The Agile Spectrum

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.

Agile is sometimes simply defined as "not the waterfall." This is a poor, if understandable, definition. Unfortunately, this means that any process that doesn't follow the classic waterfall strictly can be considered Agile. Adding to the confusion "Waterfall" can cover a number of different approaches, stage gate models like DoD 2167 and 2168 and all encompassing methods like SSADM.

In companies where strong, documentation centric, procedures have been hoisted on development teams Agile is sometimes seen as a "get out of jail free" card. Simply saying "this project is Agile" is seen to exempt work from company procedures. Unfortunately, this card is also used as a cover for cowboy development.

In truth there is a spectrum with strict-waterfall at one end and "pure Agile" at the other - Figure 1. 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.

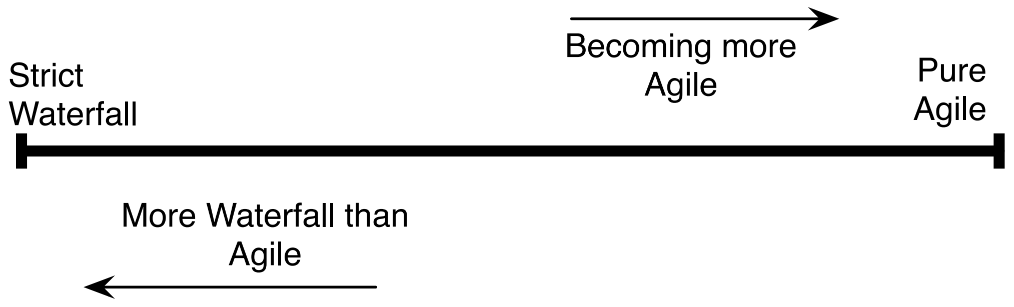

Figure 1 The spectrum from Strict Waterfall to Pure Agile almost everyone is somewhere inbetween

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. Business case, requirements, design and even testing is waterfall, just the bit in the middle is Agile.

This document attempts to both understand the different degrees of Agility and provide teams with a way of resolving the requirements-delivery mismatch.

Three Agiles

On close inspection Agile has, at least, three styles: iterative, incremental and evolutionary, shown in Figure 2. These 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.

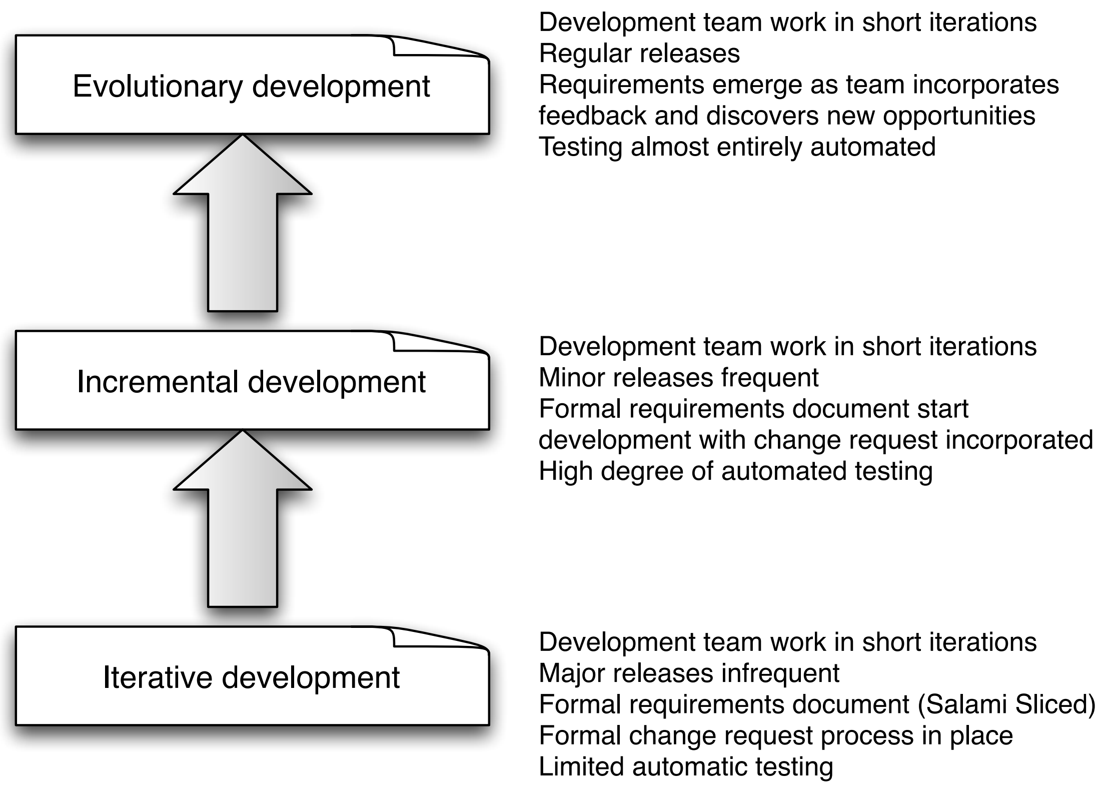
Evolutionary development
Development team work in short iterations
Regular releases
Requirements emerge as team incorporates feedback and discovers new opportunities
Testing almost entirely automated
Incremental development
Development team work in short iterations
Minor releases frequent
Formal requirements document start development with change request incorporated
High degree of automated testing
Iterative development
Development team work in short iterations
Major releases infrequent
Formal requirements document (Salami Sliced)
Formal change request process in place
Limited automatic testing

Figure 2 - Three levels of Agile

As we shall see in a moment, 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.

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, 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.

However, 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.

Salami 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

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 though of can be easily incorporated, others can be changes, enlarged or shrunk.

The iterative approach 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 each delivery is a big deal, 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 evolve. As the software is released to customers the needs are reassessed, new requirements discovered, existing ones removed and new opportunities identified.

The teams 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. 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.

Taken together

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.

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.

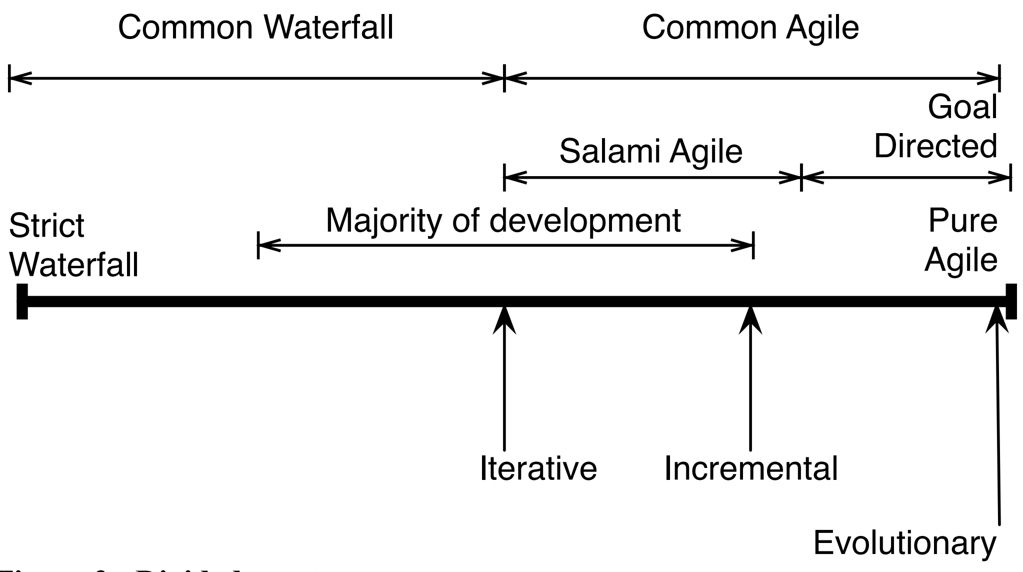

Figure 3 - Divided spectrum

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. (Before writing to take me to task please read the original Waterfall paper (Royce, 1970).)

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.

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 Table 1. While these attributes are a useful way of describing and comparing different styles and different teams they are not prescriptive.

Practices	**Waterfall**	**Iterative**	**Incremental**	**Evolutionary**
Stand-up meetings	No	Yes	Yes	Yes
Planning	Start of project; revisions as needed	Regular 2-4 week iterations	Regular 2-4 week iterations	Regular 2-4 week iterations

Status reporting	Regular, against plan	Regular	Regular	Regular against goal
Retrospectives	Sometimes at end of work	Occasional - more talked about than done	Regular	Integral
Demo "Show and tell"	Occasional snapshot	Occasional	Regular	Only as information prior to release
Planning				
Budget	Allocated at start	Allocated at start	Mostly upfront	Arrives in increments
Budget control	Monitored against plan	Monitored against plan		Value delivered v. cost incurred monitored
Technical practices				
Releases	Once: at end	Once at the end, or at irregular intervals	Regular during project	Regular like clockwork
Automated Unit Testing	No	Maybe	Yes	Yes
Automated Acceptance tests	No	No	Yes	Yes
Test First Development (TDD)	No	Some	Lots	Everywhere
System Integration Tests	At end of project	During project	During project	Ongoing during project
User Acceptance Testing	Only end of project	At end of project	During project	Ongoing during project
Continuous integration	No	Yes	Yes	Yes
Tracking charts	Gantt	Burn-down	Burn-up	Cumulative flow
Design	Big up front	Mostly upfront	Some up front	Little upfront;

	activity		plus refactoring	mostly emergent with refactoring
Goal	Requirements are goal	Requirements are goal	Mix of upfront requirements & goal directed	Governs project & directs progress
Requirements	Officially specified in advance	Specified in advance; salami sliced to developers	Specified in advance; salami sliced to developers	Emerge during project
User feedback	Minimum	Little	Plenty but little scope to change incorporate	Fundamental to project success
Change control	Traditional - changes seen as problems	Traditional	Relaxed traditional	None - changes are requests

Table 1 - Comparison of characteristics

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 greater 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 is currently writing a completely new version of their flagship product in an iterative way. The feature list is almost entirely taken from the existing product. The team work in one-week iterations, at the end of each iteration their proxy-customer reviews the work and ticks it as done.

The work to do is grouped - physically - into monthly bundles - November, December, January, February. The original aim of releasing in March but it now looks like it will be April. Nothing will be released until it is all released.

Of course once the first release is done working will change. Probably the team will take more of an incremental approach with monthly updates. They still have plenty of features - new or held over - to continue implementing for a few months. I expect that at some stage new requests and ideas will bring a more evolutionary nature to the work.

This team will to revisit their overarching goal. As I write the goal is "Get a version released with a subset of the current features." At some time in the near future they will need to question the goal lest they drift into a "find work, do work" mentality.

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. Being 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 they are opportunities arise for goal directed, evolutionary work.

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.

About the author

Allan Kelly has held just about every job in the software world: system admin, tester, developer, architect, product manager and development manager.

Based in London and he works for Software Strategy Ltd. He specialises in helping software companies adopt and deepen Agile and Lean practices through training, consulting and coaching. In addition to numerous journal articles and conference presentation he is the author of "Changing Software Development: Learning to become Agile". Allan holds BSc and MBA degrees and is PRINCE2 certified.

For more about Allan see **http://www.allankelly.net** or e-mail him at **allan@allankelly.net**.

Details of consulting and training services available from Software Strategy Ltd. can be found at **http://www.softwarestrategy.co.uk**.

Acknowledgements

Thanks to Paul Grenyer and Ed Sykes for reviewing an early draft of this article; and the Overload editorial team for their usual attention to duty.

References

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

ROYCE, W. W. 1970. *Managing the development of large software systems: concepts and techniques*.