The Agile Spectrum

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

The Agile is a broad church. It includes a lot of tools and techniques, some applicable to some teams and some environments and some applicable elsewhere. Anyone who thinks hard about how to measure Agility quickly realises it cannot be measured by adoption of practices, it needs to be considered on outputs and abilities.

As already said, Agile is sometimes simply defined as "not the waterfall."

In truth there is a spectrum with strict-waterfall at one end and "pure Agile" at the other - shown below. Since waterfall never really worked that well very few teams are at the strict waterfall extreme.

In his analysts of software development projects over 20 years Capers Jones suggests that in general requirements are only 75% complete when design starts and design is a little over 50% complete when coding starts (Jones 2008). He goes on to say that as a rule of thumb each stage overlaps by 25% with the next one. In other words, the strict-waterfall isn't strictly waterfall.

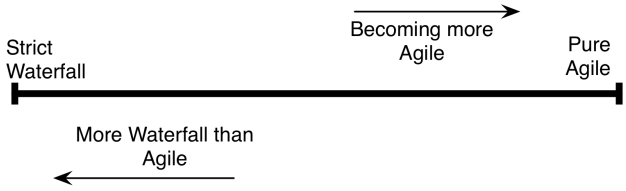

Figure 1: The spectrum from Strict Waterfall to Pure Agile almost everyone is somewhere in- between

It would seem reasonable that the pure Agile end of the spectrum is equally sparsely populated. Whether because few teams need to be so extremely Agile, or whether because experience and tools have yet to allow such a degree of Agility, some staged elements exist in many projects.

More than one software development team has encountered the situation when the team want to be more "Agile", the organization and management might even be asking them to be more "Agile" but, there are still many "requirements" in a big document and the expectation is that all these will be "delivered." Experience and anecdotal evidence suggest this scenario is faced by many teams.

This mismatch arises when the organization is largely waterfall but the development team are trying to work Agile. I have consulted with companies where senior managers believe Agile is only a delivery process for developers, specifically coders. The rest of the development process (Business case, requirements, design,

deployment and even testing) is waterfall. In their view it is just the bit in the middle which is "Agile".

I have taken to labelling three points on the Agile spectrum in order to clarify thinking around what teams are actually doing. And which tools from the Agile toolbox they are actually interested in, the different degrees of Agility and provide teams with a way of resolving the requirements-delivery mismatch.

Three Agile Styles: Itertive, Incremental, Evolutionary

On close inspection Agile has, at least, three styles. I call these styles: Iterative, Incremental and Evolutionary - shown in the diagram below. These classifications are largely governed by the development teams relationship with the requirements and whether the organization wants work defined in advance or prefers goal directed working.

It is important to understand that while one may personally consider these three styles to form a hierarchy with Evolutionary in some way superior to Incremental, which in term is better than Iterative this is not the intention. Organizations adopt these different styles for reasons. There are other forces at work which mean that, say, Iterative is the best approach in a particular environment.

These three styles occupy different places on the spectrum. But, in truth, there is no clear cut divide between iterative and incremental, incremental and evolutionary or even iterative and evolutionary. The three styles all overlap and fade into one another.

In my experience teams can use just about any Agile method (e.g. Scrum, XP, Kanban, etc.) with in any of these three styles. Logically Scrum gravitates towards Iterative, XP to Incremental and Kanban to Evolutionary but since these methods have little to say about the requirements process all three methods can be used in any of the three styles.

Iterative Development - Salami Agile

Working in bite-sized chunks from predetermined requirements with one big delivery at the end.

Iterative Agile refers to the practice of undertaking projects in small, bite-sized chunks. Every two-weeks (or so) an iteration completes and the total amount of work is burnt down on a chart. Customers will probably be shown the latest version of the software at the end of the iteration although this is little more than a demo. Most likely there will be a single software release at the end of the work - followed by several "maintenance" releases.

At the start of work there is a big requirements document. The work to be done is, at least in theory, defined in advance. Someone, perhaps a previous project,

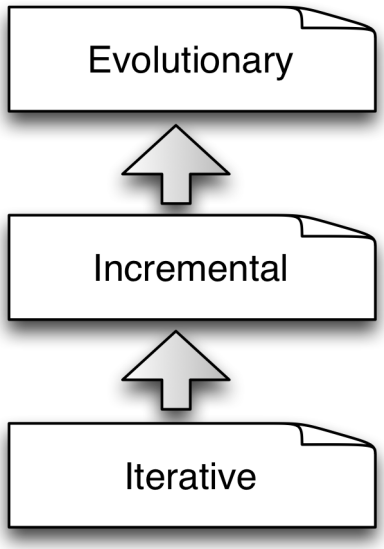

Figure 2: Three styles of Agile

perhaps external consultants, has created a list of the features and functionality the new system must, or should, have. The development team are expected to delivery, all of it, or nothing. The approach here is to see the big requirements document as an uncut sausage of salami (long and dense). Someone on the team - preferably someone with Business Analysis skills but it could be a developer, project manager, or someone else - needs to slice the requirements into thin pieces of salami (story) for development. There is no point in slicing the whole salami in one go. That would just turn a big requirements document into a big stack of development stories. The skill lies in determining which bits of the document are ready (ripe) for development, which bits are valuable, and which bits can be delivered independently.

Some slices of salami will be thicker than others but that's just the nature of the world. Over time, with more skill at slicing salami it will improve and slices will be thinner.

Working in this fashion opens up the ability to accept change requests relatively easily. But because the work has been set up as a defined project with "known" requirements these opportunities probably aren't exploited to the full. Similarly, opportunities to remove work will also appear - some slices of salami may be thrown away - but again this will depend on how rigidly the project seeks to stick to the defined work.

As well as the requirements document there are probably some estimates somewhere - maybe even a Gantt chart, which has to be updated to maintain the

illusion that it is useful. These plans assume that all requirements given in the original document will be implemented and delivered. However even in this style this assumption may be relaxed over time.

This is the land where the burn-down chart reigns supreme. There is a nominal amount of work to be done and with each iteration there is a little less. Such empirical measurement is likely to provide a good end-date forecast once some actual historic data has been gathered. Until the team have completed one sprint all estimates are guesswork.

Iterative Agile is the basis for incremental development and occurs somewhere about the middle of the spectrum. To go further towards pure Agile work has to be based less on a shopping list of features and more on overarching overall objective for the work.

Incremental development

Working in bite-sized chunks from predetermined requirements with regular deliveries and accepting changes.

The big requirements document still exists and salami slicing is still prevalent in incremental, at least during the early stages. Work is completed in bite-sized chunks and periodically delivered to customers to use. These events might, or might not, occur in tandem. While a team might work in two-week iterations deliveries might only occur every two months.

The pieces of salami are delivered to the customer early, and over time customers start to realize they don't need some things in the original requirements document so some slices can be thrown away some and some salami left unsliced and unused.

This model capitalizes on the flexibility provided by eating salami rather than steak. Requirements which were not thought of can be easily incorporated, others can be changed, enlarged or shrunk.

The iterative style still assumes the original requirements are correct so not implementing them all, or changing what is done is a sign of earlier failure. In incremental development changes are seen positively and reductions in scope are seen as savings - a sign the model is working.

That real live users are getting access to the software early is valuable to the business. It also means user insights and requests are inevitable. Still there is a major requirements definition somewhere and while the team can accept change requests easily it is still expected that one day the team will be done.

Burn-down charts might still be used to track progress but at times they may appear as burn-up charts as work is discovered.

Tensions arise when the team are instructed to refuse changes, or themselves insist on continuing to salami slicing the original requirements document but users and customer are asking for changes based on their experience. In other

words, the users and business have changed their understanding but the team do not, or are not allowed to, change theirs.

There is no hard and fast line between iterative and incremental, they are just points on the spectrum - with incremental to the right of iterative by virtue of delivering more often. Perhaps the hallmark of incremental is that the team delivers on a regular schedule. When delivery is a big deal, irregular, a special occasion, then things are really just iterative with occasional drops.

Evolutionary Agile - Goal Directed Projects

Working in bite-sized chunks from emerging requirements with regular deliveries Evolutionary Agile takes this to the next level and is the natural home of goal directed projects.

Teams start work with only a vague notion of the requirements. Over time the needs, practices and software all evolve. As the software is released to customers the needs are reassessed, new requirements discovered, existing ones removed and new opportunities identified.

The team has a goal, the team will determine what needs doing (requirements) and do it (implementation) as part of the same project. The team is staffed with a full skill set to do the complete work - analysts, developers, testers and more. The team is judged and measured by progress towards the goal and value delivered rather than some percentage of originally specified features completed.

Even goal directed Agile needs to start by establishing a few initial requirements. Some teams call this period "sprint zero" in which a few seed stories are captured from which product development (coding) can start as soon as possible. Other teams hold a "Story writing workshop" to brainstorm stories to seed the work.

From there on requirements analysis and discovery proceed in parallel with creation. Those charged within finding the requirements (Product Owners, Product Manager, Business Analysts or who-ever) work just a little ahead of the developers.

Burn-down, even burn-up, charts have little meaning for goal directed work because the amount of work to be done isn't know in advance. Work to-do and work done are better tracked with a cumulative flow diagram showing the progress in both discovering needs and meeting needs.

Governing goal directed work is superficially more difficult because it is not measured against some nominal total. Instead work needs to be measured against progress towards the goal.

These projects should be placed under a portfolio management regime that regularly - at least quarterly - reviews the progress and value delivered so far against the goal and the costs incurred. These figures should be produced within the team itself, and the team should feel confident enough to suggest its own end.

Partitioned spectrum

Adding these points to the spectrum gives Figure 3. For a team migrating to Agile the objective is to move from left to right. These three approaches might reflect three level of capability but they may also reflect the nature of Agile in a particular organization. One size does not fit all some teams are better off with one style of Agile and some with another.

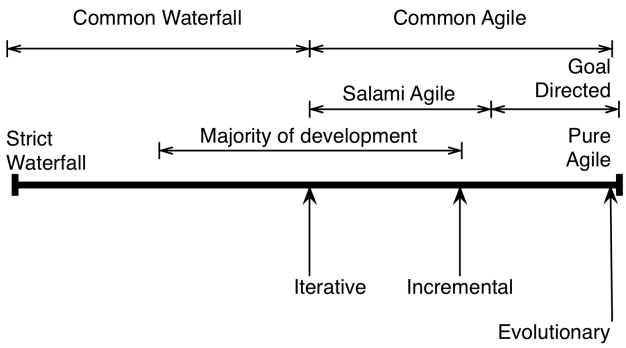

Figure 3: Different regions of the Agile Spectrum

Many organizations, rightly or wrongly, considered any development process that is iterative in nature to be "Agile". Therefore, in common parlance any method on the right of this spectrum is called Agile, while anything on the left is called Waterfall.

Waterfall approaches might split work into stages, work packages, or sub-projects which can make work look a little like iterative development. Although Waterfall development is associated with Big Bang releases many such projects released several small-bangs. And after release "maintenance" teams would continue to release updates.

Just as few teams actually embrace 100% evolutionary development, few teams ever followed a pure Waterfall approach. Indeed, I would argue that the Waterfall is so fundamentally flawed a pure Waterfall was always impossible.

In my experience most development projects lie somewhere between these two extremes, mostly clustered around the centre. Although I don't have any data to support my argument I suspect that a standard-distribution bell-curve could be laid over this diagram would show most teams following a interactive process, with a few teams more incremental and a similar number doing periodic releases on a Waterfall basis.

Style Characteristics

While there are no hard and fast rules about when a team is doing one style of development and another there are some common traits visible by looking at the practices the teams adopt. These are summarised in the tables below. While these charcateristics are a useful way of describing and comparing different styles and different teams they are not prescriptive.

Routines

Practices	Waterfall	Iterative	Incremental	Evolutionary
Stand-ups[1]	No	Yes	Yes	Yes
Status reporting	Against plan	Regular	Regular	Against goal
Retrospectives	At end[3]	Occasional[4]	Regular	Regular
Demo[2]	Occasional	Occasional	Regular	Information only[5]

[1] Stand-up meetings are sometimes called Scrum meetings although this term is also, sometimes, applied to Planning meetings

[2] Demonstration also known as "Show and tell"

[3] Lessons learned reports issued at the end of development

[4] More talked about than done

[5] Only as information prior to release

Planning

Practices	Waterfall	Iterative
Schedule planning	At start & revisions as needed	Regular with iterations[1]
Budget	Allocated at start	Allocated at start
Budget control	Monitored against plan	Monitored against plan
Design	Big up front activity	Mostly upfront

Practices	Incremental	Evolutionary

Schedule planning	Regular iterations[1]	Iterations or on demand
Budget	Mostly upfront	Arrives in increments[2]
Budget control	-	Value delivered v. cost incurred[3]
Design	Some up front + refactoring	Emergent with refactoring

[1] Iterations are assumed to be 2 to 4 weeks long

[2] Monies may be allocated in tranches, venture capital style; or rolling *beyond budgetting* style allocations made

[3] In keeping with rolling budgets costs should be controlled by regular assessment of costs incurred and value delivered

Technical practices

Practices	Waterfall	Iterative
Releases	Once at end	Irregular or once at end
Automated Unit Testing[1]	No[2]	Maybe
Automated Acceptance tests	No	No
System Integration Tests	At end of work	During work
User Acceptance Testing	Only end of work	Maybe during work
Integration	Occasional, often delayed	Continuous

Practices	Incremental	Evolutionary
Releases	Regular during project	Min once per iteration
Automated Unit Testing	Yes	Yes
Automated Acceptance tests	Yes	Yes
System Integration Tests	Ongoing during work	Ongoing during work
User Acceptance Testing	Ongoing during work	Ongoing during work
Integration	Continuous	Continuous

[1] Test Driven Development, TDD, specifically means: Automated Unit Tests written by developers written for their own code "Test First" i.e. before the production code is written

[2] Manual unit testing may be used sporadically

Management

Practices	Waterfall	Iterative
Tracking charts	Gantt	Burn-down
Goal	Requirements are goal	Requirements are goal
Requirements	Specified in advance[1]	Specified in advance
User feedback	Minimal, even resisted	Little
Change control	Changes resisted[2]	Changes seen as problem

Practices	Incremental	Evolutionary
Tracking charts	Burn-up	Cumulative flow
Goal	Initial reqs + goal directed	Goal governs & directs
Requirements	Specified in advance & salami sliced	Emergent
User feedback	Plenty but little incorporate[3]	Fundamental to success
Change control	Relaxed traditional	Changes are first class requests[4]

[1] Officially, on paper, requirements are specified in advance, in practice work may begin with vague requirements which are still being debated. This is typically seen as a problem in traditional working.

[2] Changes requests are frequently seen as problem and resisted

[3] While there may be plenty of user feedback in incremental working there may be limited ability to incorporate it.

[4] No formal change control because changes and emerging needs are the lifeblood of the work

Some Examples

Interestingly, there is one area of software development were the goal-directed evolutionary approach has long been the norm: maintenance. Maintenance teams have the goal of keeping systems working, fixing bugs and, often, small enhancements. Work emerges over time and the highest priority work gets done and other work is left undone.

I remember working on a financial reporting tool called FIRE in 1997. There was no roadmap or even plan for the product. The company had three, four, then five and even six customers. As each sale was made new requirements emerged: port from Solaris to Windows, from Sybase to SQL Server, to Oracle, to AIX. And of course bugs.

These requests arrived with more or less noise and urgency. I introduced time-boxed iterations to the team: we released each month, and put a white board on the wall to show what we were doing. Each iteration had a collection of work, we delivered and then reviewed what had arrived in the last month.

Evolutionary would be the best characterisation of FIRE. Requirements and processes emerged as the work progressed. The overall goal was never clearly stated and we only had elementary unit testing - but we had some!

Conversely, one of my clients in Cornwall undertook a complete rewrite of their flagship product in an iterative way. The feature list was almost entirely taken from the existing product. The team worked in one-week iterations, at the end of each iteration their proxy-customer reviewed the work and approved that which was done. The work was grouped - physically - into monthly bundles - November, December, January, February. The original aim was to release in March but this slipped for various reasons. The company did not wish to release anything until everything was ready and could be released.

Of course once the first release was done the working style changed. The team adopted a more incremental approach with monthly updates. They still had plenty of features - new or held over - to continue implementing. I believe that over time they will, or even have already, move to a more evolutionary approach accepting new work on demand.

With the first, big, release done, the team's overarching goal changed, from "Get a version released with a subset of the current features" to something closer to "find work, do work".

A change model

It is useful to consider this spectrum as a change model. Assume a starting point somewhere on the left of the spectrum, a team doing some form of common waterfall with all the imperfections that suggests. Becoming Agile, by any definition, means moving to the right.

As a first step the team can adopt a interactive approach and use Salami Agile to manage requirements. In time, as they improve they advance to an incremental approach. To go further the team need to move away from salami and become goal directed. This requires more of the organization to embrace the Agile ways of the team. Some teams may stall here for this reason.

When a team has a proven track record at incremental delivery the organization will come to trust the team. This puts them in a good position to find opportunities to work in a for evolutionary, goal directed, way.

Tools and Mental Models

Another insight offered by the spectrum concerns the tools we use and mental models we hold. Historically the tools, techniques and, most importantly, mental model we use to develop software originate in, and largely, assume a Waterfall model, e.g.

- Requirements once written will suffice for the whole project. Changes are anomalies that should be resisted, hence *change request procedures*.
- Requirements are fully known therefore software designers can produce an optimal design which then simply needs coding (sometimes called "SMOP" - Simple Matter of Programming).
- Plans can be made based on time and scheduled using Gantt chart type techniques. Slippage's are again an anomaly that need to be corrected.
- Testing can be left until the end and adequate time can be allowed for resulting fixes.

The list could go on. The problem is that once work starts to deviate from this model - whether intentionally or not - these tools become difficult to work with. Gantt charts need updating to reflect reach and every slippage, dependencies need reworking, requirements documents need updating, software designs are need changing and testing inevitably throws up more issues.

To paraphrase Helmuth von Moltke "No plan survives contact with the coding process."

With this toolset and mental model the solution is to add time: more time for requirements, more time for design, more time for testing and so on. But, more often than not adding more time complicates problems: costs increase, requirements change in the interim, customers get unhappy and apply pressure, designs become dated as new technologies emerge, unused resources are released and so on.

In other words: the tools and mental model associated with the Waterfall end of the spectrum do not work particularly well once we move to another point on the spectrum.

However, the tools and mental model inherent in the Pure Agile end of the spectrum work pretty well cross the entire spectrum. Suppose for a moment one was working on a project where requirements could be known in advance and would not change, suppose technology was stable and so on, i.e. the perfect environment for a Waterfall project.

Now apply the Agile toolset: User Stories could still be used for capturing work to be done, full requirements elicitation and specification need not be done (even though it could given our assumption of stability) until the work was about to be done, automated testing (at all levels) would still work, burn-down charts and velocity would still be a good way of planning.

If nothing else Agile provides a toolset and mental model of how to develop software which is more generic and applicable than the Waterfall model and associated tools.

Summary

Although Waterfall and Agile are often characterised as straight alternatives neither is particularly well defined. It is better to view them as representing different areas on a continual spectrum from a strict phased approached to no-phased approach.

On the Agile side of the spectrum there are different ways of approaching work. Many teams work with pre-determined requirements in a salami fashion. They deliver software in iteratively or incrementally. A few teams work in a more goal-directed fashion were need, solution and process are evolving. Different techniques, tools, practices and processes are used at different parts of the spectrum but there are no hard and fast rules as to what is used when.

References

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

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

About the author

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

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

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