

Ries, E., 2011. *The Lean Startup: How Constant Innovation Creates Radically Successful Businesses.* Portfolio Penguin.

Schwaber, K., Sutherland, J., 2012. *Software in 30 Days.* John Wiley & Sons.

Schoemaker, P. J. H., 2011. *Brilliant Mistakes: Finding Success on the Far Side of Failure.* Wharton Digital Press.

Senge, P., 2006. *The Fifth Discipline: The Art & Practice Of The Learning Organisation.* 2nd Revised Edition. Random House Business.

Shalloway, A., Beaver, G., Trott, J., 2009. *Lean-Agile Software Development: Achieving Enterprise Agility.* Addison-Wesley.

Sherwood, D., 2002. *Seeing the Forest for the Trees: A Manager's Guide to Applying Systems Thinking.* Nicholas Brealey Publishing.

Shore, J., Warden, S., 2007. *The Art of Agile Development.* O'Reilly Media.

Siroker, D. 2010. *How Obama Raised $60 Million by Running a Simple Experiment.* [online] Available at <http://blog.optimizely.com/how-obama-raised-60-million-by-running-an-exp>. [Accessed 10 July 2012].

Sloan MIT Review, 2001. *Product-Development Practices That Work: How Internet Companies Build Software.* [online] Available at: <http://hbswk.hbs.edu/item/2201.html>. [Accessed 25 April 2012].

Smith, P., Reinertsen, D., 1998. *Developing Products In Half The Time: New Rules, New Tools.* John Wiley & Sons.

Smith, P., 2007. *Flexible Product Development: Building Agility for Changing Markets.* Jossey Bass.

Vanderburg, G., 2005. *Extreme Programming Annealed.* [online] Available at: <http://vanderburg.org/Writing/xpannealed.pdf>. [Accessed 30 July 2012].

Wikipedia. *Claude Shannon.* [online] Available at: <http://en.wikipedia.org/wiki/Claude_Shannon>. [Accessed 18 September 2012].

Wikipedia. *Confirmation Bias.* [online] Available at <http://en.wikipedia.org/wiki/Confirmation_bias>. [Accessed 25 April 2012].

Wikipedia. *Lehman's Laws Of Software Evolution.* [online] Available at: <http://en.wikipedia.org/wiki/Lehman's_laws_of_software_evolution>. [Accessed 25 April 2012].

Williams, L., Kessler, R., Cunningham, W., Jeffries, R. *Strengthening The Case For Pair-Programming.* [online] Available at: <http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF>. [Accessed 18 September 2012].

Emergn Ltd.
40 Bank St, Level 18
Canary Wharf, London E14 5NR
UK/EMEA: +44 (0)20 8444 4503

Emergn Ltd.
Fitzwilliam Hall
Fitzwilliam Place, Dublin 2, Ireland

Emergn Inc.
One International Place
Suite 1400, Boston, MA 02110
North America Toll Free: +1 877 964 1981

The Chartered Institute for IT

Emergn Ltd.
40 Bank St, Level 18
Canary Wharf, London E14 5NR
UK/EMEA: +44 (0)20 8444 4503

Emergn Ltd.
Fitzwilliam Hall
Fitzwilliam Place, Dublin 2, Ireland

Emergn Inc.
One International Place
Suite 1400, Boston, MA 02110
North America Toll Free: +1 877 964 1981

www.emergn.com

www.valueflowquality.com