



Adapting Agile

This publication forms part of Value, Flow, Quality® Education. Details of this and other Emergn courses can be obtained from Emergn, 20 Harcourt Street, Dublin, D02 H364, Ireland or Emergn, 190 High St, Floor 4, 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)808 189 2043; email valueflowquality@emergn.com

Emergn Ltd.
20 Harcourt Street
Dublin, D02 H364
Ireland

Emergn Inc.
190 High St, Floor 4
Boston, MA 02110
USA

First published 2012, revised 2021 - printed 16 July 2021 (version 2.0)

Copyright © 2012 - 2021

Emergn All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher.

Emergn course materials may also be made available in electronic formats for use by students of Emergn and its partners. All rights, including copyright and related rights and database rights, in electronic course materials and their contents are owned by or licensed to Emergn, or otherwise used by Emergn as permitted by applicable law. In using electronic course materials and their contents you agree that your use will be solely for the purposes of following an Emergn course of study or otherwise as licensed by Emergn or its assigns. Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or retransmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of Emergn or in accordance with the Copyright and Related Rights Act 2000 and European Communities (Copyright and Related Rights) Regulations 2004. Edited and designed by Emergn.

Printed and bound in the United Kingdom by Apple Capital Print.

CONTENTS

Introduction 1

1 The reality of software development 3

- 1.1. Coping with what you have: Agile Waterfall 3
- 1.2. How big a problem is this? 6

2 Understanding the rules 11

- 2.1. Do we really need to adapt? 13

3 The essential rules: an extended Emergn view 19

- 3.1. Common ground 19
- 3.2. Patterns 20

4 Just do it 41

- 4.1. Just tell me where to start! 42

5 Conclusion 53

Bibliography 55

INTRODUCTION

During the Second World War, Japanese and American soldiers created bases on Melanesian islands. To support the military personnel, food, weapons and medicine were airlifted or dropped onto the islands. Some of this found its way to the local population – most of whom had never seen such goods or perhaps even aeroplanes before.

When troops left at the end of the war, the supplies naturally ended as well. Local charismatic figures (opinion is divided over whether they were genuine or manipulators) instituted ‘cargo cults’. People tried to copy the processes that had surrounded the airdrops, hoping that more supplies would follow. They cut airstrips from the jungle, built makeshift control towers from straw and timber and carved wooden headphones and guns. Naturally, no further supplies appeared.

It is a perfect example of the danger of mistaking the circumstances that surround an effect with the effect’s cause. Today, the term ‘cargo cult’ is used as a metaphor for confusing process with outcome and has been applied to science and economics, as well as to software engineering.

Buying an Issey Miyake black turtleneck will not turn you into Steve Jobs and, despite all the adverts suggesting the contrary, Air Jordan shoes will not make you play basketball like Michael Jordan. Scrum teams have succeeded in the past – will copying what they did help your team succeed?

The answer, of course, depends on whether you follow the principles that made the team successful or merely wear the practices like a badge. Standing in the same room together for 15 minutes may fulfil the requirement of a daily stand-up, but unless the underlying principle of communication is working, then there is no point in doing it. The 15 minutes might as well be spent hidden away behind individual computers.

This session examines how organisations combine elements from different methodologies to create customised and hybrid development processes that work for them. More importantly it examines the underlying principles and behaviours seen in successful implementations.



Figure 1. Cargo cult followers build a wooden and straw aeroplane in the hope of attracting food drops.

A lot of Agile advice books start to sound rather like self help manuals at this point: 'Free your spirit', 'Open your mind', 'Embrace your team'. They are half-funny, half-frightening. Fear not, this session will not follow them; it's safe to continue reading. We would simply say that there's no shortcut to success. Being Agile is not a case of group hugs in the morning and happiness ever after. It involves fighting over decisions, encountering unexpected, huge problems, making mistakes and hearing that some customers love you (but don't spend any money with you) and others think your product is rubbish. Creating the right behavioural patterns, rather than instituting processes and practices will help you deal with these and other problems.

By the end of this session you will appreciate:

The common compromises made with Agile implementation.

The likely impact of compromise on expected benefits.

The potential to adapt elements of Agile methods.

Whether adaptation is truly necessary.

Common ground between different Agile methods.

The underlying practices and principles that should remain constant.

Which elements of Agile are more superficial and can be modified.

Challenges and obstacles likely to arise when implementing the principles.

The feedback cycle of implementation: select, do, learn, adapt.

Suggestions for the most appropriate practices to begin with, depending upon your business need.

1

THE REALITY OF SOFTWARE DEVELOPMENT

1.1. Coping with what you have: Agile Waterfall

According to a survey of over 1,000 application development professionals who were using Agile, carried out in 2009 by Forrester and Dr. Dobbs, 47% claimed 'Our methodology is a helpful guide but we diverge from it in order to deliver on time'. Only 15% asserted that 'We follow our methodology closely and seldom diverge from it'.

In other words – whether methodologies are supposed to be adaptable or not – pretty much everyone adapts them anyway. Closing one's eyes to this reality is unlikely to make it go away.

One of the most common reasons for adopting a hybrid approach to development practices is that few organisations start with a blank slate. Your organisation already develops software in a certain way and there will be advantages and disadvantages to your current way of working. Since your development team does not exist in isolation, how you work now harmonises with the rest of the business in very important ways – from the rhythm at which you present results, deploy and gain funding, to the types of communication and process to which others are accustomed.

Occasionally you may be given a brand new team and told 'use whatever method you want', perhaps to test Scrum or Kanban in a pilot scheme. Even if this is the case, you will still need to interact with the rest of the organisation. It's quite likely that their planning, release, evaluation and budgeting schedules are not designed to support the type of development you have in mind. So, we often see a disconnection between a development team, who are using Scrum, for example, and the rest of the organisation whose business processes remain strictly waterfall. It can lead to frustration and misunderstanding, especially around issues such as budgeting or estimating.

Forrester describes the method as 'Water-Scrum-Fall', to explain how the Agile element exists solely in development, firmly sandwiched between traditional planning and production processes.

47%
diverge to
deliver on
time

15%
seldom
diverge

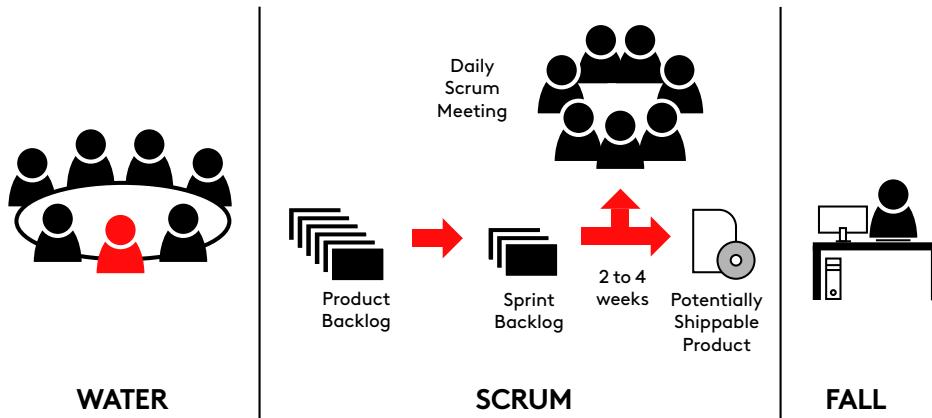


Figure 2. Water-Scrum-Fall is the reality of Agile for most organisations
(from a Forrester report by David West)

In contrast, an 'Agile' view of a project really begins at its inception – in the ideas, planning and approval process and continues right through until launch and beyond. Indeed this difference is so important that some have suggested we do away with the name 'project' altogether in Agile terms, and speak only of products. An improvement, for example, should not be seen as an isolated project, but rather as a change to the underlying product – be that system, software or hardware – in order to encourage a more holistic and long-term approach. By providing autonomy of process to the team only during the development phase, the organisation severely limits the capacity of Agile methods to have an impact.

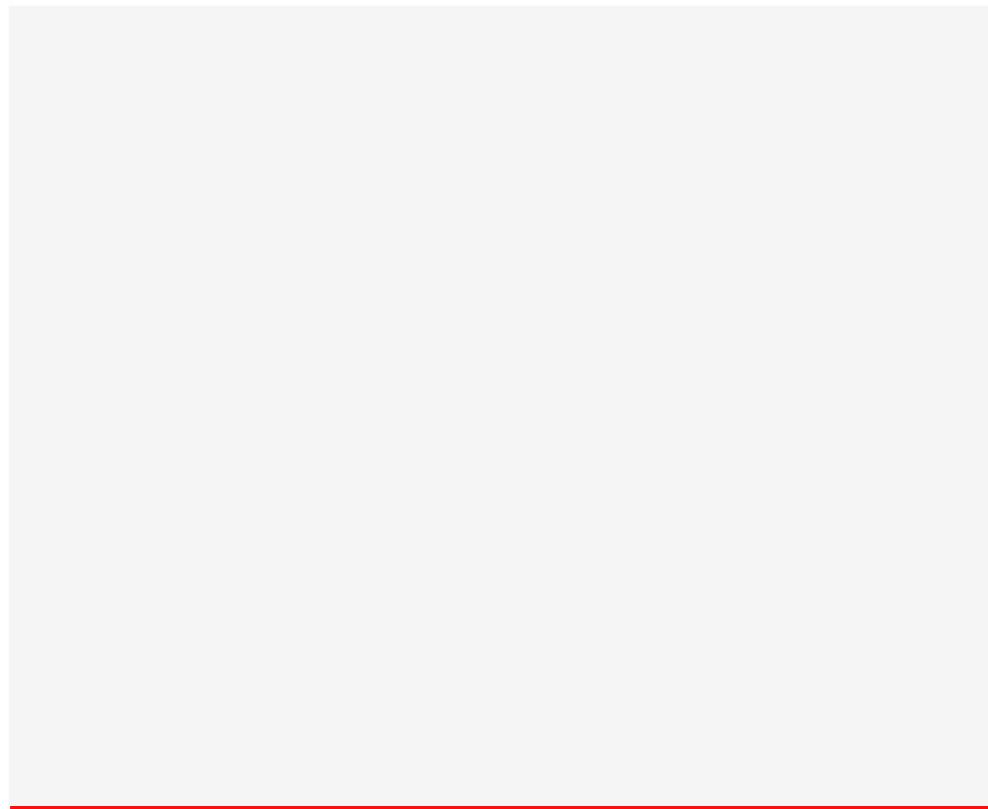
Activity 1: Predicting the pain

This activity will take 15 minutes. You can work alone or with your team.

What are the Agile processes that you think would be hardest to implement in your organisation? Are there any with which you are personally uncomfortable?

Perhaps you have already tried some practices and decided they don't work.

Write down the practices or principles that cause you the most concern – for any reason – and discuss why they are so difficult. What are the obstacles or blocks in the way of each one?



1.2. How big a problem is this?

Teams adopt Agile methods because they hope to gain benefits, not just because they fancy a change. Normally they are looking to become more responsive to changing customer demand, improve cycle time, provide a better working environment to avoid 'death marches' and burn out, and increase the quality and appropriate functionality of code. We could summarise these as looking for improvements in value, flow and quality.

Some inherited 'non-Agile' methods will prove obstacles in the path of these improvements. In most cases they will simply result in below optimal performance (perhaps disappointing business sponsors), but sometimes the obstacles may mean no gains are made at all, in spite of the significant effort and money that may have been expended.

Here are several common compromises, which can have a serious impact on success:

Teams assigned to several projects

In an effort to maximise resource utilisation, it is common for Agile teams to be assigned to several projects. We have discussed in other sessions how counter-productive this often proves. Here, we'll simply stress that it means reductions in cycle time become less likely as a team's time and energy are split.

No-one fulfils the customer role

Whether you have actual customers seated next to your team, or appoint someone to represent them, all Agile methodologies require significant customer involvement. The Scrum Product Owner, or XP Customer must have the authority and confidence to decide which are the most important features, the ability to communicate them clearly to the team and the time and commitment to work effectively with the team through the project.

Done isn't really done

Many teams are not truly cross-functional, perhaps because of the way the department is set up or due to mixed commitments. Thus at the end of a sprint, the software may still need testing or integrating. This is not 'done', and it undermines a focus on quality and speed.



Figure 3. Avoiding the software death march

The organisation requires pre-Agile re-assurance

The team may be working towards demo cycles and user stories, but the organisation may still require large quantities of documentation for governance, ask business analysts to provide detailed specifications and insist project managers report using Gantt charts, etc. In essence this requires the team to work in both an Agile and a non-Agile way, leading to significant duplication of effort.

Software is not released

Agile is enabled through short feedback loops that provide real customer input into development. This means that software should be released frequently. Organisations require the dynamic architecture and automation necessary to support the releases. Because of traditional nervousness over high-cost and high-pain releases, the business can prove an obstacle to this, insisting on compliance and governance processes that slow releases and pushes them into larger batches. Paradoxically, of course, this makes the eventual release higher risk and higher cost.

CASE STUDY: Doing Scrum and failing

IT projects fail. If you've worked in IT long enough you'll know that from personal experience. Just in case you've been worrying that it's just you, here are some statistics. The original Chaos Report in 1995 concluded that 31% of IT projects would be cancelled before they were completed, while 53% would go a staggering 189% over budget. A 2011 study by PM Solutions suggested that 37% of all IT projects fail. There is not much noticeable improvement in nearly two decades.

Given these figures it is surprising how few public analyses exist of failed projects. Almost no companies are prepared to come out and admit they took wrong decisions and wasted money. Not many individuals like to admit it either. Fortunately, a very few people, like the consultant Robin Dymond, are prepared to discuss project failures in some detail as long as they keep the company's name anonymous.

In 2007, an IT project was launched to improve a data-gathering process. The business had a process which involved the manual entry of data into a 100 row by 200 column spreadsheet. There were multiple data sources and many approvals required for each piece of information. Finally the data was re-keyed into a custom Java application which fed downstream systems.

The goal was to replace the entire process with a piece of off-the-shelf software which would allow the original data to be re-used. There would be no spreadsheet, no custom Java application and no manual retying. The result would save time and money for the business.

The project was sponsored by the Operations and IT Directors. They appointed a team, decided to use Scrum and recruited an external Scrum coach and even a few consultants. They also selected a supplier who would provide the COTS (Commercial Off-The-Shelf) application. Admittedly the COTS application did not yet have the features required, but the supplier was committed to building this enhanced functionality.

What follows is the description of events according to the Agile coach:

Iteration 1: Got initial COTS application installed, knowledgeable Product Owner from business recruited, and team put in place.

Iteration 2: Learned about development/configuration of COTS tool, built out product backlog, and investigated incremental replacement strategies for current system. Product Owner begins to doubt the capabilities of the tool and sees no reason to change a process they have used as is for 3 years. Four consultants placed on the team.

Iteration 3: Team delivers first functionality in COTS dev environment and pilots it with users. COTS tool still lacks key features required. Feedback is universally negative from business users. Sponsors make a decision to not show the app to customers for next 6 months in order to not create a bad impression with customers. The Product Owner is felt to lack vision and is replaced

by a Director. An IT Lead with little technical skill and a strong controlling personality is put on the team.

Iteration 4: Onsite vendor promises features "soon". Internal IT controls require work orders for every database change, causing the team's pace to slow. Team works around issue by adopting completely local dev setup. IT lead objects to Scrum and demands a schedule to drive towards.

Iteration 5/6: Scrum Master and Team members escalate continuing conflict with IT lead. Directors decide to wrap up contract with consulting firm. Vendor working daily with team, however key features of COTS package still not delivered. Business stakeholders question spending.

Iteration 7: Vendor delivers first version of features with numerous limitations. Directors now openly discussing shutting the project down.

Iteration 8/9/10: About 1/2 the staff roll off the project, smaller technical team continues to try make progress with tool. Project begins winding up.

What happened?

The project – quite clearly – failed.

In one sense, this is acceptable. Projects do fail. We all make mistakes and bad decisions. You have to take some losses on the chin. But this project went on for 10 iterations. According to the description above, there are several occasions on which the project could have been shut down. The cost of the project (coach, developers, supplier contract, consultants) would have been considerable.

Stopping the project after iteration 3 rather than 10 would have meant a significant saving.

So if the project was going to fail, it would have been good to fail earlier. But what could have made the project succeed? Was the IT lead who objected to Scrum right? Would a schedule have helped?

Let's start by looking at the principle features of Scrum and Agile (not the stand ups and retrospectives, which we can be sure they followed, but the reason behind such meetings). We'll discuss why these are so fundamental later in the session.

Incremental Delivery: The only measure of success is working software. The team produced very little working software. They did deliver an early version in iteration 3 – and received bad feedback. They responded by making the increment larger not smaller. No more working software is described until iteration 7.

Feedback: When the team did receive feedback, they decided to ignore it. This is not the response Scrum would advise, but it is not uncommon in many organisations. The sponsors decided not to seek any feedback for another 6 months. Normally, if you receive negative feedback, then you need to reduce your increment size, delivering a smaller element to demonstrate to the customer more quickly. Have we improved? Have we got worse? If you are consistently failing, and can't see how to improve, then this helps you decide to kill the project early and save everyone lots of time and money.

Prioritisation: If you are not delivering working software or seeking feedback then you are, almost by definition, failing to prioritise. Prioritisation depends upon the customer's perception of value – in this case the Product Owner (and business customer) failed to see the value of the project at all. They were happy with the spreadsheet and existing process. They saw no reason to change. In essence, they couldn't see why the entire project had any priority.

The COTS application and the supplier relationship clearly failed. This happens. While the Agile coach who wrote up the original case study felt that other options should have been investigated, it's not uncommon to discover situations like this. Scrum, through its emphasis on working software and feedback, ought to have highlighted problems with the supplier earlier. This should have forced a decision to kill the project or change the solution.

The case study demonstrates how easy it is to claim that the team is 'doing Scrum' and to follow the practices to the letter. Without taking on board the real principles of Agile, no benefits will accrue. Those not fully convinced by the methodology will see this as a cast-iron reason to withdraw from it. The focus shifts to judging the process, not the outcome and leads to internal divisions and politicking.

Finally, it reminds us of the dangers of sandwiching Scrum development within more traditional processes. End to end value flow did not exist within this organisation – business stakeholders failed to see the importance of the concerns of the Operations and IT Directors. Put simply, they did not feel the pain that a process caused downstream. If all Directors had taken a joint decision on the business value of this project – saving time and money – then many issues might have been avoided. It might also have meant that the Product Owner and customer at least understood the reason for the change. They could then concentrate on prioritising what functionality they required, rather than questioning the project as a whole.

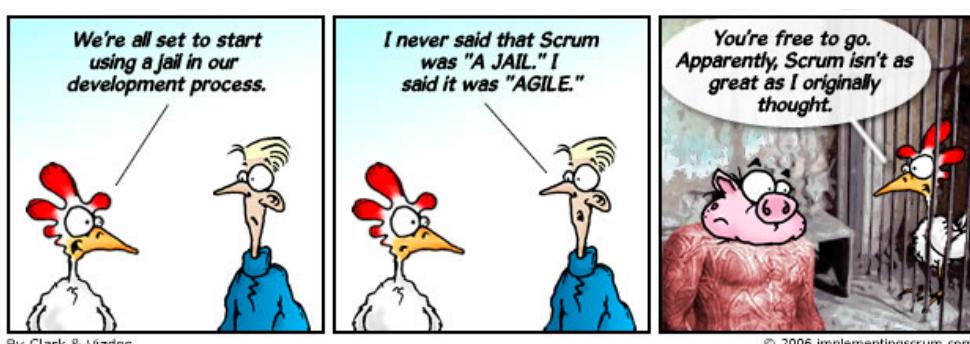


Figure 4. Disappointment in Scrum, pig and chicken style

2

UNDERSTANDING THE RULES



Activity 2: Exposing Functional Fixedness

This activity consists of two parts. In total, the activity should take 20 minutes and is best done with the whole team.

Part 1

Although this is a classroom activity, everyone needs to work independently for Part 1.

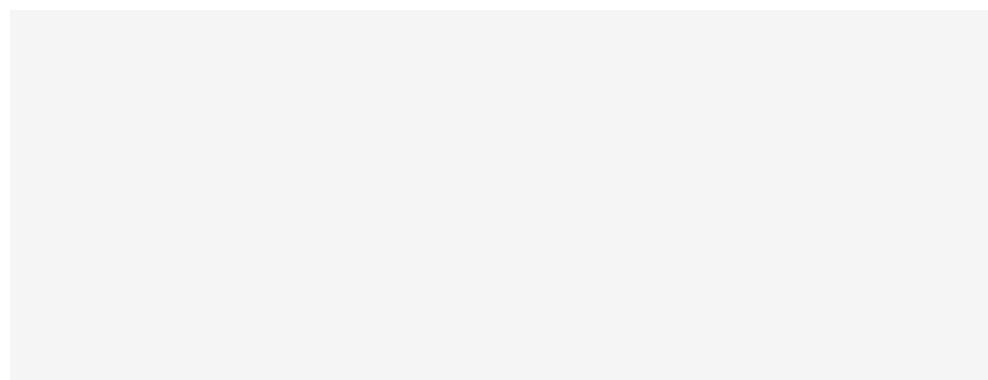
You have exactly 2 minutes to solve the problem.

You need:

- A candle
- A book of matches
- A box of drawing pins
- A cork tile on the wall

The task:

Attach the candle to the wall in such a way that when you light it, no wax will drop onto the table below.



The solution:

Empty the drawing pins out of the box, pin the box to the wall, light the candle and place it in the box (using a melted blob of wax to fix it in position if necessary).

Commentary:

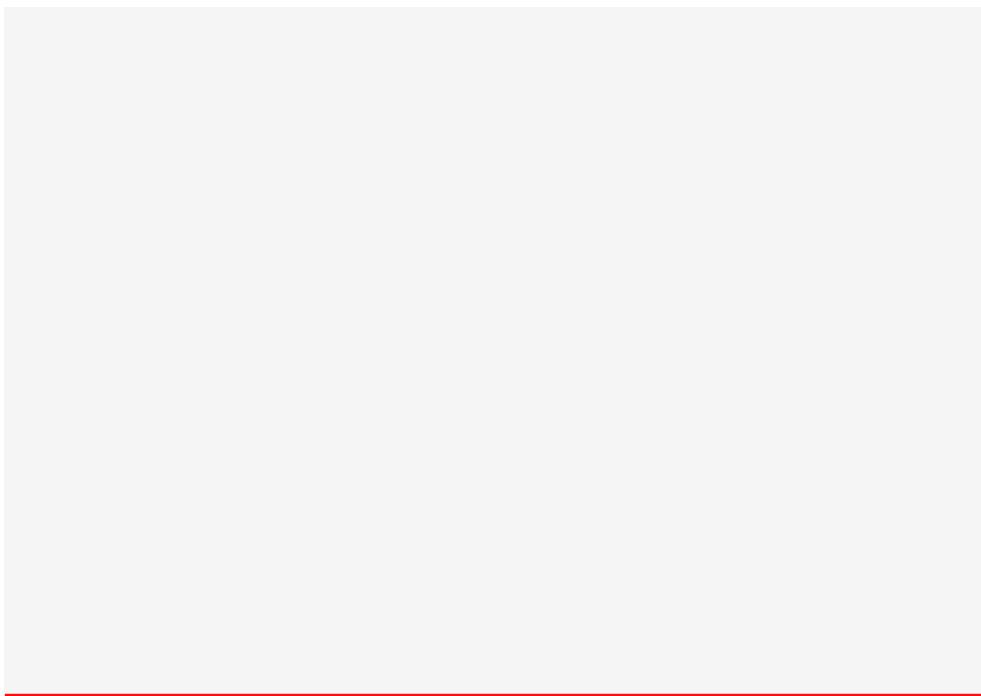
This test was created by Karl Duncker, a psychologist at Clark University, as part of a thesis on problem solving. Most people fail to grasp the solution because of a principle known as 'functional fixedness'. This predicts that because of the way the materials are presented, we see the box solely as a holder for the drawing pins, not as something to use in the task. If you present the materials differently, with the pins lying on the table alongside the box, almost everyone grasps the solution immediately.

In business, we tend to have our own functional fixedness about our organisation. There are certain 'rules' and ways of doing things that we don't question because we are so accustomed to them. Instead we challenge the new 'rules' of a methodology to which we have just been introduced. 'This element of Scrum doesn't fit with our processes', we think 'so let's change it.' We instantly adapt Scrum to fit the rules of our business, rather than wondering if we could do things the other way round.

Part 2

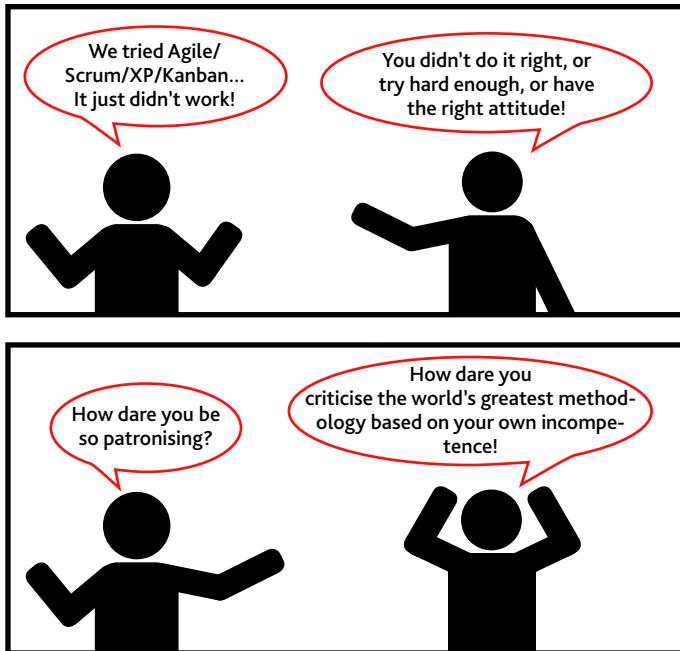
Using the practices you identified as problematic in Activity 1, ask yourself and the team to consider where Agile practices would conflict with existing business processes.

Who are the departments, people or existing processes that would be most affected by any Agile implementation? Try to list each one out so that you are aware of where the 'invisible electric fences' lie. The phrase is used by the author and product development expert, Don Reinertsen, to describe the responsibility boundaries a team can run into which can exhaust their motivation.

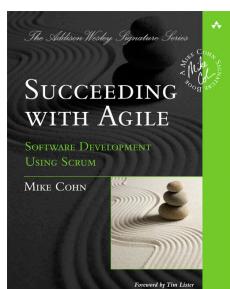


2.1. Do we really need to adapt?

There are a lot of arguments on various forums that essentially run:



Not all problems, circumstances or contexts are the same. That means no framework can truly act as a one-size-fits-all. If the methodology was written to take account of your precise circumstances then it wouldn't be applicable to anyone else. So to some extent, an element of tailoring a process is not only permissible but to be expected.



"Maybe you read a book on a [...] agile process, and it sounds perfect for your organization. In all likelihood, you're wrong. None of these processes as described by their originators is perfect for your organization. Any may be a good starting point, but you will need to tailor the process to more precisely fit the unique circumstances of your organization, individuals, and industry."

Mike Cohn, Succeeding with Agile

That's a fairly ringing endorsement of the importance of adapting processes to your unique needs. In his book Kanban, David Anderson remarks on how visitors to Corbis commented on how different all the Kanban boards were – every team seemed to have developed their own process. They were rather worried by this. The visitors presumably wanted a series of rules to say how many boxes there should be and what colour post-it notes were being used. Anderson points out that differences were necessary because the teams were developing the precise rules that worked for them.

In the Kanban board example above, the adaptations were not fundamental. The boards all adhered to the basic principle of clarity – strangers walking in should have been able to grasp what was going on with the team's work and progress. It's important to bear the distinction in mind, because while tailoring Agile to specific circumstances is entirely acceptable, it can easily be used as an excuse for not really changing and continuing old ways of working.

Before we make adaptations or ignore Agile practices we need to ask ourselves some simple questions:

Do you really need to adapt the Agile practice?

Is there a way that we can abide by the spirit of the principle?

Let's take three fairly common examples:

1. Agile emphasises interactions over documentation, but we exist in a heavily regulated industry. We have to provide extensive documentation as part of our governance in order to ensure that we are meeting our regulatory requirements.
2. XP recommends designing as you code, but we have an extremely complicated data structure. We need an extensive architecture in order to make sure that the different databases are properly linked and can be used with the many external systems.
3. Scrum demands small, co-located teams, but we work across several countries. We have a development team in India, a management team in the US and a design team in Europe. It's not practicable to co-locate these people.

We've said that it's acceptable to adapt (we'll go on to discuss fundamental principles that cannot be dropped below), but we must always ask ourselves whether we truly need to and by how much. Let's look at each business problem in turn.

Documentation is required

The question here is not that Agile rejects all documentation, but that it believes interactions are more valuable. So creating documentation should not be at the expense of personal communication.

The conversation happens first and then the decision can be made about exactly how much and what kind of documentation is needed. XP developers point out that the best documentation is the code itself. Well-written, carefully refactored code ought to be self-explanatory. This can be tested – bring a developer from another team in and ask him to read the code. Does he understand it? If not, is it because you need to make the code clearer? If the problem is in the code then rather than writing a document to explain it, you should first clarify the code. If you are trying to explain the software's purpose to a non-technical person, then documentation regarding the code is probably not going to help either. While some developers claim good code has a narrative that anyone can understand, that's probably optimistic. A short description of the software's purpose will help more, along with the acceptance criteria that prove whether that purpose has been met. Hang on! We needed to create those anyway in order to develop our product in an Agile manner, and we updated them as we clarified

our purpose using feedback ... so do we actually need anything else in terms of documentation...? Hmm...

Architecture is required up front

Once more the question is not whether some architecture is required but how much. End to end functionality, however minimal, requires decisions about architecture. In XP, a walking skeleton provides this framework. This basic architecture is a direct result of writing tests before coding. Writing tests does not mean a lack of design, in fact it tends to mean more rigour in the process. For example, if the minimum functionality required six databases to be connected, then the walking skeleton would reflect that. But if a small unit of functionality could be achieved with only two databases connected, then the skeleton would begin with just two. In the process of creating the skeleton, you might learn something important which would change the type of connection you use. You gain this learning more cheaply than if you had developed the entire architecture first.

Distributed teams

It's very common for companies to run distributed teams – you're definitely not alone. Let's consider the point of co-locating a team – the quantity and quality of spontaneous communication. When we have a team in different time zones we make it harder for them to be in frequent touch, we make delays and mistakes more likely and we need to factor in the expected waste caused by miscommunications. How important this is depends very much on what type of project you are running.

If you don't need your team to communicate informally and frequently then you can get by with installing video conferencing, web cams or running Skype on everybody's machines. If you have a highly creative project with many uncertainties then you might need to pay for regular face-to-face meetings. When you calculate the costs of such activities, you may find yourself questioning whether you could actually co-locate the team. At any rate, by looking at the problem fully you can stay focused on the principle of spontaneous communication – even if you have to drop the practice of co-location.

Activity 3: The 'Under 10' process game

This activity involves playing a game with a group of about 5-7 players. It should take about 30 minutes.

The game:

You all work for a card-picking company. The object of the game is for the company to score as low as possible. This is done over 4 rounds. In each round, each worker will pick 3 cards. These will be scored. The cards are then collected for the next round.

Company results:

If the company scores over 250 points, it will go bust. The game will end and no-one will be paid for that round.

If the company scores below 100 points at the end of round 4, it makes a big profit and a bonus box of chocolates is divided equally between all workers.

If the company scores between 100-250 points by the end of round 4 then pay is simply distributed as normal and the game ends.

Roles:

Appoint one person to be a Process Manager - they should be good at detail but also a big personality to handle this mob! Do that now. The Process Manager now appoints a loyal reliable scorekeeper and a number of workers.

The Process Manager: Your job is to ensure compliance with the rules – this means you must direct the process, check the score and reward or reprimand workers as necessary. Make those reprimands count - if the workers mess up then everyone's job is in danger! Each round that you have ensured the process has been correctly followed, your pay is 2 chocolates.

The Score Keeper: Your job is to add up the score using the rules below, pay workers as directed, and advise the Process Manager when workers have reached 3 reprimands. Your pay per round is 1 chocolate.

The Workers: Your role is to pick three cards as directed by the Process Manager. Any mistakes in the process will mean you are reprimanded. You will be paid 1 chocolate per round. If your score is below 10, then you will receive a bonus chocolate. If your score is above 20, you will be reprimanded. If you receive 3 reprimands, then any chocolates you have earned to date must be returned to the Score Keeper.

The rules:

All cards must be shuffled between rounds.

The cards must be cut by a Worker before being handed back to the Process Manager.

The pack should be spread out face down on the table.

There must always be at least 52 cards in the pack at the start of each round.

Workers must always pick up cards with their LEFT hand.

Workers should look at the card they have picked before picking the next one.

The Worker hands his 3 cards to the scorekeeper for the score to be calculated, before the next worker picks 3 cards.

The scoring:

Each card is worth its face value. Picture cards are worth 15.

An Ace is worth 0.

The Joker is worth 0.

The Queen of Spades is worth 50.

If the Queen of Spades is picked in combination with the Queen of Diamonds then both cards are worth 0.

The King of Clubs is worth -20 points.

Play:

The Process Manager should direct each Worker in turn to pick cards and see them scored. Rewards and reprimands are given and the total score calculated.

The next round begins. Keep playing until either the company goes bust or the end of round 4, when final rewards and reprimands are handed out.

Scorecard:

Workers	Reprimands	Round 1	Round 2	Round 3	Round 4
Worker 1					
Worker 2					
Worker 3					
Worker 4					
Company total score					

Commentary:

This game is based on the 'Red Bead Experiment', which was developed by the statistician W. J. Deming. It has been played by thousands of companies. The basic premise is that workers are praised or punished for outcomes over which they have no control. Rewarding the lucky and punishing the unlucky wastes everybody's time and depresses overall morale. It has been used to point out that most variation in a system is natural (94%) according to Deming, while the individual's influence is small (6%).

Don Reinertsen wrote a fairly impressive critique of the game, calling it 'an entertaining con'. The real overall objective of success for the company depends upon something very important – that the workers use their 6% of influence to change the rules. As soon as they do this, they can change the odds, so that variation changes to stack the odds in favour of the score they want.

Did your group challenge the rules?

If so, how long did it take and what sort of resistance did you face?

Barriers within the company can have their own vested interests. The Process Manager is being rewarded for keeping the process exactly as it is, it can take a major shift to make him see that he will gain more if the company succeeds and the chocolates are all shared.

Scrum comes into conflict with existing processes quite quickly – especially those around planning and approval. The temptation is to follow the path of least resistance and to give way to these processes. We frequently see a Product Owner who only has approval over the Sprint Backlog and who does not own responsibility for the ROI of the product, which is still held by the board or senior management. This has two negative effects: the pre-approval phase increases total cycle time and because the project is 'pre-approved' it becomes harder to kill or make radical changes of direction based on feedback. This is fundamentally in opposition to agility.



Figure 5. The Red Bead Experiment

3

THE ESSENTIAL RULES: AN EXTENDED EMERGN VIEW

The Emergn View

3.1. Common ground

VFQ sessions cover several of the most popular Agile methodologies, but these are not the only ones that exist. Look hard enough and you will still find proponents of Crystal, Evo, DSDM and FDD among others. Methodologies wax and wane in popularity as much as skirt lengths and hairstyles, but they often share a great deal of common ground. Indeed, most thought leaders happily acknowledge debts to one another – processes are built through learning and experience rather than dropping fully developed from the sky. Martin Fowler wrote, ‘It’s always been a world of borrowing from each other, and that’s what works well’.

Similar underlying principles form the foundations for all Agile methodologies. That’s fortunate because it means once you understand those principles you can adopt the practices that make sense and have value for your organisation. The alternative would be endlessly investing in training or coaching to keep abreast of whatever the fashion of the moment happens to be.

Here, we describe what we believe to be the essential boundaries of Agile. Following many practical years of experience, we have come across similar patterns in organisations successfully using Agile principles and practices and have distilled them into the six that we describe below. Others may disagree with our exact way of putting it or where we draw the lines, but we think they’d all agree that this lies firmly within the ‘common ground’ described by most authorities on the subject, and by most practitioners as well.

Am I doing Agile?

There’s very little value in a label. Presumably, there are various outcomes that you and your organisation are looking to improve. If you succeed in these and break every rule we have mentioned, then who cares whether you call it Agile or not! The outcome matters far more and we don’t believe anyone should feel pressured into sticking Agile into their company description.

More commonly, however, we come across organisations who are concerned that they’re not seeing the expected improvements, in spite of “doing Agile”. They want to know if the problem is with Agile itself, or with their implementation of it, and to that end they often want a sort of checklist to ask ‘am I Agile or not?’

Of course, you could consult The Agile Manifesto and check off your processes and projects against the principles and values. This may help to a certain extent. For example, if you re-read the second principle, ‘Welcome changing requirements, even late in development’, and remember that the entire team went on strike last time someone suggested a change to the UI, then this might offer you a clue.

The Manifesto was written back in 2001, and naturally the way we work has not been frozen since then. Some things have changed or developed. Several principles in the Manifesto are either vaguely defined or highly subjective.

Take the fifth principle: ‘Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.’ It might be very easy for a manager to believe this is being upheld, but for half the team to disagree vehemently.

We have a great deal of sympathy for those frustrated by this subjectivity. One of the reasons that Agile can strike some people as thrilling and others as infuriating, is because much of it is rather nebulous: a ‘mindset’, rather than a set of practices. It is easy to pay lip-service to terms like ‘transparency’ or ‘trust’ while behaving in a very different way.

That’s why this extended view includes, not only the essential rules, but also many of the questions you can ask to test whether you are really following the essential principle or not.

3.2. Patterns

1. A commitment to improve the outcomes



Figure 6. Why Change session book

What a ridiculous thing to write. Of course we want to improve the outcomes. Would we be doing this course or introducing Agile if we didn’t want to improve?

Tick.

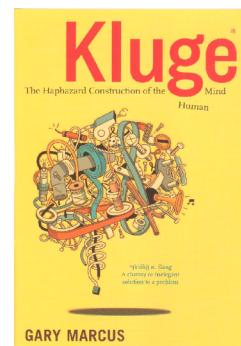
Next?

Hmmm.

Every company we have ever met sincerely believes that they

are committed to improving their outcomes. And yet, their behaviour does not always reflect that. Improvement tends to mean change, and people don’t really like change. Gary Marcus in his book on the evolution of the human mind, *Kluge*, suggests that we evolved to be attached to the familiar. This makes sense when it comes to not trying those bright red, potentially poisonous berries. It is less helpful when it comes to responding to competitors or customers.

We find it very easy to fool ourselves. That means that we have to look for evidence that an organisation is making a deliberate and concerted effort to respond to customer-driven change.



Change often means doing rather uncomfortable things. To begin with we need to ask ourselves questions and take action depending on the answers. We need to keep doing this, over and over again. In Agile it's known as 'continuous improvement'.

- Can we identify what we want to improve?
- Can we make changes, measure the results and then adjust our plan again?

A good example is found in pharmaceutical companies. They invest hundreds of man-hours and thousands of dollars in creating huge thick manuals on the best way to manufacture drugs. They distribute the manuals to their suppliers, and then the following year, they do it all over again. The pharmaceutical companies have been asking how to do things better during the year and they have found out some answers. They need to communicate these changes – even at the cost of rewriting and reprinting everything.



Figure 7. Pharmacy manuals

Risks of ignoring the pattern:

A commitment to change and improvement has a further, difficult corollary. It goes hand in hand with a willingness to give up certainty. Not knowing what's going to happen is implicit in the concept of 'change'. Numerous organisations claim to want to be Agile, but they also want to know exactly what will happen. You will have to break the news to them that crystal balls don't work.

When companies demand, 'all 300 features for £1 million in six months time', they are simply dooming themselves to disappointment. Such certainty requires a great deal of waste – both up front in planning, as teams try to ensure requirements and estimates are really cast-iron, and in padding the schedule, as they try to build in contingency time. The process rejects change – if the requirements change the plans will lose their accuracy, if something changes in the market during development then in order to react to it the schedule will slip, if problems occur during development then something will need to give (and because everyone is being measured on time and budget, what normally gives is quality, leading to greater cost in the future).

Instead, an Agile team would say, 'we think we can deliver these 3 things next month'. At the end of the month they will be in a better position to say how much more they can deliver by the following month, and you, of course, should be in a better position to know whether the project is delivering what you hoped it would deliver. This is predictability and forecasting, but it is not certainty.

Sometimes companies are so entrenched in a system that demands certainty that they find it astonishingly painful to give up. We appreciate why that's so, but we also firmly believe that without embracing this commitment to improvement and all that it implies, you cannot develop software in an Agile manner.

What to look out for:



Figure 8. The perils of best practice from Dilbert

Are you dominated by 'best practice'?

Agile practitioners ought to be very wary of the term 'best practice'. This would suggest that you have achieved perfection, or at least a state in which you can be content. It is at odds with the continual striving for improvement that should be your true pattern for working. It is also why teams should be wary of any people claiming to have the perfect methodology, toolkit or process. If they do claim this, ask when it was last updated, or the last time it was changed. Did it really arrive completely perfect? Have they ever been wrong and learnt anything?

How often do you change or update your desired outcomes?

Organisations need to update their own policies and routines regularly – if the mission statement is decided by the board, pinned up on the wall and stays there until a new CEO decides to update things by paying a branding agency to craft a new tagline, it's a pretty good sign that the company is not really committed to meaningful improvement. Most importantly, do you know what the outcomes are that you are trying to improve? Can you see how the work you do contributes towards this? If something gets in the way of the goal, would you be able to remove the conflict or block? For example, if the company is trying to improve speed of delivery, but the approvals process is holding up decisions, can the approvals process be changed? If not, then we suggest the commitment is not held very deeply.

Is the commitment demonstrated in practices?

A key practice intended to demonstrate a commitment to improvement is the retrospective. Be warned though, it is very easy simply to hold meetings once a month at which everyone lets off steam and eats chocolate cake and then goes back to work. Holding the meeting is not a sign of a commitment to improvement; analysing the team's performance, coming up with actions, carrying them out and observing the results, is.

2. Delivering early and often



Figure 9. Delivering Early and Often session book

This is pretty unambiguous. The Agile Manifesto's first and third principles are both about delivering using increments.

PRINCIPLES BEHIND THE AGILE MANIFESTO

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale...

Figure 10. First and third principle of the Agile Manifesto

Agile believes in getting pieces of working software in front of customers. This is intimately connected with the next pattern we will discuss – feedback. Incremental delivery enables us to gather feedback, by producing working software early and often.

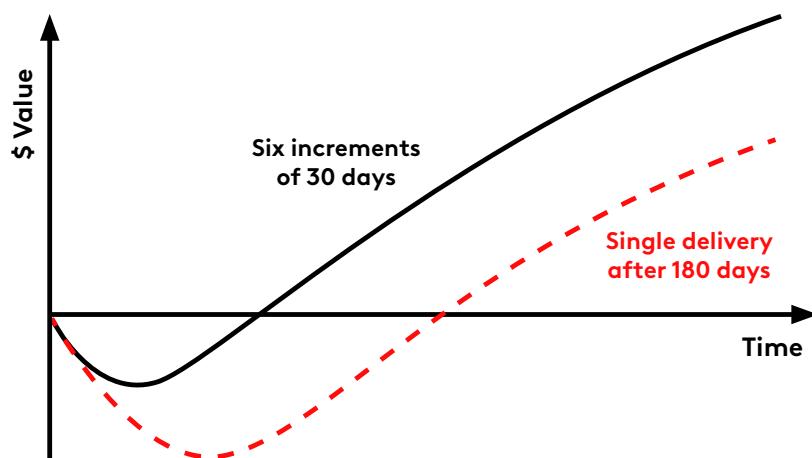


Figure 11. Earlier delivery of value through smaller increments

Increments have lots of other benefits – they generate cashflow (or savings) earlier, they reduce the risk of major changes in the external environment derailing the project and they encourage simplicity of engineering (if any of this feels like breaking news, then have a look at the Delivering Early and Often session). Most important, however, is the opportunity that incremental delivery provides to generate real and meaningful feedback from the most important people of all – customers.

What size increments?

Here's where there is space for flexibility. The Agile Manifesto suggests 'a preference to the shorter timescale', but this is open to interpretation. If you currently release software every year and you are reducing it to every 6 months, then you are on an Agile journey. As the company builds its capability, you'd expect to continue towards a shorter timescale. The really Agile element is committing to reducing the size of increments. Going as small as possible is the goal, but what 'possible' means will be different for every company. Continuous deployment does not work for everyone. In general, the riskier or less certain your project, then the more you need feedback and thus the more frequent your increments should be.

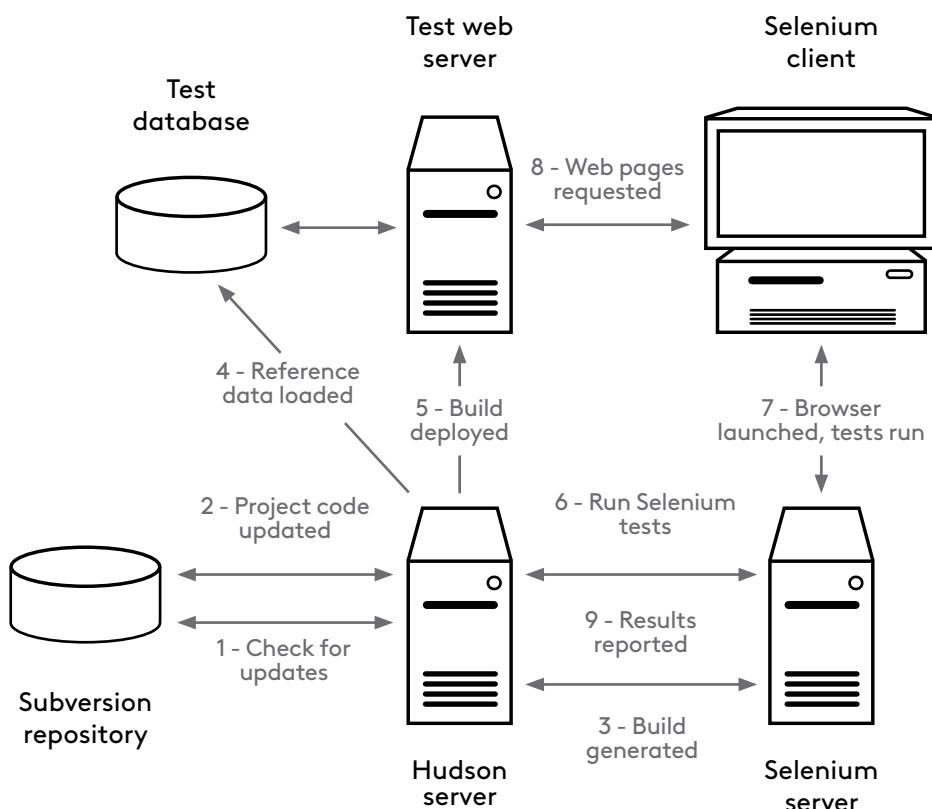


Figure 12. An example of a continuous deployment setup

Risks of ignoring the pattern:

Smaller increments can feel scary. Deciding what is valuable and then separating an independent chunk out of a complex problem riddled with dependencies is not easy. The temptation is to keep releases large – although this will increase long-term complexity and risk. Some companies feel terrified that launching early will give away their competitive edge. In fact, a company that is continually innovating can benefit from forcing competitors to play catch up.

Paul Graham, who founded the start-up Viaweb, recalls always choosing the harder of two problems. He reasoned that if his team found it difficult, his competitors would find it even harder, 'We delighted in forcing bigger, slower competitors to follow us over heavy ground'.

What to look out for:

Are you really delivering increments?

A great dodge is to claim to be delivering increments every 30 days, but actually to deliver 'parts'. An increment must be complete, offering genuine end to end value to the customer. It has to be finished – that means tested, integrated and ready to deploy. If you can't press the button and deploy the software right now, then it's not finished.

Are you able to go smaller?

One of the reasons that delivering in increments is hard is that dependencies are difficult to break down. These might be technical in nature because of the type of architecture you have. If so, is the organisation willing to invest in changing or adapting the architecture to make it easier? Alternatively, it might be internal processes that are in the way – perhaps the Finance department is horrified at the idea of launching something which does not yet permit reporting. Are people prepared to put aside normal 'requirements' in order to permit an earlier incremental release? Finally, is the organisation willing to invest in the kinds of tools that might help incremental delivery – automating integration builds, a dedicated server, etc.?

If you can't answer yes to any of these questions then the organisation's commitment to incremental delivery is questionable.

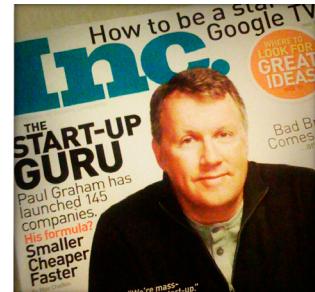


Figure 13. Paul Graham – the start-up guru

3. Feedback



Figure 14. Discovering Quality session book

Feedback is essential to Agile because it is the empirical base that powers improvement. We form a hypothesis, we examine the assumptions behind it and then we test the hypothesis. With the feedback that results from the test we adjust our hypothesis and act.

In practical terms for developing software, what this really means is that we iterate our designs and processes. Agile doesn't expect to get everything right first time, and rather than investing heavily in perfecting a specification or requirements up front, it favours the route of – build, check, rebuild. It rests on the unspoken assumption that things will change anyway, and therefore it's better to have a way of working that welcomes change.

Feedback comes in many forms and can be sought in many ways, but in essence we are always looking for what a customer will value – or to put it even more simply, how to make something that somebody wants. That's what all our feedback is for. We must combine feedback with iteration, whether that's in the detail – refactoring our code – or the big picture – evolving the design. If we don't, then feedback is just waste.

Risks of ignoring the pattern:

Short feedback loops reduce risk. The longer between feedback cycles the greater the risk that you are heading down a dead end or making a costly mistake. You ignore feedback at your peril, and moving without it is like flying blind. We have said elsewhere that customers don't always know what they want, but that doesn't mean you don't seek out feedback. Even the most innovative start ups, heading out into uncharted technologies, pay minute attention to the behaviour of their users.

What to look out for:

Do you seek out unwelcome feedback as well as the positive?

It is incredibly normal and human to reject or ignore unwelcome feedback. This does not even have to be a positive intention. If the client said, 'I hate feature x' probably you would change it. But things are rarely so clear-cut. A demonstration means showing everything to the customer as it is: there is often a temptation to turn it into a sales pitch, but this will undermine the real purpose of the meeting. You need guidance on what they like and will use, as well as what they find difficult or unhelpful. You can be clear that an early version will still require some workarounds, but only by doing so honestly will you find which elements are truly 'essential' to the customer. Taking feedback seriously also means trying to ensure demonstrations are regular and in person. You will receive much richer feedback if you are able to study body language as a customer plays with the software, as well as reading an emailed report.

Looking for full feedback also means searching out customer complaints. Do your development teams regularly meet with call-centre representatives to listen to users' bugs and annoyances? Or do these get handed off onto a routine maintenance team?

Are the teams involved in testing alternatives? The best teams are eager to try out different ways of doing things – they don't know which way is better, so they set up a test. If this is left to a Marketing Manager or Product Owner to suggest, then feedback is not fully integrated into the team.

Do you gather real customer feedback?

Scrum has a 'demonstration' at the end of each Sprint, and XP asks for a constant 'customer' to be part of the team, but these are not the only methods, or even always the best one. For a start these are not our real customers. When dealing with a proxy customer, devolving sole responsibility onto an individual and thinking that you have 'customer feedback' can be a major error. Gaining feedback from real customers can mean waiting longer than the Scrum cycle, but the team should always be agitating to get software into the hands of real users.

When feedback conflicts (as it often does), is there a hierarchy of whom should be listened to first? A range of users and customers will help ensure changes aren't being made on the basis of what the CEO's daughter said last weekend...

Do you use feedback?

It's not uncommon to gather feedback – and then just abandon it. Are you deleting unused features? Are you observing real marketplace data and using it to inform future releases or even completely different products? Of course, you can choose to ignore feedback – but this should always be a deliberate decision, not an accidental one.

4. Self-organisation



Figure 15. Teams session book

Organisations using Agile most successfully have working practices that we have summarised as 'self-organisation'. This pattern of working tends to elicit one of two responses from organisations thinking about Agile: either, 'yes of course we do that', or 'that's anarchy!'.

There is probably no principle that has been so abused. Every decade there seems to be a new slogan 'empowerment', 'autonomy', 'self organisation'. As long as they remain only slogans, they evoke nothing better than cynical laughter. Guy Browning, a comic writer for Management Today, particularly enjoys lampooning the concept: 'I've told everyone who works for me that they're 100% empowered, I've delegated all my work to them and told them only to contact me if anything they do affects the share price.'

While subscribing to the idea of self-organisation, its practical consequences horrify many companies.

'Allow them to choose who works on what? Decide who is on the team? Select their own spec? Are you insane! We need to have some proper control! Have you any idea how many expensive licenses a developer will demand if you let him?'

Self-organisation does not mean anarchy. As Jurgen Appelo points out in Management 3.0, self-organisation is the normal way that humans (and indeed the entire universe) behaves. In his words, 'Everything that management does not constrain, (and much that it attempts to), will self-organize'. The difference is that we provide context and direction for the self-organisation to work within. We don't say 'oh just do whatever you like', instead we provide a goal. How detailed that goal is depends on the organisation.

The Agile adoption of this principle is intended to answer a crucial question: are you limiting the team's capacity to respond to valuable customer demand?

Risks of ignoring this pattern:

Figure 16. President George W. Bush visiting the New York City Command and Control Center

'Command and control' is widely seen as the opposite of self-organisation. There are some valid reasons as to why it developed as a form of management – consider the spectacular successes of Roman army discipline against much larger forces who did not attack in close formation, for example. But it works in a limited set of circumstances. The Roman army commander needed to be on the field, for example. It would have been considered absurd for a 'Dux' to send back to the Emperor every time he needed to manoeuvre the army.

Approvals up and down a command chain take time and add delays to projects. Agile teams need a set of boundaries within which they can make decisions, spend money and change direction in order to complete the job more swiftly. Exactly where these boundaries lie will be different for every team, but an economic framework can make it much easier. If a project has a £200,000 cost of delay for example, the framework might authorise the team to spend up to £100,000 on specialists, extra staff or tools in order to deliver it a month early.

Self-organising teams take responsibility (and are held to account) for their own work. The opposite extreme is the worker who takes no initiative, ignoring common sense, because he was told to do it a certain way. You don't need to be an automaton for this attitude to take hold – it's very easy for boring or mundane tasks to be 'forgotten' if it's seen as someone else's responsibility to parcel them out.

Self-organisation is powered by transparency. It is the principle which both enables self-organisation, and ensures it is not abused. Agile has a series of practices designed to promote transparency in work, from daily stand-ups to task boards. The pull system, in which teams draw the next item of work, explicitly stops cherry-picking of the best work, and ensures that the dull tasks are not forgotten. This is self-evidently not chaos. A good team measures and holds itself to account, tracking cycle time, queues and flow in order to improve the working process.



Figure 17. Nothing is as motivating as your team's approval or disapproval!

Some organisations go further, using self-organisation to ensure the highest motivation and productivity by allowing individuals to choose whom they work with, when and where they work, and allowing a certain amount of time to be spent on any 'innovation' project they want. There's no requirement to adopt these specific examples, but the principle of allowing individuals and teams to choose how they solve the problem is important.

There are two major benefits. The first is that motivated individuals and teams tend to be more productive and have a lower turnover rate. The second is that self-organised teams require fewer layers of management – which reduces your overheads. Of course, the unkind amongst us might suggest this is why some mid-level managers object to the idea – they know they might be voting themselves out of a job. Consider this: Google employs 1 manager for every 20 staff, compared to an industry average of 1 to every 7.

What to look for:

When making work transparent, check your own motives

Some managers re-interpret transparency to mean that they can keep a closer eye on what developers are actually doing, rather than empowering self-organisation. There's something so suspicious about the way developers emerge to announce that feature x will take another two weeks. How can a manager know for sure? It's a bit like when a plumber purses his lips and talks about backflow valves – is he a rogue tradesman or a conscientious professional? You could spend all day watching the plumber with his arm down the toilet, but this is rather a waste of your time. Similarly, using Agile tracking as a way of providing more data for managers without the attendant autonomy, is simply a waste of everyone's time. A classic sign of this is when the task board is rarely updated, because all the 'real' reporting and task allocation occurs elsewhere.

Is control ceded to the team on the day-to-day issues that relate to the task in hand?

Who decides the coding guidelines? Do the team own the process of estimation, or are they told ‘this feature must be completed within 3 weeks’? Is there a clear definition of boundaries in terms of what the team can and cannot do? Do the team know who to ask to get answers to questions or problems, and are they able to get these responses quickly?

5. Collaboration



Figure 18. Communication, Collaboration and Coordination session book

There’s a large overlap between this pattern and that of self-organisation. Many of the outward signs of it working well (or badly) may be fairly similar. In general, Agile teams try to incorporate all the skills they need within the team – that means they are cross-functional and include end to end functionality. Traditional development passed work between groups of specialists, from analysts to developers to testers. Agile tries to build a single team containing the necessary knowledge and skills from all of these areas in order to create a product with the greatest chance of success.

This does not mean that specialists are no longer necessary. Maybe you need an expert in Ruby or an analyst with deep knowledge of the insurance industry occasionally, differing organisations will have different specialists. But we try not to be dominated by specialists, since another way of looking at this ‘specialisation’

SKILLS NEEDED TO IMPLEMENT TOP 'X' BACKLOG ITEMS						
	Test	Database	Web	Java	Domain	Change Mgmt
Dan	●	●		★		★
Mark	●		●	I'm good at Java!	★	
Paul	●	★	●	★		●
Helen	●		★	●		●
Lisa	★			●	●	●
David			★	★	●	

Note: The matrix shows various skill levels represented by dots and stars. A callout bubble for Dan says 'I can test, but I am not so good at it'. A callout bubble for Mark says 'I'm good at Java!'. A callout bubble for David says 'I won't even go near a database!'. A callout bubble for Lisa says 'I don't know CM but I'm willing to learn!'.

Figure 19. Skills matrix within a team

is the separation into departmental siloes that affects so many organisations. Instead, Agile advocates the sharing of skills, demonstrated through a blurring of roles and divisions.

In order to do this effectively we have to be transparent about decision-making and share knowledge. This may sound obvious, but in practice many organisations find it astonishingly difficult. Collaboration requires trust and a sense that everyone shares the same overall goal – that is why the goals of teams and departments must be aligned and clearly understood by all.

Jean Tabaka puts it beautifully in *Collaboration Explained*, 'The team learns how to hear difficult information from both the customer and from the IT members without fear of recrimination. They create a safe harbour in which the team can negotiate, disagree, diverge, and then converge on a collaborative solution.'

Aiding collaboration means encouraging and being prepared to invest in rich forms of communication. Creative working sessions and face-to-face meetings should be valued over email or documentation. That is, incidentally, one of the key points of the Agile Manifesto. Several others also map directly to this pattern of behaviour: 'Business people and developers must work together daily throughout the project.' and 'The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.'

Practices designed to foster collaboration include: planning meetings at which developers, business owners and product owners are all present; requirements written in a form to encourage conversation; stated sprint goals to give clarity to what the team is trying to achieve in that increment, and pair programming or other techniques designed to share knowledge.

Risks of ignoring the pattern:

The risks of a lack of collaboration should be obvious – not only in theory but because most of us have experienced it in practice. There are few people who have not come across internal politicking as departments jockey for larger budgets or greater influence. While entirely understandable, the behaviour is ultimately destructive to the interests of the organisation as a whole. Apart from anything else it is inherently wasteful – this is true of projects that are shut down early because of a lack of high level support as much as of projects that drag on for years without ever achieving the goals for which they were set up.

What to look for:

Does the team have clear boundaries?

Knowing where responsibility ends is almost as important as having any in the first place. The boundaries of a team's remit need to be clear and visible, otherwise you risk what Reinertsen describes as 'running up against invisible fences'.

Can decisions be questioned and challenged?

The transparency that fosters collaboration does not end with the project team, it expands to cover the whole planning and approval process. How are decisions made? The decisions themselves and the rules, thinking and assumptions behind them should be equally visible.

This is not the same as insisting that everyone agrees with a decision – speed often depends on quick decisions. But transparency means that when the outcome of the decision begins to be seen, the decision can be challenged, revisited and revised. This is very painful for organisations who like to keep everything (especially financial data) under wraps. Mistakes are not publicised and investigated, rather they are hidden under the carpet, while only the right decisions are talked about.

The fact is that we do best when we look at what we got wrong and when we hold one another to account. In planning, several business owners should normally be present to help keep ‘value’ estimates as objective as possible, just as several technical people help keep effort estimates reliable. Estimates can still be massively wrong, of course, but a truly collaborative organisation is not afraid to go back, consider why, and then apply learning to the next planning round. If we don’t do this, then no amount of planning games or reviews are any use. The only practice that will help in this situation is ‘classes of service’ which is essentially designed to make Product Owners feel the pain of a poor decision by charging them. While this works in some cases, when dealing with internal departments it can prove more divisive than collaborative.

Does the team share learning?

Collaborative teams try to avoid being too dependent on any one individual – no matter how brilliant. This means the team needs to invest in sharing knowledge, sometimes by having a more expert practitioner pair with someone who knows less, or by more formal investment in training. Supporting such a culture often involves ensuring the company is motivating the team rather than just individuals. There needs to be shared responsibility and reward to ensure that people are not in competition with one another, and that collaboration is actively encouraged.

Does the organisation provide tools and opportunities for collaboration?

Face-to-face meetings, regular reviews, co-located teams, wikis and other means of sharing information or feedback... these are all symptomatic of collaborative cultures. Especially with geographically distributed teams, organisations need to invest in web-cams, video conferencing and even flights to ensure some face-to-face contact.

Who is allowed to look at or change the code?

Collective Code Ownership allows anyone qualified to do so to fix or make changes to the code – quite a painful step for some developers or teams. While you need plenty of safeguards to ensure fiddling with the code doesn’t crash everything else, the emotional issues can be harder to overcome. ‘How dare you change my beautiful and elegant code with your scrappy fiddles?’ Or worse – ‘don’t let that business analyst look at the code – he thinks he can hack and now he’ll want us to change the architecture for some stupid reason!’



Figure 20. Does commitment to teams stop at a motivational poster?

6. Code is treated as an asset

Organisations successfully using Agile treat their code as an asset. As well as expending appropriate money on its creation in the first place, they go on to maintain it carefully. These companies are prepared to update and refactor their code as necessary. They ensure that the system and code-base is well-understood by a large number of staff. Even if the building or maintenance of software is

outsourced to a supplier, they check on it in much the same way that they might check on how an outsourced warehouse or delivery system is being run and maintained.

Whether this is a business creating a software product, or an organisation that relies on software to carry out its operations, the IT is treated as an integral part of the business. There is a rolling maintenance and upgrade programme which is planned and budgeted for in order to change or support legacy systems. New features are added, but changes are also regularly made to existing code. There is little or no interruption of service when this happens and the system is sufficiently robust and protected by suites of tests that any issues are reported immediately and with a thorough understanding of where the problem is. Occasionally the decision is taken not to maintain a particular piece of software but rather to run it down, knowing that a planned replacement or upgrade is due to occur. This is done in exactly the same way that a vehicle operator might decide not to service an old truck, but wait until it breaks down and then replace it.

Risks of ignoring the pattern:

You'll notice that this is the first pattern that can be said to be distinct to software development as opposed to any other area of business. It should be completely obvious. If you spend £10 million acquiring a Picasso, you will treat it as an important asset – you will guard it with an expensive alarm system, register it on international lists, insure it against fire, accidental damage and theft. Not doing so risks losing your investment. Indeed, should you not pay for an alarm, your insurer will probably refuse to cover you – calling your behaviour wilful negligence.

As businesses, we regularly spend more than £10 million on developing software, but instead of treating it as a valuable asset to be protected and cared for, we focus on short-term expediency. Our whole operations may rely on the software – if it breaks, our capacity to do business may collapse. Yet we outsource the building of it to the cheapest off-shore developers we can find, we refuse to allow face-to-face meetings with them because of the cost of flights, we decline to spend any money in training, development or specialist skills, and we run the system without maintenance.

Code which is not treated as an asset often turns out to be a liability.



Figure 21. Cars often prove a liability when not maintained

What collective madness grips our industry to behave like this? Is it because software is not understood by many people? Where a car or a machine compels our attention when it runs with a clunky noise or a splutter, does the essentially abstract and invisible nature of code blind us to the need to maintain it?

Sometimes we don't even create a decent asset in the first place. Instead, we push out something that works without any thought to its complexity, ability for change or future use. 'Wait a minute!' you say. 'Aren't you always telling us not to gold-plate? To deliver fast and just-good-enough?' This is true up to a point. You should certainly deliver a working solution, however wobbly, as a way of testing the bones of an idea. But if it turns out to be the correct idea, you shouldn't then abandon your wobbly bridge, driving ever-heavier traffic across, and simply hoping it won't collapse.



Figure 22. Technical debt credit card

Simon Baker of Energized Work wrote an essay, No Bull, which is worth referencing at some length. 'Technical debt built up in the rush to get software out the door always comes back to bite you.' The 'bite' is difficult to maintain, brittle code which may collapse without warning, requiring developers to spend their time patching, rather than delivering new value. Since the code is so poorly understood, not only does the patching take time, but there is always the risk that the patch itself will cause further problems. This becomes expensive in its own right as well as carrying with it a much larger opportunity cost. Baker writes, 'I've learned that there must be continuous investment to reduce unnecessary rework so that more of the available capacity can deliver value.' He concludes with a stark warning of the true risk of ignoring this: 'The cost of poor quality software will eventually kill a company's competitiveness.'

What to look out for:

What investment do you put into the way you build your code?

Is there a single standard of hardware for all employees, or can developers have a second monitor, a larger desk to permit pair programming? If a developer asks for a new tool can it be discussed and approved by the team or is it sent over to a central procurement purchasing hub along with a fully worked out ROI for approval?

Do you invest in new training or processes? Dojos? Coding Game wars? Tech talks? Or are these viewed as perks that can only be afforded if the company is in double digit growth?

When hiring developers do you check how they code? Do you run a practical pair programming session on some real work or have them in for a day to check what the other developers think of them? If outsourcing development, do you look for practical demonstrations of the type of product, or is it simply a case of lowest cost and a contract with heavy penalties for non-delivery?

Companies that treat code as an asset are prepared to invest in it and in those who build it. They take the work of code seriously and appreciate that a low cost for the initial delivery of code may not outweigh the life-time cost of poor quality code.

How do you treat code as it is being built?

Are you prepared to change your standards or languages depending on the need of the product? What type of documentation do you have? Are there big folders full of printed material that no-one reads or have you built interactive tutorials to ensure people know how to get started? Treating the code as a unique asset often means organisations are prepared to sacrifice a norm or standard in order to achieve specific value. Making sure people know how to use the code and deal with it afterwards also suggests that you are serious about its role in your business.

Do you refactor regularly? Do you have a suite of tests around the code that gives you confidence in making changes? Questions like these also mean you need to think about how this is built into the way developers work. If refactoring is not built into delivery time or workload, then it is easily ignored because there is more pressure to deliver 'new features' than to keep the code tidy. Since refactoring is not easy, only companies that invest in the skills that can achieve it tend to succeed.

How do you maintain code (if you do at all)? Is time for clean up, maintenance and improvement built into the team's workload and productivity assessment, or does it need to be squeezed in to slack times?

What confidence do you have in your code?

Do your developers understand the codebase? Are there whole areas of your code base that no-one dare touch in case they break it? If you bring a new developer in can he or she read it and know what's going on? If your code has declined beyond when anyone can understand it or only minor changes can be made to it, would you throw it away and start again (even if it's still working)? Is there a plan for upgrading or changing the system? Is there a contingency in place for what happens if anything goes wrong?

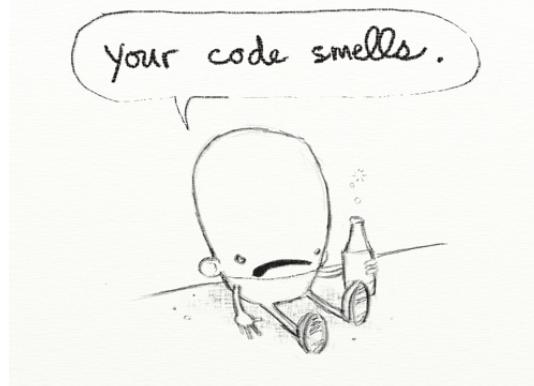


Figure 23. Honest feedback, developer style

These are crucial tests. Legacy systems filled with highly idiosyncratic code become increasingly dangerous because often only one or two people understand them. The risk to the company of these people leaving becomes increasingly severe – it's like having only one person who knows how the invoicing works, or a single individual able to access customer names. Without their specialist knowledge, it's impossible to do business. While feeling indispensable is flattering to the individual, it is disastrous for the company. Asking yourself about what happens if code collapses is also a way to help build support for necessary investment and to force others to take the issue seriously.

Finally – do you even ask yourself any of these questions? If you have never even considered such issues, then it is almost certain that your code is not being treated as an asset, but is well on its way to being a liability.

Summary

We believe that the six patterns of behaviour we have described above form the practical red lines to 'Agile'. If you can't answer 'yes' to any of the questions and you have examined your behaviours and find they don't conform to any of the good ones listed, but the risks sound all too familiar, then we suggest you have not implemented Agile.

This is painful, because such change may feel very hard. Setting up a culture in which decisions can be challenged, work is transparent and assumptions are tested and then built on is far more difficult than just putting a new process in place. Wouldn't it be lovely if a daily 15-minute meeting were all that was needed? But if you want the gains from Agile, then we genuinely believe that the questions listed above need to be asked and changes implemented.

Incidentally, not doing Agile may be just fine as far as your business is concerned. Maybe you don't want to gather feedback, and you'd rather tell your team exactly what to do and deliver in one big bang launch. No problem! As long as your company's making money and your competitors are helplessly floundering – we'll all be listening.

CASE STUDY: Good and bad Agile Google

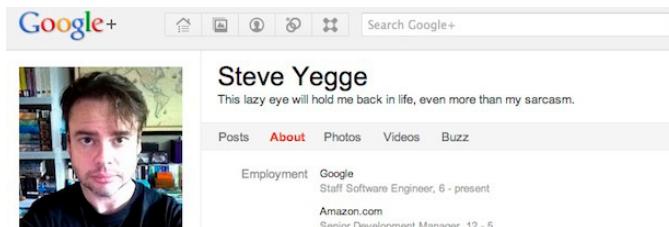


Figure 24. Steve Yegge's Google+ page

Steve Yegge is a well-known blogger (or ranter, as he sometimes calls himself). He begins one post with a rant about Agile: 'It's an idiotic fad-diet of a marketing scam making the rounds as yet another technological virus implanting itself in naïve programmers ... who believe writing [rubbish] on index cards will suddenly make software development easier'. He eviscerates XP, comparing it to Scientology and derides pair programming, before laying into Scrum.

Not the kind of person, you might think, who would readily embrace the Agile principles we've been discussing throughout this session. But you'd be wrong. He goes on to express in a 'happy rant' his love for what he calls the 'good kind of Agile', as exemplified by Google.

Yegge begins with a lengthy description of the culture of Google. He describes developers who can change teams whenever they want, who can work on what they want, and who aren't told what to do. He thinks about what motivates individuals and teams without obvious controls: the rewards and peer respect, the company-wide sharing of success. Teams are seated near one another in open office space; meetings are strictly controlled to ensure developers aren't distracted, but there are plenty of opportunities to chat and even (ssh! Yegge likes it occasionally) to pair program. Google also puts in place the discipline and control to enable practices such as collective code ownership: strict rules and guidelines, impressive unit testing and tools, and a rigorous approach to keeping the code base standard. To us, it all sounds remarkably as if Google has a serious commitment to self-organisation, backed up by being transparent in the way it manages work, decisions and rewards success. It certainly treats its code as an asset.

Next, Yegge points out that Google is driven by 'time' not 'dates'. That is, it wants to deliver software quickly, but not to a random calendar milestone. This is – in essence – a company that is prepared to give up certainty. We chose to call that a commitment to change and improvement.

Finally, Yegge mentions a priority work queue – a pile of work, ordered by importance, which is completely transparent. That's prioritisation. Whether you hold it in software or on index cards is unimportant.

Yegge has less to say about the other two principles we described above: increments and feedback. In fact he sums their use up in one catchy phrase: Google moves fast and reacts fast.

It's worth unpicking that statement a little. Fast reaction is a sign of a short feedback loop. It means you know if customers like or don't like something. Google has dozens of examples of features it has tried, changed, or just deleted. It is famous for dealing with issues quickly – it took developers a day to fix the security flaw that had allowed hackers access to private emails, for example. It has support pages dedicated to describing known problems and asking users to report new ones and where developers suggest workarounds and fixes. There are 'suggestion' pages and forums, which are deliberately set up to make it easy for customers to give feedback and report problems.

Such initiatives, coupled with 'moving fast', are inextricably linked to incremental delivery: launch, gather feedback and update. The original launch of Gmail was a good example – the company began by launching to internal users, then to about 1,000 people (who were allowed to invite family and friends). It launched to the public by invitation only, a gradual launch that allowed Google to find and fix problems as well as building new features in.

In short – judged by the patterns of behaviour we delineated above, Google is Agile. We don't need to divide it into Good Agile and Bad Agile, just point out that the company has the right attitude and practices it needs to be Agile. Even if that makes for a less entertaining rant by Steve Yegge.

Activity 4: Adapting Agile

This activity requires you to implement a change in your work. By all means enlist the support of your team to help ensure the change happens.

You want to do one thing which will help you: reduce the size of your increment, increase the frequency of your feedback and ensure you are working on the most valuable work (prioritisation).

We have 12 suggestions below – carry out one of them this week. You do not have to try these if they are not appropriate to your organisation – they are simply intended as inspiration.

Go to the Sales department and ask if there is a friendly external customer you could speak to about their requirements. Invite a customer/user in to a team meeting so the team can get a feel for what they're like, as well as what they think of proposed ideas.

Use an online survey to discover what target users most want from your product in the future. Work out how you could deliver the most requested feature within 2 weeks.

Re-prioritise the items in your queue (macro or micro level) using a quantifiable measure such as cost of delay/effort.

Ensure that all features are demonstrated to a customer before they are released.

Find one feature or element of your product that could be delivered in 2 week's time. Deliver it.

Invite people from outside your team to perform a code review on your code base.

Choose a team with whom you normally work via a handover. Take steps to improve the handover so that error rates will reduce – this might mean collaborating together on the piece of work, it might just involve a better understanding of what happens to the work downstream.

Hold a daily stand-up for a week. Keep it rigorously to 15 minutes.

Use big visible charts to show progress within the team area.

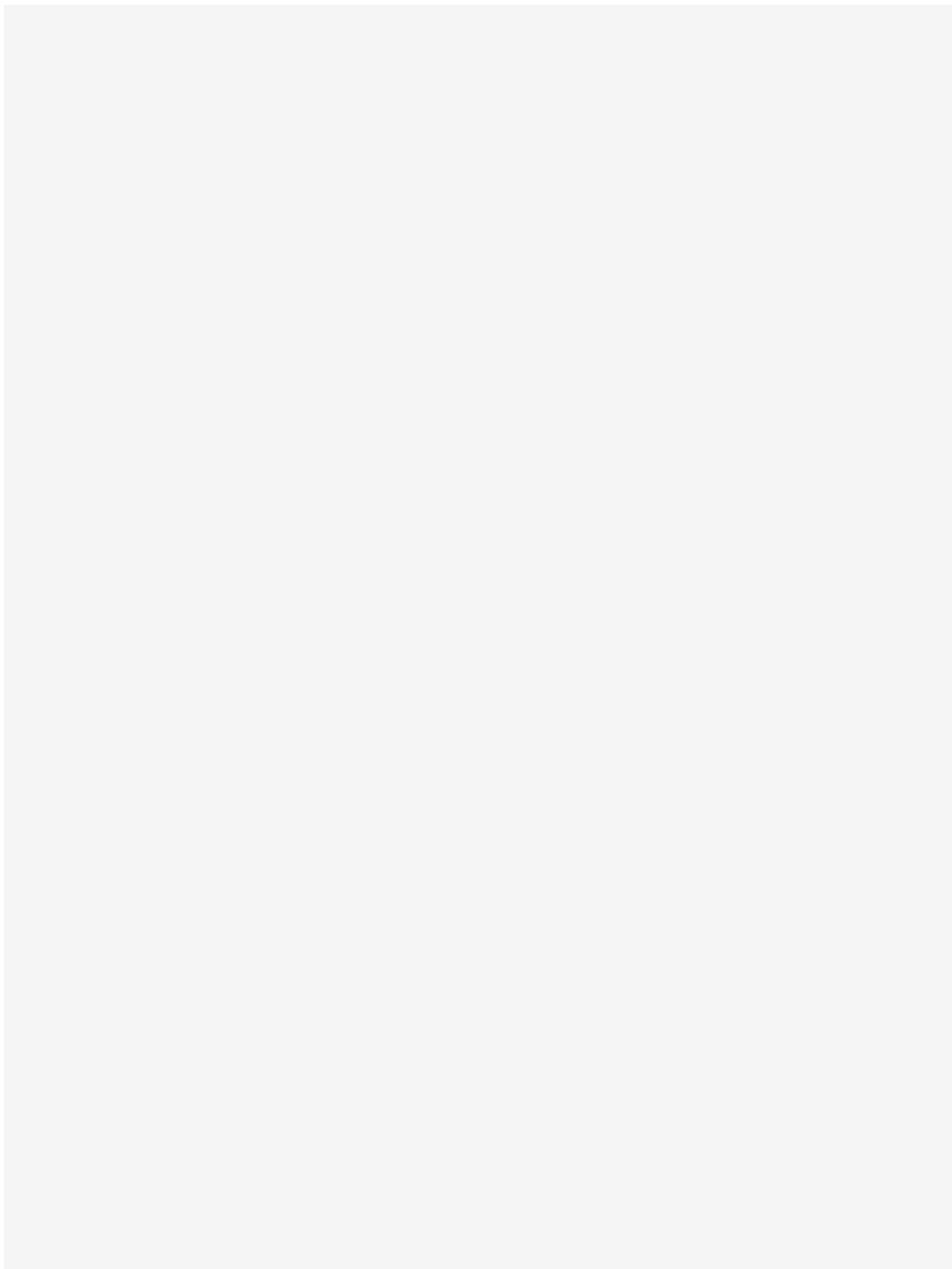
Instead of assigning tasks, allow people to choose which they'll work on.

Ask a specialist to work with someone who wouldn't usually perform that task – see if this helps spread knowledge and expertise throughout the team.

If you're part way through a project, schedule a retrospective. Ensure that at least one of the suggested changes is implemented and that you gather observations on how the improvement performed prior to the next retrospective.

Over the next 2 weeks record the reactions and changes that your selected suggestion makes. We suggest making notes against each of the potential conflict areas you listed in Activity 2. At a team meeting after 2 weeks have passed, discuss with the team what went well, where any conflicts arose and how the change could be extended or improved.

Now implement the improvements.



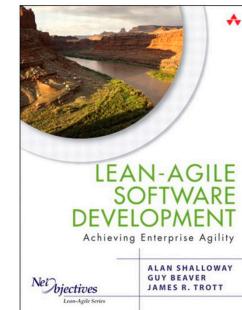
4 JUST DO IT

Given that you have accepted the foundation of Agile and given yourself the necessary permission to modify the practices that don't work perfectly for you, what next?

There is no magic wand with which you can turn the whole organisation Agile. You have to start with what you have, partial and unsatisfying as this may be. You may know that the real gains in cycle time will come with attacking 'the fuzzy front end', but until you can convince the Head of Innovation about that, you may have to be content with proving greater predictability of development cycle time.

The way the rest of the organisation works is a very real constraint. Some training sessions or books do themselves a disservice when they act as if a team can ignore it and impose Agile methods on everyone around them.

In their book *Lean-Agile Software Development: Achieving Enterprise Agility*, the authors comment that 'For Agile to become useful to the entire organization, you must consider the entire value stream: from customers to management to product enhancement to development teams to customer deployment'. This is true, but before such enterprise-wide initiatives can be considered, it is essential that Agile is shown to succeed with individual teams using it in a more limited sense.



Showing improvements in a small way will win adherents to making more changes, while announcing that you are trying a few changes is less likely to lead to the unrealistic expectations and resulting disappointment than announcing that from now on the IT department is Agile! The Team Handbook calls this approach the 'onion patch' strategy, making change efforts in areas within your control, focusing on results that can be communicated to others – such as reducing waste – and publicising the success stories.

In other words, you need to find a way to introduce a principle that will help deliver value or improve flow even if the major benefits you know could be achieved are a long way off.

Kanban explicitly insists on starting with what you have. David Anderson writes: 'When you first implement Kanban you are seeking to optimize existing processes and change the organizational culture rather than switch out existing processes for others that may provide dramatic economic improvements'. Practitioners can easily adopt a practice from Scrum or XP – pair programming or a daily 15-minute stand-up, without anyone worrying that they are about to stage a coup.

Picking where exactly to start depends upon analysing what problem you have, identifying the practices intended to deal with it and then prioritising which order to introduce them in. But there's no need to spend too long doing this. The underlying principles of inspect, adapt and using empirical evidence will help us, meaning that if we pick the wrong process or idea we can correct ourselves or improve as we go.

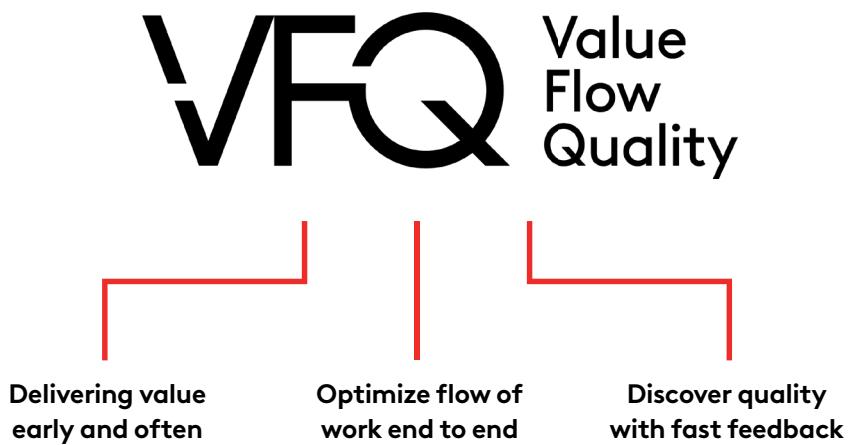
The authors of *The Art of Agile Development* summarise the idea simply: 'Start with an existing, proven method and iteratively refine it. Apply it to your situation, note where it works and doesn't, make an educated guess about how to improve, and repeat. That's what experts do.'

Processes evolve as you add new practices, and remove or adapt others. Changing the way you work should be as Agile as every other development process. As always, the team should be looking for evidence to support their decisions and this means measuring progress against certain goals. When pair programming, for example, you need to measure the speed with which features are completed and the number of errors, but you may also want to add a 'weighting' for bringing newer members of the team up to speed. The results from such measurements may guide you towards which features you wish to use pair programming on and which you don't.

4.1. Just tell me where to start!

This little section comes with a major health warning. It is intended as a guide, a suggestion of ideas that might help you get started. It is not a prescription or a method – how could it be? We don't know the exact circumstances of your business. You do. That means no-one is in a better position than you are to decide where to start and what is likely to lead to the greatest success

However, we do believe that every business should aim to:



In the following tables, we have suggested practices that correspond to the 6 patterns of successful Agile that we described above. Our suggestions begin with practices that should be relatively easy to implement and which are reversible. That is, if you start and discover that you do not deliver software faster but instead seem to deliver none at all, while costs skyrocket, then you can stop. Other practices require a greater level of investment, whether of skill, effort or money.

Moving on to these is a natural progression, when initial improvements have been made. As Kent Beck helpfully notes, 'If you begin deploying daily, for example, without getting the defect rate down close to zero you will have a disaster on your hands'. There is a second school of thought which points out that releasing buggy code will quickly focus your attention on the problem. Once you've fixed the bugs, you will have to fix the underlying problem.

In general, however, start by confirming that you've embraced the initial practices from each table and then add on others depending on results and your business needs. Each table is backed up by the practices and tools described in more detail in our Technique Library. For example, in Self-organisation, 'the team actively manages requirements' could be supported by several tools, including a task board, Kanban cards, or a cumulative flow diagram (CFD). The tables should help provide a context for when you might want to consider introducing the tools.

1. Commitment to improve the outcomes

Helps: Deliver Value, Optimise Flow and Discover Quality

Options to start:

Teams hold regular retrospectives where they capture learning and drive improvements.	All stakeholders should participate in regular retrospectives, not necessarily only the team.
Teams change and update desired outcomes during delivery.	Use double-loop feedback and make sure original goals are adapted as you learn.
Product flow is end to end.	The greatest cycle time gains come from attacking the fuzzy front end - this is recognised and organisations strive to break down functional divisions.

Practices to consider later:

The organisation as a whole updates goals and 'best practices' regularly.	Quantitative and qualitative measures should be collected and monitored to identify areas for improvement in process and practice.
---	--

2. Delivering using increments

Helps: Deliver Value

Options to start:

Increase the frequency at which software is delivered.

Your increment size can probably be smaller. Push it down until the economics no longer work. Small batches, WIP limits, engineering discipline, simple design and focus on cycle time can all help reduce increment size.

Use a prioritisation method to inform which features would be most valuable in the increment.

We list many methods in the Prioritisation session. Don't forget to ensure you define 'value' so that you capture features that are not only financially valuable but also have an information value.

Ensure that each increment has an associated business case.

Try to quantify intangible as well as tangible benefits, recording assumptions as necessary.

Practices to consider later:

Once each increment has a business case, extend this to requirements.

Do not expend too much effort – you still need to make decisions swiftly.

Adopt a prioritisation technique that uses quantified measures.

We have discussed measures such as cost/benefit ratio, while our preferred technique is weighted shortest job first (CD3).

Eliminate low value features or complete the project once the effort outweighs potential value.

Rather than focusing solely on sunk cost (what's already spent), focus on the remaining value that can be captured. Then decide whether the cost needed to do so is worth expending.

3. Feedback

Helps: Discover Quality

Options to start:

Someone with business knowledge who represents the customer is always available.

Initially this could mean the customer is available by phone or email. Eventually, you should try to seat the customer with the team. You will gain enormously if you have daily contact between the development team and real customers. No meeting or formal feedback loop can be as effective as the spontaneous opportunities for idea transfer that come from co-location.

Shorten the feedback cycle.

Requirements should be sized so that they can ideally be completed in less than 2 weeks – this ensures features are developed and ready to demonstrate quickly. Following feedback the team can iterate the design or deploy.

When a requirement is completed it is demonstrated to the customer for feedback.

Demonstrations should be regular events at which a customer representative MUST attend to feedback and accept the work. If possible, the team should look for users and real customers to provide supplementary feedback.

Practices to consider later:

All changes to the codebase are immediately tested and reported on.

Tools should at the least include a version control system and should eventually cover tests suites which will be incorporated into Continuous Integration, and a dashboard that monitors build health prominently.

Within a week of code completion, feedback is obtained about whether a release candidate is deployable.

Normally through user Acceptance Testing be that manual or automated.

4. Self-organisation

Helps: Optimise Flow

Options to start:

The team actively manages the number of requirements being worked on (in at least part of the process).

This can mean managing the work accepted, either through smaller batches, WIP limits or a pull system.

The team has a clear vision and goal which they exist to fulfil.

The team should understand and be invested in the project strategy and business outcome, appreciating how each increment and requirement contributes to this so as to make informed decisions.

The customer representative owns the ROI.

The customer representative has authority to make decisions about prioritisation and direction and is responsible for measuring the outcome.

Practices to consider later:

The team evolves its own set of practices and processes.

This includes training and self-education about Agile practices and the responsibility to introduce, adapt and measure successes. The team define and develop information monitors (whether task boards, CFDs, burn-down charts or an automated set of metrics) and use them to provide greater visibility to themselves as well as management.

The team has all the skills necessary to develop a solution.

This means moving from component or process teams to fully cross-functional feature teams.

A framework exists which allows the team to spend money or change direction.

The 'economic framework' should be created by senior management, following a business plan that covers cost of delay. Decisions are faster and better informed since the team is clear on their boundaries and the economic parameters. An example rule might be: if the cost required to deliver a feature a month early is less than half the monthly cost of delay, then the team can incur the cost.

5. Collaboration

Helps: Optimise Flow and Discover Quality

Options to start:

All requirements are expressed in a way that forms a placeholder for a conversation.

Today, most organisations use user stories for this. They are not obligatory – for some clients use cases and specifications may continue to have a role. The crucial principle is that knowledge is transferred better through conversation than through documentation, and that conversations require less investment. It is easy to invest days in writing a spec for no more value than would be achieved through a 10-minute conversation. Just like this comment is rather longer than the point would take to make in conversation.

The team sits together or has high quality tools for immediate communication.

For distributed teams organise face-to-face kick-off meetings and use video calls for daily stand-ups. Favour open chat rooms for visible team conversations that include everyone rather than 1 to 1 Instant Messaging or email, which are essentially private conversations.

People use creative working sessions and share learning regularly.

Pairing – whether for programming or business tasks – encourages faster learning. Swapping pairs can fast-track learning across the team. Whiteboard sessions, brain-storming and ‘games’ such as Planning Poker also foster collaboration.

Practices to consider later:

Relevant stakeholders participate in regular meetings.	All team members should participate in the daily stand-up, just as all business stakeholders should participate in planning or demonstration meetings. To reduce co-ordination costs, meetings should be regular and time-boxed.
Issues and blocks are captured and then prioritised by the team.	To optimise flow, 'blocked' tasks or team members should be offered help by other team members until the bottleneck or issue is cleared.
Decisions and the thinking behind them are shared and can be challenged using results.	Assumptions should be stated, recorded and then compared to actual results. This transparency encourages the team to hold themselves collectively to account and to learn for future products.
The product, be that hardware, software or documentation is accessible and can be contributed to by all.	In software this might translate to collective code ownership; with hardware, it might include access to prototypes and early designs. In other stages of development it can include all team members adding to requirements. This is not a mandate for 'design by committee'; overall decisions still sit with the customer.

6. Code is treated as an asset

Helps: Discover Quality

Options to start:

Developers have the toolsets they deem necessary to complete the task.

From hardware and software to intangible tools including training and personal skill development. Invest in your development as much, if not more, as you invest in your processes. Few things add value like excellence in engineering, nor is any part so hard to retrofit as quality engineering.

A defined set of metrics on code quality is monitored and reported on.

To begin with, monitoring and reporting can be done simply by peer code review, but eventually an automated process and report generation will offer greater continuity. Common measures include code coverage, code duplication and cyclomatic complexity. Note that any single measure is rarely enough to provide quality assurance.

Developers use code quality reports and measure the impact of technical debt to suggest and make improvements to the codebase.

Maintenance is seen as having recognised value since it improves your ability to make future changes and build on your software. This includes removing unnecessary code, cleaning up poor code and making structural improvements.

Practices to consider later:

Executable tests are written before code to drive simple, more flexible design and support refactoring.

Strong suites of tests make the code more robust and thus make it easier to change and improve. Unit testing, TDD and BDD are common examples.

The deployment pipeline is automated, allowing for rapid or continuous deployment of software.

Integration, system, acceptance and performance tests should be automated so that reports show us why and where any fail has occurred, permitting easier and faster fixes.

Off-shoring is used appropriately.

While the cost of developing code is important, off-shoring comes with the collaboration cost attached to any distributed team. Sometimes, when maintaining or running down code, a slower speed and potential misunderstanding is justified. At other times, perhaps in the launch of a strategically crucial element of software, the potential for miscommunication is too high.

CASE STUDY: Scrum to the nth

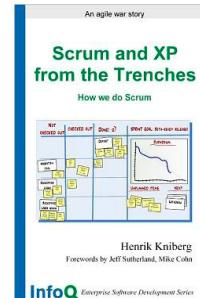
Henrik Kniberg wrote what must be the longest-ever case study of a Scrum implementation in his detailed book *Scrum and XP from the Trenches*. In the book he goes through each practice, artefact and meeting in detail, offering advice and anecdotes. There's no need to summarise each one here, only to point out how every single element adopted from the Scrum framework exemplified the patterns of behaviour we described above.

'Scrum worked!' Kniberg announces with delight at the beginning of his book, 'for us at least'. He is quick to point out that his description is only of his personal and practical experience of Scrum, adapted to his client's explicit circumstances. The adaptations, however, are cosmetic, not fundamental. Kniberg's Scrum teams tried several different experiments – in team size, sprint length, definition of 'done', ways of using the practices (including ways of presenting the backlog or demonstrations) and adding in practices from XP, such as pair programming and continuous build. Such experiment is a natural result of using feedback from the team's retrospectives and following Scrum's essential pillars of transparency, inspect and adapt. It's also an essential proof of commitment to improvement.

Ensuring backlog items were always presented as business goals, not technical instructions, for example, was intended to offer the team the autonomy to develop the best solution. The project backlog was kept in an Excel spreadsheet that anyone could access to ensure that developers could enter notes or change estimates (although during planning they increased visibility by using index cards on the wall). We would call both excellent examples of the commitment to self-organising.

Prioritisation was taken seriously by the team. They decided on a form of project backlog with 6 fields: ID, Name, Importance (measured in relative points), Estimate (measured in 'ideal man days'), How to demonstrate (some would call this an acceptance test) and Notes. They ensured they didn't end up with lots of high priority items at story level 100, however, by insisting that any story to be considered for the backlog should have a unique importance level score so that the team would never be in doubt about which order to work on items.

Scrum naturally delivers in increments, but the team put a great deal of work into their sprint planning, insisting the Product Owner attend each meeting and always formulating a Sprint goal that described the point of the increment. Both incremental delivery and transparency were crucial in this success, because it involved making difficult trade-offs, reducing scope on a crucial feature to deliver it early, or reducing the importance of other items to allow the team to focus more effort on the crucial feature.



The team recognised the importance of small increments – in Scrum these would be defined as short sprints. Kniberg's reasoning is impeccable: 'short sprints are good. They allow the company to be "agile", i.e. change direction often. Short sprints = short feedback cycle = more frequent deliveries = more frequent customer feedback = less time spent running in the wrong direction = learn and improve faster.' He also goes on to explain the team's preference for slightly longer sprints to build up flow, in order to come up with the duration that worked best for them – 3 weeks. Once there, the team kept to this length, allowing a rhythm to develop – everyone knew there would be a release in 3 weeks.

The feedback was reinforced by Kniberg's insistence that teams always did a product demo at the end of every sprint – however little was really 'done'. He claimed that apart from the obvious utility of gaining feedback from stakeholders, it ensured that teams were fully motivated to finish off more items in the next sprint. Indeed, Kniberg valued feedback so highly that he normally introduced 'slack' between each sprint to allow the team time to process all the information and feedback they received.



Figure 25. The team following the Scrum process
(from Scrum and XP from the Trenches)

Although Kniberg and his teams were fairly disciplined in following Scrum, they did choose one major adaptation, they also introduced several practices from XP. They selectively used pair programming as an experiment and felt that it improved code quality and focus, as well as helping to spread knowledge through the team. The team also uses TDD, incremental design, coding design and various other XP practices layered onto Scrum as the engineering tools to encourage good development.

A final adaptation to the Scrum ideal was that although the team aimed to produce a shippable increment at the end of each sprint, the work still needed to go through acceptance phase testing where testers access the system manually in an attempt to replicate real users.

5 CONCLUSION

If you want to explain Agile to your colleagues in a single sentence, then here it is:

Pick the idea you think is best, test it quickly, learn and keep going
– and talk to each other while you do it.

Don't blame us if they are unimpressed. 'That's it? You read 52 pages to learn that?'

And there's the problem. The idea is simple. It's the doing it that's hard. So make sure they've tried out the activities in the related sessions, and start by introducing a practice you think might help today.

Learning outcomes

By the end of this session, you will understand:

The importance of understanding the reason for change, rather than blindly embracing process

Common compromises made in an Agile implementation

- Agile practices in development are sandwiched between traditional processes in planning, budgeting and approvals
- Teams are often assigned across several projects
- No individual assumes the customer role
- Teams are not truly cross-functional
- Alternative governance and control processes run in parallel
- Software is not released frequently

The likely impact on expected benefits

- Frustration and duplication of effort
- Disappointing results, leading organisations to see Agile as 'failed'

The potential to adapt elements of Agile methods

- Unique circumstances mean individuals and teams can tailor practices to their needs

Whether adaptation is truly necessary

- Much 'adaptation' is really just a reluctance to change
- Most Agile practices can be implemented

Patterns commonly found in organisations successfully using Agile

Risks of ignoring the pattern

Ways to check you are following the principles

- A commitment to improve the outcomes
- Delivering using increments
- Feedback
- Self-organisation
- Collaboration
- Code is treated as an asset

Challenges and obstacles likely to arise when implementing the principles

- Knowing your business allows you to identify the most likely issues in process or culture
- Plan for these upfront

How to take small steps towards change, selecting a single practice designed to improve an existing problem

Inspect and adapt using empirical evidence

Appropriate practices to begin with

BIBLIOGRAPHY

- Ambler, S.**, The Agile System Development Life Cycle (SDLC) [online] Available at <<http://www.ambysoft.com/essays/agileLifecycle.html>>. [Accessed 15 March 2012].
- Ambler, S.**, 2012. Disciplined Agile Delivery - A Practitioners Guide to Agile. Addison Wesley.
- Anderson, D.**, 2010. Kanban: Successful Evolutionary Change For Your Technology Business. Blue Hole Press.
- Appelo, J.**, 2010. Management 3.0: Leading Agile Developers, Developing Agile Leaders. Addison Wesley.
- Beck, K., Andres, C.**, 2004. Extreme Programming Explained: Embrace Change. 2nd Edition. Addison Wesley.
- Cohn, M.**, 2009. Succeeding with Agile: Software Development Using Scrum. Addison Wesley.
- Davies, R., Sedley, L.**, 2009. Agile Coaching. The Pragmatic Bookshelf.
- Elssamadisy, A.**, 2008. Agile Adoption Patterns: A Roadmap to Organizational Success. Addison Wesley Professional.
- Highsmith, J.**, 2002. Agile Software Development Ecosystems. Addison Wesley.
- Highsmith, J.**, 2009. Agile Project Management: Creative Innovative Products. 2nd Edition. Addison Wesley.
- Jeffries R.**, 2009. Beyond Agile – Inspect and Adapt? How? [online] Available at <<http://xprogramming.com/xpmag/beyond-agile-inspect-and-adapt-how/>>. [Accessed 15 March 2012].
- Kennaley, M.**, 2010. SDLC 3.0: Beyond A Tacit Understanding Of Agile. Fourth Medium Press.
- Kniberg, H.**, 2011. Lean from the Trenches: Managing Large-Scale Projects with Kanban. The Pragmatic Bookshelf.
- Leffingwell, D.**, 2007. Scaling Software Agility: Best Practices for Large Enterprises (Agile Software Development). Addison Wesley.
- Marcus, G.**, 2008. Kluge. Houghton Mifflin Hardcourt.
- Schwaber, K., Sutherland, J.**, 2011. The Scrum Guide - The Definitive Guide to Scrum: The Rules of the Game.
- Shalloway, A., Beaver, G., Trott, J.**, 2009. Lean-Agile Software Development: Achieving Enterprise Agility. Addison Wesley.
- Shore, J., Warden, S.**, 2007. The Art of Agile Development. O'Reilly Media.
- West, D.**, 2011. Water-Scrum-Fall Is The Reality of Agile for Most Organizations Today. [online] Available at: <http://www.cohaa.org/content/sites/default/files/water-scrum-fall_0.pdf>. [Accessed 4 September 2012].
- Wikipedia**, 2011. Declaration Of Interdependence. [online] Available at: <http://en.wikipedia.org/wiki/PM_Declaration_of_Interdependence>. [Accessed 15 March 2012].



valueflowquality.com

emergn.com