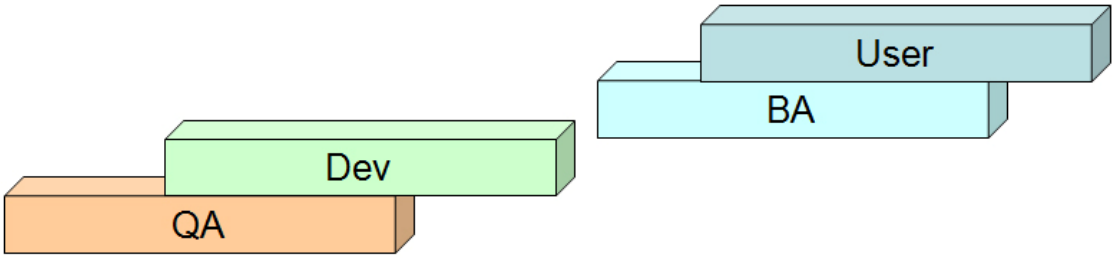
The Pipelining Anti-Pattern

If you have analysts working ahead of development, or have testers working significantly behind development, then you may have "Pipelining" problems.

Pipelining is the term used to describe the situation when business analysts are working ahead on the requirements for a future iteration; the development team is working on the current iteration, and the test team is engaged on a previous iteration.



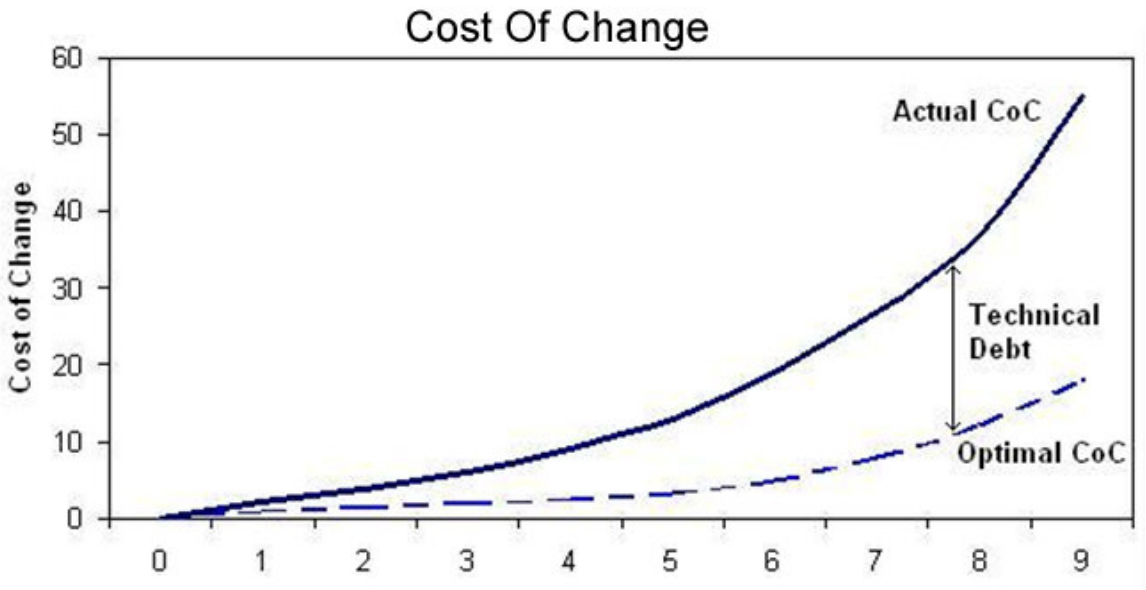
In some circumstances analysis may be several iterations ahead and testing several iterations behind. To some people this may seem an efficient use of resources with each group running at their optimal speed, unfettered by the co-ordination constraints of different groups. However from an agile and lean perspective this is problem, a bad-smell that needs fixing.

Here are the problems with pipelining:

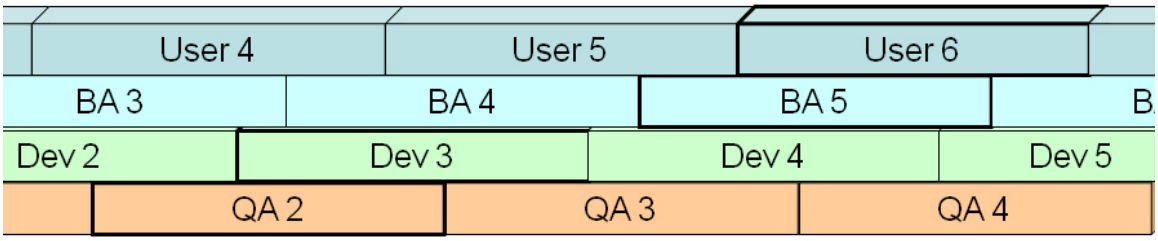
Three teams not one – in a project where pipelining is occurring we do not have one cohesive team we have three teams (or more). It is hard enough co-ordinating the members of one team towards a common goal aligned to business benefits. When there are three teams it is just too easy for people to claim that they did their bit and problems lie with other groups. Yet, the fact remains that if the software does not meet business satisfaction then it is everyone's problem.

Increased Work In Progress (WIP) – Requirements whether they are in the form of user stories, use cases, or formal specifications all represent work invested that has not delivered value to the business. The same goes for code, until this functionality has been tested to the satisfaction of the business it is not valuable. As the time increases between capturing the requirement and finishing the last test two problems occur. The first is classic accounting, money have been invested for no return yet and there is a risk associated with future returns. The second is that requirements decay; the longer we keep requirements around for, the higher the likelihood that they will no longer be required or will have to change.

Increased time from defect injection to defect remediation – the cost of change increases the longer a defect goes undetected. In a pipelining project, defects introduced by faulty analysis could take months to be detected in testing or user review. Fixing the problem after this period of time will entail refactoring far more code (for the work happening in the interim) than if it was detected earlier, and will increase technical debt.



Time fragmentation costs when groups collaborate – on pipelining projects the test group could be working on iteration 2, the developers on iteration 4, and the analysts on iteration 5 or 6.

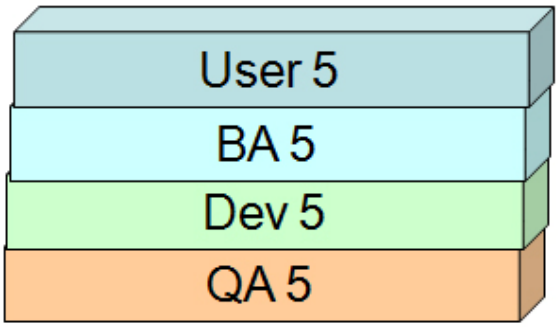


If a developer has a question for an analyst or a tester for a developer then task switching costs will be incurred. Task switching is the time required to respond to an off-topic interruption.

We must mentally "park" our current work, try to think back to whatever it is this person who has interrupted us is asking about, recall the details, answer their questions and then resume where we left off. The problem with this is that studies have shown people, while great at many things, are generally very poor at task switching. Our recall of past details is poor, defence mechanisms in our brain designed to dull unpleasant experiences erode memories of nasty problems, and we are terrible at remembering where we parked our current thoughts. The net result is that just a few requests to recall past events really mess up our performance and yield low quality answers to the enquirer.

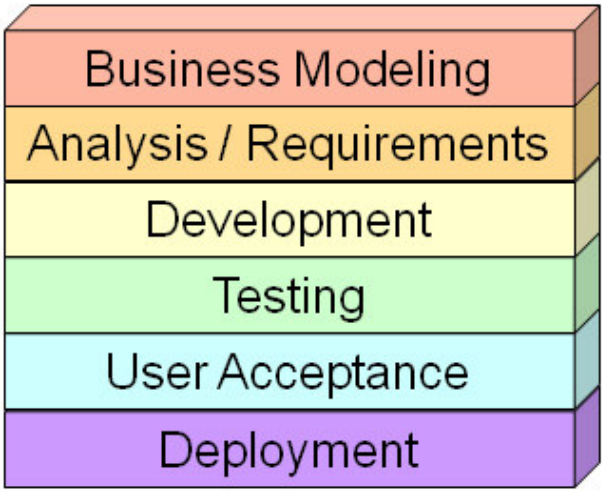
Throughput optimization – three or more teams running through work at their own speeds with fewer dependencies on others may seem like it is working, but from a systems perspective it is a problem. Processes running at different speeds require work buffers to avoid running out of work and they build piles of unfinished work. This inventory is waste in the system and leads to scrap and more rework as processes change and partially completed work needs to be updated.

What we really want is concurrent development by a multi-disciplined team working on the same iteration.



Now, obviously there is a little staggering of activities; analysts, developers and coders are unlikely to start on stories simultaneously. However, the goal is much smaller batch sizes, less role specialization, more team cohesion on common goals for the iteration. Less work gets started, but more work gets finished.

What we are aiming for is a vertical cross section of tasks on the same user stories each iteration



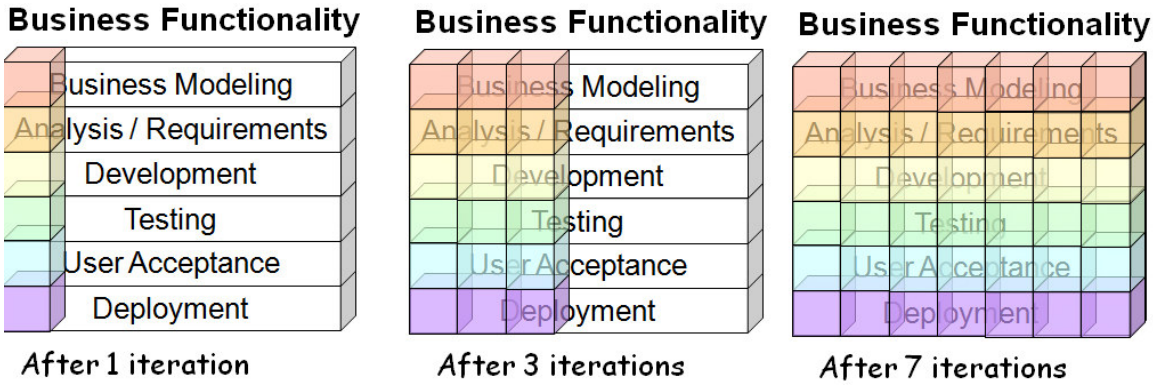
Several of the tasks may be performed by the same people (tasks not roles). However we are looking for complete development of completion in the same iteration. The benefits include:

Better team cohesion – We create one team with a shared responsibility for the same short-term objectives. There is less “us and them” mentality and ownership dismissal concepts such as “throwing things over the fence” for development or testing are reduced.

Better load levelling – With less role specialization, developers do more testing or requirements analysis as the workload demands. We focus more on story completion and people undertake more diverse roles to get the job done. A team of generalizing specialists who know to pick up tasks to speed story completion is the best load balancing asset available.

Less WIP – Less Work In Progress means less scrap when things change and less invested effort without a return. Deliberately light requirements documents that are essentially “promises for a discussion on a topic” also help minimize WIP. If a requirement goes away or is replaced by a higher priority one then it is no great loss since little effort was invested in its creation.

So for a release of software we are looking for the incremental delivery of features, iteration by iteration.



Fixing the Pipelining Anti-Pattern
Recognizing the pipelining anti-pattern is the first step, determining how to fix it is the next. A number of alternatives exist; I have listed three below in order of preference

1) **Stop starting new work and finish the current work in progress before tackling new items** – Take the business analysts off of driving ahead with new requirements for later iterations and engage them in clarification and testing effort on work currently under development. Focus development effort on defect fixes and help the testers catch up.

2) **Split the teams** – if diverting all the BA resources to catch up is unfeasible then at least divert some of them to this job. Solve the problem by slowing the rate of charge-ahead and speed the rate of catch-up.

3) **Suspend current work in progress and all move forward with new development** – Stop what you are doing and move everyone forward onto new work together. This may or may not be practical depending on whether future work is dependent upon work currently underway. Also it leaves a lot of unfinished work (to be tackled at the end) which is counter to the concept of lowering work in progress.

Switching iteration patterns on real world projects is complex and messy. Releases and teams need peeling apart and putting back together in an aligned way. Project managers and ScrumMasters should not try and figure this out on their own, but instead engage the team who are more intimately aware of the issues. I would suggest using a Team Solving approach to determine how best to restructure.

When to Retain the Pipelining Anti-Pattern
Each project environment is different; sometimes the political obstacles of removing pipelining exceed the benefits. Enterprise organizations that have distinct role definitions seem very prone to the pipelining anti-pattern. Occasionally getting people to undertake a wider variety of roles creates issues (usually imagined ones) about job erosion. It is not worth forcing team members into work practices they do not want to do and alienate them so much that they disengage.

A bargaining technique worth a try is to suggest that they try it for one or two iterations before deciding. However, if the pushback is strong then the next best thing is to reduce the batch size so at least the issues of pipelining as minimized.

Being aware of the issues with pipelining can help teams change to a better way of working. Many projects adopt pipelining as an interim step towards, or degenerate step from, fully integrated agile teams. It is a troublesome anti-pattern because it seems like a logical way to work for many people. However, being more appreciative of the lean principles it violates can help motivate us to correct the problem and reap the rewards of an aligned team.

Mike Griffiths is an independent consultant specializing in effective project management. Mike was involved in the creation of DSDM in 1994 and has been using agile methods (Scrum, FDD, XP, DSDM) for the last 13 years. He serves on the board of the Agile Alliance and the Agile Project Leadership Network (APLN). He maintains a leadership and agile project management blog at www.LeadingAnswers.com.