CONTENTS

Contents .................................................................................................................................................. 3
Welcome ................................................................................................................................................ 4
Business Drivers for Kanban Adoption .............................................................................................. 7
BDD: A Lean Toolkit .......................................................................................................................... 15
Lean Software Management: BBC Worldwide Case Study ............................................................ 23
Using Product Portfolio Management to Improve the Efficiency and Predictability of Teams ........ 45
Converting a Scrum team to Kanban ................................................................................................. 53
Feature Bits: Enabling Flow Within and Across Teams .................................................................. 68
Feeding the Agile Beast: Improving Economic Outcomes through Effective Alignment and Prioritization .......................................................................................................................... 84
Scrum, XP, and Beyond: One Development Team’s Experience Adding Kanban to the Mix .......... 99
Dear Attendees,

Welcome to the inaugural Lean Software & Systems Conference in Atlanta, Georgia.

In May 2009, I organized a small event in Miami. We had 18 speakers, a single track each day and only 37 paying attendees. In the two days prior to the conference, some industry thought leaders came together and formed the Lean Software & Systems Consortium.

This year is therefore the inaugural Lean Software & Systems Conference, as it is the first event that is formally organized by the Consortium. We are proud to say that our event has grown and we have around 200 attendees. This includes 43 speakers contributing talks across three tracks each day. This year we solicited presentations on a wider range of topics including business & management, systems engineering, software engineering practices, innovative edge ideas, as well as Kanban and field experience reports.

One of the goals of the Lean Software & Systems Consortium is to create a body of knowledge and provide venues for a wider community around the world to access it. We do this through the leanssc.org website and the Kanban community website, limitedwipsociety.org, as well as by organizing conferences and supporting local groups in metropolitan areas around the world. Videos of presentations from Miami and London in 2009 are available online. This year we hope to make 18 videos available from the event, plus slides and audio commentary from all 43 presentations.

Our proceedings book contains submissions from eight of our speakers. Every one of our speakers is incredibly busy so we are grateful to have these detailed papers to share with you as yet another knowledge artifact in the emerging field of Lean applied to software development and systems engineering.

Once again, our conference wouldn’t be happening without the wonderful support of our sponsors. Software Engineering Professionals organized the event and Kelly Wilson has put in tireless weeks of work to bring our conference together. Net Objectives has underwritten much of the financial risk as title sponsor and Pillar Technology was very generous as our reception sponsor. All our other sponsors and exhibitors are invaluable and we couldn’t do it without them: VersionOne; Rally Development; Bandit Software; Boeing; Target Process; Ultimate Software; Methods & Tools; and Agile Development Practices (SQE).

Our conference relies entirely on volunteer effort. In addition to Kelly Wilson, Janice Linden-Reed, Eric Landes, Dennis Stevens, Andrea Bain, Chelsea Carrato, Mikiko Fujisaki, and Aaron Sanders all deserve a mention, as well as a special thanks to Eric Willeke for bringing this proceedings book together.

And finally, as we start our event, you’ll be aware that the skies of Europe have been closed by the volcanic ash from Iceland. As a result, we've lost some of our best speakers and presentations. Luckily, several of them contributed to the proceedings book, so please enjoy the submissions from Elizabeth Keogh, Peter Middleton and David Joyce, and Mattias Skarin. At the time of going to press,
we believe they will not make it and we’re scrambling to find great presenters to fill their spots. Their presence will be missed and we’re all fortunate that we can read their submissions here in the proceedings.

I hope you enjoy our inaugural 2010 event, that you will spread the word, and that we’ll see you back again in Los Angeles in 2011. Bring your friends and colleagues and work with us to grow a long-lasting healthy Lean Software & Systems engineering community.

Best regards,

David J. Anderson
Chairman, Lean Software & Systems Conference
Enabling Agile Transformation
Strategically Aligned, Business Driven, Human Powered
Agile Capability & Business Analysis
Kanban Training
Strategic Project Execution

www.synaptus.com
BUSINESS DRIVERS FOR KANBAN ADOPTION

David J. Anderson

ABSTRACT
Senior business leaders indicate their greatest concerns with technology development are a lack of predictability, an inability to enable business agility, and a lack of good governance, cost control, transparency, and quality decision making. Agile software development has done little over the last decade to address these concerns. What has been missing is a focus on improved organizational maturity. The desired outcome, a predictable, well-governed business that responds rapidly to market threats and opportunities, reflects a high level of organizational maturity similar to that described in level four of the CMMI.

In recent years, several teams adopting the use of Kanban systems have been observed to rapidly improve their organizational maturity. The emergent kaizen culture and Lean software development processes appear to enable these high maturity behaviors. Introducing Kanban is empirically observed to deliver the outcomes desired by senior business leaders.
BUSINESS DRIVERS FOR KANBAN ADOPTION

When I speak with senior executives at client companies I regularly hear the same frustrations expressed. They want their technology development organizations to be more reliable and predictable. They want to be able to make promises to the market, to their customers, to the shareholders, and to their colleagues around the executive committee table, and to be able to keep those promises. They want to be seen as reliable, and to do so they need a technology organization that delivers predictably. They hold this reliability and predictability attribute above all others. The most important thing for their business is to deliver reliably and to have predictability in technology development.

Secondly, they tell me that their business needs to be more agile in the marketplace. They need to respond to opportunities and competitive threats more quickly. They need business agility: speed to market and shorter cycle time on customer orders. These are the metrics they want to be measured by, and they want and indeed need better performance.

A distant third on this executive wish list is better cost control. As the economy contracts, saving money is essential. While finance is hard to find in the current economy, liquidity is in short supply and revenues are falling, senior executives know that they need to continue to innovate. They need to continue to bring new products to market and technology projects to fruition. Many businesses rely on technology to give them a competitive advantage: investment banks, brokerage firms, telecommunication firms, logistics and supply chain firms, stock photography companies, news wire services, investment newsletters, automotive component manufacturers, media firms with magazines, TV, consumer electronics firms, telecommunication equipment manufacturers, music and movie franchises, and retailers. These all rely on technology to compete in the marketplace.

Meanwhile, senior executives are frustrated. Their frustration is born out of their lack of understanding of technology and software development, and their lack of visibility into what’s happening. All too often technology development is an opaque black art. The middle management reports progress but the data can never be trusted. Projects are green and “on schedule”, then suddenly they are red and behind and it’s too late to intervene. Promises get broken. Budgets are exceeded. Deliveries are late. Quality is poor and customers are dissatisfied. Underlying all of this is a suspicion that investors’ funds were not optimally allocated at best and perhaps even squandered in an ad hoc fashion. The senior people feel powerless to help. They want more transparency and more objectivity. They want to understand how decisions are made. They want to be able to pass scrutiny over them and perhaps to intervene and help before it’s too late. They want to have confidence in their middle management team.

I have had the pleasure to work for senior business leaders such as Bill Gates and the leadership team at Sprint PCS and observe them in action. I’ve seen the frustration amongst leaders of large firms that somehow all these smart technical people can’t just work it out. Why is it, they ask, when there is so much intelligence, so much knowledge, and decades of experience available, that a team of technical people cannot work out how to deliver a project on time, with agreed function, high quality, and within a reasonable budget? What are they missing?
In the past decade, Agile software development has become the latest management fad. In the First Class cabin of airliners to formal briefings in the corner office with analyst firms, senior executives have been told that the solution to all their problems is to embrace Agile methods. However, I regularly encounter these executives several years after they followed this “go Agile” advice, still with the same complaints and the same desires. Their technology teams are still unreliable. Their business is still not responsive enough and hasn’t acquired the anticipated agility. Costs are still not under control. What is missing?

The reality is that while Agile software development is necessary, it isn’t sufficient. What these executives have failed to understand is that their organizations are immature and chaotic. Simply improving the ability to write, test, and deploy software isn’t enough. The missing skills and the maturity of their organization and culture are not addressed by Agile methods. Delivering on the business objectives they have expressed requires the adoption of a managed business process for technology development.

A simple question I often ask when encountering a client for the first time to make a quick assessment of their organizational maturity is, “When something goes wrong on a project, does everyone panic?” If the answer is “No” then I ask, “When responding to the problem, does the organization follow a known procedure for dealing with issues and resolving them?” If the answer to both questions is “Yes” then I know I’m dealing with a more mature organization. However, all too often the questions aren’t met with answers but with laughter and derision. Most organizations know that they are in chaos and constantly fire fighting. What their leaders have failed to connect is that this chaotic state is mutually exclusive with their business goals of reliable delivery through predictability and business agility through agile development methods with an underpinning of solid cost management.

To reach their business goals, executives must pursue a strategy of improving the maturity of their organizations whilst implementing Agile software development methods to give them high quality products delivered regularly with predictability against short lead times.

Developing the required level of organizational maturity is a well understood problem. It’s been studied for well over 20 years and the teachings are based on studies of many large companies who have collaborated with government agencies and universities to document how organizational maturity develops and emerges.

Even at the most basic of levels, managers must take action to encourage development of skills and capabilities that mature the organization. Capabilities such as issue management and resolution that define how to respond to emerging problems, analyze them, discuss symptomatic and root cause fixes, and then choose a tactical work around or strategic root cause elimination. It’s essential that proficiency in a set of practices known to contribute to organizational maturity is achieved in an institutional fashion. It is not good enough that one or two people “get it”. The whole organization must “live it”.

Empirical study over almost 30 years has shown that some practices are foundational and must be developed first. These are the lower maturity practices and behaviors. There is no shortcut. If an
organization is not good at these things then any higher level behaviors will be unstable or unreliable.

Other practices are dependent on these foundational practices and build upon them. For example, good risk management is not possible without basic planning, tracking, and issue management.

It’s important for managers not to stress an organization by expecting behavior and performance that requires proficiency in a practice where sufficient foundational behavior is not in place. For example, you wouldn’t walk in to a second grade class and ask them to solve quadratic equations. While many of the children in the class will one day be capable of doing so, the gap between their basic understanding of math in second grade and the required level to be capable of tackling a problem like a quadratic equation is too great.

I talked with a client’s senior executive in 2008. This financial expert expressed concern that the technology organization was not re-using sufficient amounts of code and was often duplicating functionality. Wondering why, and asking for help from me, I replied to him, “But your organization struggles to schedule meetings; how can you possibly expect a high maturity behavior like code re-use to exist until you fix these basic behaviors?”

At this point it’s worth mentioning that Agile software development could never be the silver bullet it is purported to be, simply because Agile methods do not address all the capabilities required to truly mature an organization. Agile methods say little about risk management and have no real guidance on making decisions based on objectively assessed alternatives and options. Again, while Agile methods are necessary, they are never sufficient to meet the business objectives expressed by many business leaders.

Developing capabilities and organizational maturity requires investment in appropriately tailored processes and tools to support those processes. Processes need to fit the specific organizational situation and align with the existing maturity of the organization. Again, it’s no good asking for behavior that is beyond the capability of the organization and no use asking its people to use tools for which they lack the capability to use.

For example, you might ask that second grade class to use a pocket calculator to do basic addition and subtraction. They might even experiment a little and discover multiplication and division, but they’d be unlikely to know what to do with it, as they lack the underlying concepts and model to comprehend multiplication or division. Equally, it would be futile to ask them to use a computer program to solve quadratic equations. While the software may encapsulate the knowledge required to solve the equations so that the seven-year-olds don’t need to know how it works, they still lack the skills to fully control the computer and get reliable results consistently.

So managers must be prepared to invest in process and tooling and to continually revise those processes and tools as the organization matures and its capabilities improve.

It’s important to set an expectation of the desired final outcome. That outcome is a business that is reliable through predictable technology processes that delivers both business agility through agile development methods and also sound fiscal governance through transparency and objectivity that
lead to better quality decision making. To deliver on this final outcome regardless of how long the
journey may take, a business needs to invest in a full application lifecycle management (ALM) tool
at the beginning. It must leverage this tool to provide the basic tracking or work that will provide
the underlying data for all higher level decisions. A good tool will provide the whole organization
with the transparency required to improve decision making and develop good sound governance
and control costs.

Management needs to decide that it wants this final outcome. That it isn’t just talk or corner-office
day dreaming. There needs to be a management commitment to develop a focused initiative around
the pursuit of organizational maturity. Gaps in skills and capabilities should be identified and
positive action including training, coaching, process, and tools development or deployment taken in
order to close the gap. The result will be an ever-improving and maturing organization that
gradually gets closer to delivering the desired business outcomes. This process will take time. There
is no shortcut to developing proficiency in all the skills required for a high maturity business.
Expect it to take 18 months on the low end and up to five years in slower, more conservative, more
bureaucratic organizations.

The challenge of all of this is that change is hard. Asking people to change their behavior is
challenging. The skill is in knowing how to nudge an organization towards the goal with a minimal
level of resistance. Is it possible to change a culture so that the workforce embraces change and
continuous improvement and believes in the goal of a reliable business that delivers its promises to
the market, in a competitive timely fashion, while exhibiting sound governance and cost
management? I truly believe so!

Back in 2000 when I first tried to make these kinds of changes at scale with Sprint PCS in Kansas
City, I ran into resistance. I had been running a very successful software development department
in the Internet business unit, Sprintpcs.com. It had come to my vice president, John Yuzdepski’s
attention that my team was meeting its deliverables and appeared to be highly productive and
producing very high quality software. He wanted this success replicated across the organization
and asked me to work with my colleagues to introduce our techniques to the whole business unit. I
believed that I needed positional power to force change upon the organization. If only I had a higher
rank in the organization, people would acquiesce with my guidance and everything would be fine. I
learned a valuable lesson. You don’t get promoted just because you feel you need the authority to
get things done. My boss and his boss, Chief Marketing Officer Scott Relf, coached me that I had to
find a way to influence change without positional power. Unfortunately they didn’t have any
suggestions on how to achieve this. I failed to solve this riddle. How did you create change without
positional power? How did you influence and lead people to change their behavior without
authority over them?

In 2002, frustrated and burned out, I left Sprint PCS and joined a startup company that was swiftly
purchased by Motorola’s PCS division – the unit that makes cell phones. Now working in a Seattle
branch office rather their Libertyville, Illinois headquarters I once again had the challenge of
leading change without positional power and in this case with little to no political influence. Again, I
failed but I was learning.
During my time at Motorola I wrote my first book, Agile Management for Software Engineering – Applying the Theory of Constraints for Business Results. As I researched this work and developed the manuscript, I came to the realization that the first principles of Eli Goldratt’s Theory of Constraints held a potential answer to my riddle of changing behavior without positional power and significantly reducing resistance to change. My thesis was that a transparent value stream with suitable tracking and reporting would allow the bottleneck on performance to be recognized. Management could then make the equivalent of a laser guided precision intervention and everyone on the team would understand why it was necessary and what the benefit would be. I had come to realize that prescriptive change based on some process recipe developed out of context represented too much change too quickly.

I got the opportunity to try this technique in 2004 while coaching a manager in Microsoft’s IT department in Redmond, Washington. The results were spectacular. The productivity tripled and the lead time shrank by 90%. The team delivered with a 98% due date performance. I appeared to have solved the riddle – at least in part. Together with Dragos Dumitriu, the manager, we had dramatically improved performance with a minimal resistance to change. We had delivered on key business drivers of reliability, agility, and productivity. Over the following years I’ve learned more. I documented much of my new knowledge in my second book (Anderson, 2010). To lead enterprise scale changes that will transform your organization and deliver reliably with speed, agility, good governance, and cost control, you need to pursue organizational maturity. One way of doing this is with a Lean initiative catalyzed by the introduction of a Kanban system. Kanban enables you to lead change with minimal resistance. As I’ve mentioned earlier, I believe that Agile software development has a role to play in delivering the desired business results. However, Agile may be necessary but not sufficient. Attention must be paid to the maturity of the organization. Skill must be used to introduce the right changes at the right time to minimize resistance and changes need to be relevant and in context.

I’ve discovered that introducing Kanban systems as a catalyst for change is not only a way of introducing Lean concepts in an evolutionary incremental fashion but is also a way of gradually improving the organizational maturity of a technology development organization to deliver against the business drivers.

The Kanban technique is designed to minimize the initial impact of changes and reduce resistance to adopting change. Adopting Kanban should change the culture of the organization and help it mature. If done correctly, the organization will morph into one that adopts change readily and becomes good at implementing changes and process improvements. The Software Engineering Institute (SEI) refers to this as a capability at Organizational Innovation and Deployment (OID) (Chriissis, 2006) within their Capability Maturity Model Integration (CMMI). It’s been shown (Sutherland, 2007) that organizations that achieve this high level of capability in change management can adopt Agile methods such as Scrum faster and better than less mature organizations. 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. This has led to the criticism (Larman, 2009) that Kanban merely optimizes something that needed to be changed. However, there is now
considerable empirical evidence (Willeke, 2009) that Kanban accelerates the achievement of high levels of organizational maturity and capability at core high maturity process areas such as Causal Analysis and Resolution (CAR) and OID. When choosing to use Kanban as a method to drive change in your organization, you are subscribing to the view that it’s better to optimize what already exists because doing so is easier, faster, and meets with less resistance. You should also understand that the collaborative game aspects of Kanban will contribute to a significant shift in your corporate culture and its maturity. This cultural shift will later enable much more significant changes, again with lower resistance than if you were to try and make those changes immediately. Adopting Kanban is an investment in the long term capability, maturity, and culture of your organization. It is not intended as a quick fix.

Perhaps one of the strongest examples of Kanban resulting in higher maturity is presented later in these proceedings by Pater Middleton and David Joyce describing their work with BBC Worldwide (Middleton, 2010).

I believe, as we go forward as a community in 2010 and 2011, we will see more and more evidence that Lean initiatives catalyzed with Kanban systems lead directly to improved organizational maturity and as a result begin to deliver the true needs of senior executives and sponsors. I fully expect more presentations on Lean, Kanban, and high maturity in the proceedings of future conferences.
REFERENCES


BDD: A LEAN TOOLKIT
Elizabeth Keogh

ABSTRACT
Behaviour Driven Development, often described as “Test Driven Development done well”, is strongly aligned with Lean principles. It’s a pull-based approach to delivering software that matters throughout the whole software lifecycle, from vision to code. It minimises rework and over-delivery, bakes quality in, and is founded on respect for people in their various roles and the language that we use to collaborate and communicate. BDD has its origins in NLP and the Sapir-Whorf hypothesis: the idea that the words we use affect the cognitive model of our work. It decouples the learning associated with TDD and Acceptance Testing from the word “test”, using the more natural vocabulary of examples and behaviour to elicit requirements and create a shared understanding of the domain.

In this paper, Liz Keogh introduces BDD, shows how to model it using PDCA, and demonstrates why leveraging learning as a constraint in projects can be valuable.
BDD: A LEAN TOOLKIT

A SHORT HISTORY OF BDD
Dan North originally envisaged BDD as a small, simple change from existing practices – replacing the word “test” with the word “should” as a neuro-linguistic experiment, to see if it helped alleviate the cognitive dissonance that some development teams experienced when trying out TDD or Acceptance Testing for the first time. In its early days, BDD only covered a small fragment of the value stream, and was focused on helping developers produce the right code for the requirements.

However, Dan has always said that BDD should be about “writing software that matters” – software which ships, gets used, and makes a difference. As part of a development team, we can write the most perfect, beautiful, bug-free code, and it still won’t matter if the requirements were the wrong ones in the first place!

Many of the complaints and difficulties I addressed around early BDD were similar.

- “You tell us to write scenarios to describe the system’s behaviour using the business’s language. How do we do that if they don’t care enough to have the conversations?”
- “Why are you writing software that people don’t care about?”
- “We’re using up this year’s budget.”
- “We’ve got three developers free for a month.”
- “There’s a bonus for managers who can deliver successful projects.”

I replied that BDD wasn’t designed for solving that problem. At the time, I believed it to be true.

BDD HELPS US DELIVER VALUE BY DEFINING BEHAVIOUR
BDD’s patterns are rather nicely encapsulated with that magic word, “should”. When we consider the work that we’re doing, we think about:

- What should this do? (Behaviour)
- Why should this do it? (Value)
- Who / what needs this? (Stakeholder or Consumer)
- What shouldn’t this do? (Scope and Responsibility)
- How will we know we’ve got it? (The Tests)

When developers write a BDD example or scenario, we often start with the events and outcomes. The event defines the scope of the behaviour. The outcome is the goal we’re after. We’re not always sure, when we define the outcome, exactly what it is that it should do. Writing it as a test – an example of the kind of thing we might expect if our goal is met – helps to clarify our thinking.

Sometimes, though, the context in which the scenario takes place will be different. We ask, “Does this always happen? Is there a different context that, for the same events, should produce different outcomes?” This lets us find missing contexts, which may affect both the outcome and the scope of the work.

- Given this context,
• When I perform this event,
• Then I should get this outcome.

**Figure 1: Finding New Contexts**

When we’ve defined the examples that describe the behaviour, we do just enough to make those examples work, then we move on to the next piece of work that needs to be done. Here’s a scenario from a retail store:

- Given John has purchased a Barrylux 20k DVD player for £200
- Given the Barrylux 20k DVD player works, but John doesn’t want it
- When John brings it back for a refund
- Then John should get a refund of £200
- And the Barrylux DVD player should be added to stock.

We can automate this scenario, so that we are always sure that the behaviour we’re describing matches the behaviour of the system. This helps us ensure that the work is always usable, and gets around the ever-increasing cost of regression testing.

**Figure 2: The Cost of Regression Testing Rises as Code is Added**

*BDD is Focused on Learning*

On one training course, César Idrovo asked a project manager, “How long did your last project take?”
“8 months.”
“How long would it take to do it again with the same team, the same technologies and the same requirements, knowing what you know now?”

“2 months.”

“Why?”

“We’ve learnt everything that we needed to know.”

In many projects, learning is the constraint of the project. BDD focuses on learning by encouraging questions, conversations, creative exploration, and feedback. This is why we use “should” instead of “must” or “will” - because we can ask, “Should it?”

**BDD HELPS US TO PRIORITISE THE UNKNOWN**

The model of known and unknown can help us to look at the areas in which we need to learn. By front-loading areas of greatest ignorance, we reduce the risk of wasting our learning. We can do this creatively; first by exploring the domain language with the business, product visioning, future- and retro-spectives, and also by getting feedback as quickly as possible, so that the areas where we are ignorant are revealed. We do this to be able to form some ideas about our tests, around which we can start exploring, as with the scenarios.

We try to start with the largest granularity we can, because then we can find the large areas that we know nothing about. When we’ve discovered each of these, we move to the next area of risk within each, and so on. We’re looking for anything that tells us we need to re-evaluate our vision, or to find a way to address a particular risk.

Once we know where our ignorance lies, we can start to quantify the risk associated with each area. It’s not always necessary to apply maths to this! We can ask our stakeholders, “Imagine that we’re at the end of the project and it’s failed. Why did it fail? What stops you sleeping at night?” These kind of open questions are often more useful. Sometimes the stakeholders and domain experts actually have the knowledge – it just doesn’t occur to them to pass it on! So something which looks like an unknown unknown to the development team is often an unknown *known* from another point of view.

---

1 *Note. Credit for this coaching pattern goes to Ashley Johnson, Gemba Systems.*
FIGURE 3: UNKNOWN UNKNOWNS ARE OFTEN SOMEONE ELSE’S KNOWN UNKNOWNS

The known unknowns are really a mix of unknown unknowns and known knowns, just at a lower level of granularity. The only thing we can actually have certainty about is, of course, working code. So a known unknown actually looks like this:

FIGURE 4: IT’S ALL JUST A MATTER OF GRANULARITY

“A bit done” is, of course, another known unknown, with all the associated granularity below it. Eventually we reduce ourselves to two kinds of knowledge – the things that are done, and the things that aren’t started. Everything which is in progress is made up of components, which are done or not started. The relationships between those components may be quite complex – projects are more like frogs than bicycles!

IN BDD, ONLY SOFTWARE THAT MATTERS IS WORKING

If we’ve been working on a differentiating project, then we are doing some things that nobody has done before. Because nobody has done them, we have no guarantee that they will work, at whatever level of granularity they are different. We may not know:

- that they will work technically.

---

that the necessary contexts have been covered.
that the requirements have been interpreted correctly.
that the software will be usable.
that all the necessary stakeholders have been taken into account.
that the vision is achievable.
that the vision will matter once it’s released.

These are tests at various levels of granularity. There are other tests between these that we could perform; it’s not a complete list.

In the early days of BDD, we focused on the correct interpretation of the requirements, and considered our work “done” when our implementation of the requirements was complete and tested. The same patterns that we use in scenarios can be taken to higher levels. In the same way that we can look for contexts which affect our outcomes, we can look for missing stakeholders, market forces which haven’t been considered, etc. Our learning has now moved up the value stream.

Feature Injection, the brain child of Chris Matts, takes the patterns of BDD up into the analysis layers. In simple terms, it works like this:

- A primary or core stakeholder has a vision.
- In order to make the vision succeed, the goals of some incidental stakeholders must also be satisfied.
- The goals can be achieved through particular feature sets, or themes.
- The feature sets can be split into features and stories.
- The stories can be defined using scenarios.
- The code to make a scenario work can be described using examples.

At each stage, we perform the same steps of learning, defining our test, performing the work, checking to see if the work has been done, and acting on the feedback. The act of performing the work, of course, requires another round of planning, doing, checking, and acting on feedback!

**Figure 5: Plan, Do, Check, Adapt at Different Granularities**
Because of this, we can only define a test for the particular granularity in which we’re working, which is to say, a fragment of the value stream. We can write our tests at any level.

**BDD Focuses on Stakeholders**

Sakichi Toyoda developed the 5 Whys, in which we ask “Why?” 5 times, and come to the root of the problem. We perform a similar process with Feature Injection. Ideally, when we come to deliver working software, we have an understanding of the vision. If the vision isn’t clear, we can find it by asking “Why are we doing this?” until we reach the point where we are able to say that we’re making, saving, or protecting money.

Another way of asking this question is, “What will this get for you?” In NLP, this is called “chunking up”. Eventually, we find out what the vision is – or perhaps, that there isn’t one!

Of course, nothing really matters unless the software is delivered into production and starts making money! It’s still useful to think about progress and feedback in between the creation of the vision and the implementation and delivery of it. So when do we do this?

We can only get learning and feedback from someone else! So, the point at which the value being delivered moves from one person in the chain to the next, or one team to the next, is a good point at which to write our tests – because we have real stakeholders, consumers of value, who can tell us whether our tests are accurate.

This is true of all our tests – whether we’re defining how much money we think we will make, how much market share we’ll take, how easy our application will be to use, how performant our system will be, if our architecture is maintainable, whether our scenario is achievable, or whether our code behaves as it should.

---

3 We don’t yet know whether this works for projects which aren’t focused on money, eg: charities, not-for-profits, hospitals, etc.
FURTHER READING
http://www.infoq.com/articles/pulling-power
http://dannorth.net/introducing-bdd
http://skillsmatter.com/podcast/agile-testing/how-to-sell-bdd-to-the-business

Domain Driven Design, Eric Evans

Waltzing with Bears, Tom de Marco and Timothy Lister
Lean Software Management: BBC Worldwide Case Study

Peter Middleton & David P. Joyce

Abstract

This case study examines how the Lean ideas behind the Toyota Production System can be applied to software project management. It is a detailed investigation of the performance of a nine-person software development team employed by BBC Worldwide based in London over a 12 month period.

The data was collected by one researcher in 2009. It involved direct observation of the development team, the Kanban boards, the daily stand up meetings, semi structured interviews with a wide variety of staff, and statistical analysis.

The evidence shows that over the 12 month period, lead time to deliver software improved by 37%, and consistency of delivery rose by 47%. Output increased, while defects reported by customers fell 24%. The significance of this work is that it shows how to considerably improve the performance and consistency of Agile software development techniques.

The conclusion is that the adoption of Lean and Kanban ideas has enabled the maturity of an Agile software process to move rapidly towards CMMI Level 4. The faster, more responsive delivery has reduced both technical and market risks. The drawbacks are that it may not fit well with existing corporate standards and managers may find their new role more challenging.

Index Terms - Lean, software, agile, CMMI, lead time
LEAN SOFTWARE MANAGEMENT: BBC WORLDWIDE CASE STUDY

INTRODUCTION

Lean thinking is important because it can reduce error rates to 1 per million units. It has been shown without question to have the potential to at least double the productivity of both manufacturing and service organisations. It also significantly reduces the time taken to deliver new products while substantially reducing cost. The evidence from Toyota (Japan), Porsche (Germany), and Pratt & Whitney (USA) confirms this (Womack et al., 1990, 1997).

Applying the ideas from the Toyota Production System (TPS) (Shingo, 1981) or Lean thinking to the management of software projects therefore promises great improvements. This case study records the practical experience gained between April 2008 and October 2009 by the London based BBC Worldwide when it applied Lean thinking to managing software development.

BBC Worldwide is the main commercial arm and a wholly owned subsidiary of the British Broadcasting Corporation (BBC). Its mission is to create, acquire, develop, and exploit media content and brands around the world in order to maximise the value of the BBC’s assets for the benefit of the UK licence payer. In 2008/09 BBC Worldwide generated profits of £103 million (before exceptionals) on revenues of £1.004 bn. (BBC, 2010)

The basis of Lean is the absolute elimination of waste (Ohno, 1988). This requires a focus on the flow of work through the system to ensure that material is produced only when it is needed and in the exact quantities required. This enables near zero inventory levels to be approached, which makes production more flexible and also allows sources of defects to be quickly identified.

Waste is defined as anything that does not produce value for the customer. The objective is to achieve the just-in-time delivery of materials. The Lean techniques are only of use if there is a commitment to eliminate waste and make fundamental, continuous improvements to the production system.

‘Perhaps the most striking difference between mass production and lean production lies in their ultimate objectives. Mass producers set a limited goal for themselves – “good enough”, which translates into an acceptable level of defects, a maximum level of inventories, a narrow range of standardised products. To do better, they argue, would cost too much or exceed inherent human capabilities. Lean producers on the other hand, set their sights explicitly on perfection: continually declining costs, zero defects, zero inventories, and endless product variety.’ (Womack et al., 1997)

LITERATURE REVIEW

The first recorded experiments with Lean software development were by Middleton (1993). Microsoft reported how the Lean mistake proofing of a software process eradicated whole classes of errors. Tierney (1993) and Hou (1995) for the US Department of Defense concluded Lean techniques were the only way forward. Morgan (1998) from Cummins Engine Company ‘...provides some compelling evidence that the ideas of lean manufacturing are indeed applicable, in principle, to software development.’ Hamilton (1999) with the Department of Defense concluded that:
'...shifting to lean principles improves cycle time reduction and overall quality in the software development process.'

Lean software development is an evolutionary, incremental approach as advocated by Gilb (1988). Although it has different intellectual roots it has much in common with Agile software development (Cockburn, 2002). But the reliance on data within Lean means that it has the quantitative rigor required by CMMI Level 4 (Chrissis et al., 2004). The mathematical basis of Lean is well described by Hopp and Spearman (2001).

Timberline Software in Oregon in 2002 with 450 staff was the first recorded full industrial implementation of Lean software development. They reported considerable improvement but their focus on setting a common tempo or ‘Takt’ time for software development based on similar sized work units has now been superseded. (Middleton et al., 2005)

The Lean software ideas were developed by Poppendieck (2003) and Middleton and Sutton (2005). Anderson (2003) and Ladas (2008) provide valuable insights and perspective. Moving upstream and applying Lean thinking to influence project selection and definition creates great benefits (Seddon, 2005). The proceedings of the first Lean & Kanban Software conference (Willeke et al., 2009) and the work of Shalloway et al. (2010) show adoption is spreading. However, there is a clear need for more rigorous case studies of implementations.

**RESEARCH METHODOLOGY**

The research focus was to establish the benefits and costs of using Lean and Kanban to manage software projects. The research method was for an experienced researcher to observe and write up the operation of the BBC Worldwide Webmedia Department’s software processes. The seven visits to London of 2–3 days each took place between June and October 2009. These were supplemented by numerous phone calls and e-mails.

The advice of Eisenhardt & Graebner (2007) that with case study research ‘...close adherence to the data keeps researchers “honest”.’ was followed. The pure positivist approach is that truth can always be discerned from untruth, and that the truth can be discerned either by deduction or by empirical support and by no other means (Jankowicz, 1991). The interpretivist model is that an in-depth understanding of a phenomenon may only be gained by studying it in context from the participants' perspectives (Walsham, 1995).

For this study the position of Wynekoop and Russo (1997) that both approaches are useful was adopted.

Seaman (1999) emphasises the importance of ‘triangulation’ in gathering data from as many different sources as possible to assist accuracy. Data was collected from:

- The most mature software development team, called Digi-Hub.
- Semi structured interviews with developers, project managers, business analysts, and managers.
- Walking through the operation of the Kanban boards that visually displayed the flow of work, so enabling it to be controlled.
• Recording the precise operation of the Kanban boards.
• Observing the daily ‘stand up’ meetings at the Kanban boards where work allocations were discussed and agreed.
• Review of statistical analysis of the outputs from the system.

In Easterbrook’s (2008) terms this is an exploratory case study. The research is asking knowledge questions focused on recording and understanding how the system was operating.

The key features of a Lean system are:

• Levels of work in progress (WIP) e.g., requirements, designs, and code, are deliberately kept as low as possible.
• Work is ‘pulled’ into the software development system only when there is capacity to work on it, rather than ‘pushed’ in regardless of capacity available.
• ‘Autonomation’, where technology is used to speed the flow of work through the process.
• Process improvement and waste elimination are routine.
• Visual indicators used extensively so even a casual observer can see the status of the work in progress.

RELIABILITY OF THE DATA COLLECTED
With a case study there is a danger of bias in the data collected which would undermine or destroy the validity of the results reported. The following areas were reviewed to identify any possible distortions in the data.

Time Line
The implementation of Lean and Kanban was started in April 2008. Due to the necessity to stabilise the processes and adapt to the changes, data collection did not start until three months later in August 2008. The data used in this paper refers to the 12 months from October 2008 to October 2009.

Size of Projects
The data shows that the size of the projects being handled over the 12 months of the study did fluctuate, but there is no clear trend towards smaller projects. This means the results are not biased by project size.

<table>
<thead>
<tr>
<th>Month</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/11/2008</td>
<td>11</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>01/12/2008</td>
<td>5</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>01/01/2009</td>
<td>7</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>01/02/2009</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01/03/2009</td>
<td>18</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
While the units of work going through the system were deliberately made as small as possible, the size and nature of the projects themselves were unchanged.

**Complexity of Work**
If the projects became less complex over time, then this could be a source of bias in the results. The list of all the work undertaken was reviewed. It was clear the work was very varied and came from different sources. There was no evidence to suggest the complexity had changed. A reduction in the complexity of the work was therefore not distorting the results.

**Governance Arrangements**
The structure was:

- Business Board (Strategy & budget)
- Project Board (Detail & authorise specific work)
- Product Owner (Reconcile business & customer wants)
- Users requesting work (10 sign off work completed)
- End users (200 - 300 people)

Note: End users are those internal to BBC Worldwide. The digital assets created were ultimately used by millions of people.

This governance structure had been unchanged since before April 2008, but over the period of the study it was reported that stricter identification of the business benefits was required before projects were authorised. This may mean projects are better thought-through as regards return on investment, but at the technical level the work was unchanged.

**Composition of Team**
The team personnel were the same since October 2008 with the same Project Manager. The data reflects the work of all the team, not just selected high performers. Also, all work carried out,
including low priority and legacy improvements tasks is reported. The skills of the team improved over the 12 months.

**Engineering Practices**
Work to improve engineering practices started in April 2008, which involved the following:

- Test Driven Development (unit tests)
- Automated Acceptance Testing (main suite completed April 2009)
- Source Control Software
- Bug Tracking Software
- Decoupling – improving legacy software (April – July 2008)
- Minimal Marketable Feature (MMF) introduced April 2009

This resulted in higher test coverage and releases increasing from one every two weeks to almost daily. The same engineering practices were in place but their use was consolidated and improved over the period studied.

**Overview of Data Collected**
The size, complexity, type, and volume of the work handled did not change materially during the 12 months of the study. The team and the governance structure remained comparable. The engineering practices were improved but most were in place by October 2008. Certainly a stable team with better tools would be expected to show improvement over a 12 month period. The only other difference was the introduction of Lean thinking, which will now be explored in more detail.

**DIGITAL HUB (DIGI-HUB) TEAM**
In 2009 the team had operating costs of £1.5m and a development budget of £675K. It was made up of nine staff: Project Manager, Business Analyst, Software Architect, Tester, Lead Developer, three Developers, and a Support Developer. It was working on a mix of developing new software and software maintenance. The technologies used were C#, .NET, MS SQL Server, and legacy Connected Service Framework (CSF) code.

In April 2008 the Kanban boards were installed and all the value adding stages of the development life cycle (value stream) were drawn onto them. Restrictions on the amount of WIP allowed at each stage were put in place. All work at each stage was then listed on the boards. The Project Board agreed on the priorities to be released in the next three months.

To break the work down into smaller units that could be delivered more quickly, the concept of Minimum Marketable Features (MMF) was adopted. This is a chunk of functionality that delivers a subset of the customer's requirements that is capable of returning value to the customer when released as an independent entity. These are then broken down into Stories (New Features) and then further into Tasks, which are just ‘To Do’ items.

**Office Layout and Work Flow**
Office layout is a key component of success. In the Toyota production lines there are lights displaying the status of production at any time; the same principle can be applied to software development. Information radiators and Kanban boards were placed all around the work space to ensure that progress on a project was completely transparent and available for all to see (Fig. 1). This enabled team members to be self managing.

The strategic direction and prioritisation of work was still set by the Project Board and the Steering Group, but the delivery team now had a much clearer idea of their capacity and current WIP. In the Digi-Hub project, two Kanban boards (A,B) and four information radiators (C,D,E,F) were used and positioned as shown in the diagram below. The layout of the boards evolved as the projects and staff understanding progressed.
It is important that work flows are kept as stable as possible. This is because sudden peaks and troughs of work are disruptive and will damage productivity. It is therefore necessary to try and influence the upstream work flows as much as possible. This information is captured on Kanban board A: the ideation pipeline (Fig. 2).

<table>
<thead>
<tr>
<th>Proposed Ideas</th>
<th>Ready for Decomposition</th>
<th>Decomposed Engineering Ready</th>
<th>Released</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waste</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On Hold</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1: Layout of Kanban Boards and Information Radiators**

**Figure 2: Kanban Board A: Ideation Pipeline**
Any ideas or potential work from customers were recorded on a card and retained in Proposed Ideas in case they trigger suggestions from the team. Any work that is abandoned or has its priority changed is also recorded. Once the ideas have been clarified and broken down into small deliverable units then they are ready to be ‘pulled’ to the next board when the team has capacity to work on them. This Kanban board B (Fig. 3) tracks the progress of Minimum Marketable Features (MMF), Stories and Tasks:

![Kanban Board B](image)

**Figure 3: Kanban Board B: (Development Phase)**

Any problems in Development (Dev) will quickly become apparent, as they will reach their WIP limit of four Minimum Marketable Features (MMF) and become a visible bottleneck.

**Daily Stand Up**

The daily stand up lasts for about 15 minutes, and normally starts about 10.15 a.m. each morning. It is carried out with all team members standing in front of Kanban board B, which tracks the Development Phase. See above diagram (Fig. 3). This is because this is where the bulk of the work is carried out. The daily stand up is vital for the operation of the Lean system. It is essential to facilitate the identification and removal of blockages and bottlenecks, and update the status and prioritisation of work items.

The structure of the daily stand up rhythm is given on information radiator C (Fig. 4). Firstly, everyone checks to ensure his or her work status is correctly displayed. Secondly, anyone who is ‘blocked’ and unable to progress due to something outside his or her control reports this, and appropriate action is decided to remove the obstruction. Thirdly, any clusters of cards indicating a bottleneck are noted and the people reorganise to alleviate this. Lastly, the work is reviewed to see if priorities have changed or if the work flow can be improved. There is an Expedite work flow which can be used in exceptional circumstances to accommodate urgent items or if there is a high priority late change.

There is no need or time in a large team for an individual report from each person. It is more effective to just flag problems to be resolved. The Kanban boards make it clear to all the team the
exact status of progress, blockages, and bottlenecks, and it also signals possible future issues for
which to prepare. This shared information enables the team to self organise to ensure the work
flows smoothly. The different coloured cards used are listed below (Fig. 4) and enable the exact
status of all the team’s work to be seen at any time.

**Stand Up Rhythm (Daily)**

<table>
<thead>
<tr>
<th>Data</th>
<th>Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocker</td>
<td>Purple</td>
</tr>
<tr>
<td>Bug (in QA / UAT)</td>
<td>Pink</td>
</tr>
<tr>
<td>Fixed delivery</td>
<td>Small red (attached to card)</td>
</tr>
<tr>
<td>Expedite</td>
<td>Small yellow (attached to card)</td>
</tr>
<tr>
<td>New Feature</td>
<td>Yellow</td>
</tr>
<tr>
<td>Live Defect</td>
<td>Red (bug found by customer, high priority)</td>
</tr>
<tr>
<td>Technical Story</td>
<td>Green</td>
</tr>
<tr>
<td>Something missed</td>
<td>White</td>
</tr>
<tr>
<td>Kaizen</td>
<td>Blue</td>
</tr>
</tbody>
</table>

**Figure 4: Information Radiator C: Kanban Board and Team Performance Indicators**

Information Radiator D: Release notification and daily support process tasks

The board was used to ensure that any scheduled releases or other operations that have to be
carried out during the week are visible. It acts as a reminder and check.

Information Radiator E: Architecture, estimating, and breaking down projects

The board was used to record decisions on the architecture for the software, initial estimates, and
how the work had been broken down into MMFs.

Information Radiator F: Kaizen board and technical debt

The team voted on which items they wished to work on to reduce technical debt or improve the
Lean system. Reducing technical debt involves work such as improving poor legacy code or making
a modification that could increase future productivity. Legacy software can be a severe constraint
on current productivity. It is therefore necessary to explicitly reduce any technical debt by allowing
time for improvements to be made, even though these are invisible to, and not requested by, the
customers.
**PERFORMANCE DATA ON THE LEAN SOFTWARE SYSTEMS**

Lean systems are typically rich in data to enable a team to be self-organising and initiate continual improvement. To evaluate the effectiveness of Lean software management over 12 months of operation, some of the data used by the team is presented here. The team monitored the time taken for work to flow through various parts of the value stream. They also tracked quality and throughput measures.

**Lead Time**

Lead time is the total elapsed time from when a customer requests software to when the finished software is released to the customer. It is measured because it tracks how quickly and reliably software is delivered to customers. Lead time is defined as the number of working days the work takes measured from Kanban board A: Decomposed Engineering Ready to Kanban board B: Release Ready.

The main work requests are New Features, which are a subset of an MMF. Other work would include Technical Features and Live Defects. Decomposed Engineering Ready means the customer has agreed to proceed, and it is then that the Lead Time clock starts. The items are then created for the Engineering Ready input queue. Items are ‘pulled’ into Engineering Ready only when capacity becomes available. The Lead Time clock stops when User Acceptance Testing is complete and the items have reached Release Ready.

The lead times were analyzed using time series statistical process control charts. These are powerful tools, as they can show trends over time and the amount of variance in a process. A reliable process will have low variance, so a key objective is to continually reduce variance. Fig. 5 illustrates how the top lines, Upper Control Limits, continually decline, meaning that lead times are becoming more predictable. To show trends, the periods on the charts have been split from November 2008 – March 2009, April – June 2009, and July – October 2009.

The data shows that adopting Lean has allowed the mean for lead time to be reduced by 37% or 8.4 days, from 22.8 to 14.4 working days from November 2008 to October 2009.
The 37% drop in lead time does demonstrate improved ability to respond to the needs of the business. The spread of variation has also dropped significantly, as shown by the upper control limit in Fig. 5 reducing 47% from 70.7 to 37.3 working days. This means that now the team is delivering new functionality faster and with considerably more predictability.

Projects are broken down into Minimum Marketable Features (MMF), which are then broken down into New Features, so this is tracking when customers first start to receive New Features and MMFs, rather than when the entire project is complete and delivered. Lead time was reduced by 37% and variation in delivery time by 47% in 12 months. This enabled the business to receive new software faster and with more predictability.

**Development Time**

The Development Time measure gives insight into the efficiency of development. This portion of the value stream was directly under the team’s control and not subject to delays from upstream, downstream, or third parties. It does not include Engineering Ready, QA, or related queuing times.

Development time is recorded in working days, from Kanban board B stages: Dev. Ready to Dev. Complete. The work units are either Stories or Tasks, which can be either standalone or part of an MMF.

The periods on the statistical process control chart (Fig. 6) has been split from February – March 2009, April – June 2009, and July – October 2009 to show trends. The results show that the mean development time for all the work completed has reduced 73% from a mean of 9.2 to 2.5 working days over the nine months.
FIGURE 6: VARIANCE OF MEAN DEVELOPMENT TIME

The spread of variation fell by 78% from 30.5 days to 6.8 days in the nine months that data was recorded, indicating improved predictability of delivery.

Throughput
Throughput is defined as the number of items released to customers. An upward trend would be expected as the code base was decoupled, projects were broken into the smaller units of MMMs, Stories and Tasks, and cycle time was reduced. The chart below (Fig. 7) shows the number of releases per month from November 2007 – October 2009. There was a release freeze in February hence the drop for that month.
The Releases chart does not show how much value is being delivered but it does show an eight-fold increase in releases from two in November 2007 to 16 in October 2009. This indicates an improvement in configuration management discipline and capability. The more frequent releases reduce both technical and market risk by allowing customers to evaluate tangible product rather than just progress reports.

Live Defects per Week
Live Defects are bugs reported by customers. It is vital that the reduction in lead and development times and the increase in throughput are not at the expense of quality. Live Defects are recorded on red Kanban cards and added to the Dev. (Development) stage of Kanban board B. The chart below (Fig. 8) is split between October 2008 – June 2009 and July – September 2009. It shows that the mean number of bugs reported by customers each week fell by 24% from 2.9 to 2.2.
As the number of bugs reported by customers was low, more data collected over a longer period of time would be preferred. With this caveat, this data does show that the rate of defects is predictable and under statistical process control.

The number of bugs open at any one time has fallen by 33% from an Upper Control Limit of 7.6 to 5.1. This indicates they were being fixed more quickly, possibly due to the improving structure of the code base. The necessity of allowing software developers time to improve the quality of their code was often mentioned by the team as a big factor in the improved bug rates. As legacy issues (Technical Stories) were resolved the bug rate had fallen, allowing more Customer Stories to be completed. So while teams are customer focused and responsive to customer needs, they need to pay down any technical debt to increase their productivity levels.

**Continuous Improvement**

The daily stand up is concerned with identifying and removing anything that is preventing progress. To do this ‘blockers’ are actively identified, assigned, tracked, escalated, and removed. This is a mechanism for making continuous improvement routine. Evidence of the effectiveness of this is shown below in a statistical process control chart (Fig. 9). The periods on the chart have been split from September 2008 – March 2009, April – June 2009, and from July – October 2009.

Over the 12 months the number of working days items were blocked was reduced by 81% from a mean of 25.8 days to 4.9 days. The outlier in 2008 was a result of waiting for a third party to complete its work (a special case). This is powerful data to use when discussing performance with third parties.
Actively looking for and recording blockers increased the number of blockers raised, which is beneficial. These were then being removed at a faster rate by the team as more experience was gained. This data was also used in Retrospectives and quarterly reviews. Reoccurring blockers were investigated and root-cause analysis performed. The 81% reduction in the number of working days that work was ‘blocked’ is evidence of effective continual improvement.

**AGILE v. LEAN**

This case study may appear to record an Agile / Scrum-like approach, where care has been taken to collect data. However, the Lean approach described here does have significant differences from Agile.

**Push v. Pull**

Scrum has time boxed iterations with a fixed release cadence. It is therefore in essence still a push, batch model. Lean uses WIP limits to ensure a team is not overloaded. Work is ‘pulled’ in when the team has capacity. Teams don’t sign up for arbitrary deadlines, as support issues occur that blow things off course. Arbitrary deadlines tend to lead to game playing and poor quality as attempts are made to shoehorn work into the reduced time.

**Reliance on Data**
Scrum has ‘Inspect and Adapt’ in their Retrospectives, which is a trailing indicator. The boards are not so important because in Scrum they focus more on the people rather than the work. The Scrum ‘stand-up’ directs attention to the people and what they did yesterday and what they are doing today.

In contrast, Lean enumerates the work, not the people. In Lean thinking, data is seen as a source of empowerment. The team members are expected to collect and analyse their own data so they can control and improve their own work. The Lean approach uses the Kanban boards to expose the problems and expects the team to take action. This is a leading indicator. The Lean ‘stand-up’ focuses on the work and what the team is going to release. Lean uses the data to help the team look up and down stream to enable innovation.

Continual Improvement
Scrum uses ‘Retrospectives’, but benefits from these are largely anecdotal and not quantifiable. The concept of ‘velocity’ measured as number of feature/story points delivered per iteration is often used. However, there is a risk that velocity estimates and number of feature/story points delivered are too subjective and at risk of manipulation. These teams don’t put the number of feature/story points delivered under Statistical Process Control (Deming, 2000) so the natural variation in results is not known.

The dearth of data in the Agile community generally is remarkable. A major study seeking evidence to support Agile (Dyba & Dingsoyr, 2008) reported ‘the strength of evidence is very low, which makes it difficult to offer specific advice to industry.’

In contrast, Lean uses ‘lead time’ which is much harder to game, as it records total time from when a customer requested the work to when the finished work was received by the customer. Lean looks at ‘blockers’ or impediments as First Class items to be addressed. Lean seeks data to be used by the team for self management and routine process improvement.

Multi Skilling / Collaboration
With Agile, the Scrum Master has the ‘impediment list’ but it is not the team’s responsibility. With Lean, because of the WIP limits and the visibility of the Kanban board, the staff can’t ‘cherry pick’ what they would like to work on if they are blocked. They must help with the bottlenecks and items blocking work. The focus of the daily ‘stand ups’ is on the flow of work, not on the individual reports and performances. Staff, regardless of skill, must help with the bottlenecks. The objective is to deliver value as quickly as possible to the customer.

DISCUSSION
The data shows that over 12 months this nine-person software team has achieved better quality, predictability, and throughput. The use of Statistical Process Control indicates a team that is managing process improvement in a quantitative way in line with CMMI level 4.

The main use of the Kanban boards was to give the team a clear, current picture of where exactly it was. Customers rarely visited the Kanban boards and did not attend the daily stand up meetings, as they both contained more detail than customers needed and would be time consuming. Also, a
team may be working for several customers so much of the meeting would not be of interest. Customers were encouraged to attend, but overall it was not feasible for them to be closely involved with the Kanban boards.

Managers could see the status of projects from the Kanban boards, but this was not the boards’ primary function. Managers could ask questions when the Kanban board showed several red notes, denoting live defects, or if excessive ‘waste’ was being recorded, but otherwise they were not be closely involved with the Kanban board. The regular meetings for the Steering Group or Project Committee were not held at the Kanban boards.

The boards evolved, with changes to layout being made roughly on a weekly basis. They were a living tool reflecting both the evolving project and the changing understanding of the team of their work. The social value of the daily stand up cannot be underestimated. The need to daily share progress and problems with your peers is a powerful motivator and source of discipline.

The Lean approach can handle big, complex projects. The constraint is the ability of the human mind to handle complexity, but any large project can be broken up into smaller projects. A master Kanban board can then be used to record and summarise the progress of all the smaller projects. There is no need or benefit from one gigantic Kanban board recording everything. Scale or complexity of projects was not observed to be a problem.

The Lean and Kanban approach worked better when it was supported by:

- A stable, experienced team with low staff turnover.
- A Project Manager who knew the skills and abilities of his or her people.
- Good source control software to manage different versions of the software product.
- Bug tracking software.
- Well automated release and deployment processes for software.
- Automated unit and coverage testing that reduces live bugs.

Lean and Kanban is therefore not a substitute for good software engineering tools and practice, but supplies information that enables a team to self-manage. The transparency Kanban boards provide does assist software development teams to rapidly raise their maturity levels.

Lead time could potentially be further reduced as customers were batching work for User Acceptance Testing (UAT), which was slowing the process before Release Ready. An investigation of how to make UAT easier for customers could be beneficial. The other area that could possibly be improved is in the ‘fuzzy front end’ (Smith & Reinertsen, 1998). The time from when an idea is logged on Kanban board A: Proposed Ideas to Decomposed Engineering Ready is currently not recorded and this could be significant.

The concept of a fixed ‘heart beat’ of production, or Takt time used in Lean manufacturing to ensure that the rate of production meets the rate of demand, was not used. Takt time does not work with software or services because each unit of work is different as are the abilities of each software developer. It would therefore be impossible to schedule production in this way. The solution for software is to have incremental development with frequent releases. This provides rhythm to
production and ensures a direct connection with the customer. When this is combined with the
daily stand ups, the work effort can be directed and controlled as the customer requires.

CONCLUSION
The data presented here is felt to objectively record the behaviour of this software team. No
significant bias was detected from variations in project size, project complexity, governance, team
composition, and engineering practices.

The Lean ideas of low work in progress, statistical process control, pulling work when capacity is
available, self managing teams empowered by data, daily routine of continual improvement, and
multi skilling did help to achieve the results recorded.

Lean also provided a framework that encouraged other beneficial improvements, such as rewriting
parts of the legacy code, developing team skills, and reducing staff turnover. The quantitative and
qualitative evidence indicates that the Lean ideas do provide a remarkably effective way to manage
software projects.

The data recorded the following benefits:
- The mean lead time to deliver software to customers was improved by 37% (8.4 working
days) with a 47% gain in predictability.
- Development time was reduced by 73% from a mean of 9.2 to 2.5 working days.
- Predictability of delivery improved by 78% from 30.5 to 6.8 days.
- Product releases increased eight fold, which reduced both technical and market risk.
- Live defects per week reported by customers fell by 24% from 2.9 to 2.2. The total number
  of defects open at any one time fell by 33%, showing problems were being fixed more
  quickly.
- Continual improvement was effective, with an 81% drop in the number of working days
  items were blocked.
- High staff morale, motivation, and pleasure in their work were observed.

The costs or negatives of the Lean approach are the following:
- While capital costs were trivial there is a need for considerable space to display the Kanban
  and information boards. Organizations with offices designed around a corporate 'look' may
  not welcome walls of Post-It notes.
- If the organization has a heavy plan-driven process with standardised corporate reporting
  on projects, then this emergent approach will not fit easily. Lean handles risk by being
  highly transparent, reducing WIP, breaking projects into small parts, and frequent
  deliveries. Lean does not work well with targets, milestones, Gantt Charts, and Traffic Light
  reporting methods.
- Truly delivering value to customers will require the development team to proactively move
  upstream to work with customers to define and analyse their problems and then work
  downstream after release to see if business value was actually created. Organizations may
  feel the IT teams are going beyond their remit.
A self managing team can be challenging, as managers need to move toward a facilitating role which they may feel uncomfortable with. Staff may not be used to being encouraged to identify problems or having to multi skill.

**Future Developments**

It is clear that there is much potential in the Lean and Kanban approach to managing software projects. It shows how the Agile approach can be improved and the quantitative behaviours of CMMI level 4 implemented. The two areas which appear to offer the most scope for further improvement are:

- Move to also measuring the 'Touch Time' when value is added to the work, to supplement elapsed time in days. This may well suggest ways to further improve the process.
- To discover how to move further upstream in the organization to ensure a total systems view is taken so that the correct IT projects are identified in the first place.
REFERENCES


Seddon, J. Freedom from Command & Control, Buckingham, U.K. Vanguard Education, 2005


USING PRODUCT PORTFOLIO MANAGEMENT TO IMPROVE THE
EFFICIENCY AND PREDICTABILITY OF TEAMS

Alan Shalloway

ABSTRACT
Product portfolio management has become an essential discipline for all development
organizations that want to achieve enterprise agility. The repeated process of selecting, sizing, and
prioritizing the work to be done ensures that their development teams are delivering the most
valuable products and enhancements for the business’ clients. This is required for both external
clients in the case of product companies and for internal clients in the case of IT organizations.
However, the subject of this paper is another, possibly even more important, reason: avoiding the
overloading of the organization’s development teams, which greatly lowers their efficiency.

This article will discuss: 1) how overloading teams significantly degrades their performance, 2)
principles that can guide the sizing of the features coming out of your product portfolio
management process, and 3) why product portfolio management is a critical component of a
transition to Agile methods.

An earlier version of this article appeared in the April 2010 edition of the Agile Journal.
USING PRODUCT PORTFOLIO MANAGEMENT TO IMPROVE THE EFFICIENCY AND PREDICTABILITY OF TEAMS

EFFECTIVENESS AND EFFICIENCY
Agile organizations need to be both effective and efficient. Effectiveness is doing the right thing and efficiency is doing it without wasted effort. Being inefficient, even if working on the right thing, is like trying to pile boxes up to climb over a wall when a ladder is available. Efficiency without effectiveness is like quickly climbing the ladder that is placed on the wrong wall. Agile organizations have to look both at selecting the right work for teams to do and then using appropriate methods to get teams to do their development faster. We can think of this as a two step process – selecting and developing, as depicted in Figure 1. Done properly, we call this “fast-flexible-flow” because we can readily select what is needed (effectiveness) and development it quickly (efficiently).

![Diagram of Fast - Flexible - Flow]

Selecting what to work on

Developing It

Fast – Flexible - Flow

FIGURE 7. THE FLOW OF SELECTED PRODUCTS TO THE DEVELOPMENT TEAMS

Also illustrated in Figure 1 is the relationship between these two parts of the development process. Consider what happens when too many things come out of the pipeline on the left, forcing their way into the pipeline on the right. Or what happens if these selected items are very large. In both cases, the development pipeline jams up. To achieve flow across the entire pipeline, the organization has to not only prioritize the proper features, but size them appropriately and only allow a certain number to reach the teams at any one time.

INDUCING WORK
Thanks to the spread of Lean Thinking, software development organizations are talking more and more about the need to “eliminate waste”. It is good to get rid of waste. It may be even better not to create it in the first place!
Unfortunately, software development organizations create new work for themselves without even realizing it. How does this happen? Consider Figure 2, showing two ditch-diggers, one making work for the other.

**Figure 8. Creating Work Due to a Bad Process**

One digger is throwing his dirt into the other man's hole. Now the second man has to shovel that out in addition to his own dirt. Their system of coordination (or lack of it) has created new work - double the work - for the poor guy! And once created, the work must be done. I call this “induced” work, because the way they are working together is causing it to occur.

Is it absurd? Maybe. You would think the second guy would quickly see what was happening, turn around, and change how they are working. This is because in the physical world it is readily apparent what causes induced work. In software development, the act of inducing work is not so apparent and therefore, is more insidious and more difficult to stop. Furthermore, what we often look at to guide our process often increases our induced work under the guise of making us more productive.

To quit shoveling work into each other’s ditches, we have to look at new kinds of measures. Rather than defining efficiency by “productivity”, we have to look at “time”. Consider the following software related example. In an attempt to make themselves more efficient, a testing group decided to batch up the code being tested. The assumption was that this focus would increase their efficiency and, therefore, their productivity. The result, however, was that developers subsequently had to wait days or weeks longer to be told that they made an error. These delays made the developers less efficient and lowered the quality of the code they were delivering to be tested as well. Consider the following conversation I've had numerous times with developers.

**Me:** How many of you would say you spend most of your time fixing bugs?

**Developers:** (Almost all raise their hands).

**Me:** Consider what happens if you write a bug and are immediately told there is an error. How long does that take you to fix?
**Developer:** Not much at all.

**Me:** Now, consider the same error, but this time, you are not told about it for a week. And imagine nothing else has changed. How long would it take now to fix it?

**Developer:** Obviously, much more time. And, if something has changed, if someone else has been working on the code, it will take a *lot* more time! There would be so many changes to have to wade through.

The fix was the same in both cases, but now there is extra work. The delay, from error until detection, caused this extra work. I call this induced work because it was created by the delay.

If you agree with the developers I have talked with, that they spend significant amounts of their time fixing bugs, then surely the amount of induced work we face is significant. That is a lot of dirt!

**THE COMMON PROBLEM: DELAY**

In software development, there are many different types of induced work; however, they all have a common element: delay. The most common delays are the time between when information is needed and when it is used and between when an error is made and when it is discovered. And they are made worse by delays in getting information needed to act.

The primary cause of delay is doing more than one thing at a time. The math is simple. Doing two things at once doesn't take twice as long; it takes twice as much time plus the time to do the newly created (induced) work. Accounting for waiting, detecting, remembering, stopping, and starting, the resulting time can be significantly longer.

But, you say, “We need all of this work done. We can’t afford to do them one at a time!”

I have heard this justification time and again. All of the work items are critical. The development teams need to work on all of them. But what we *need* and what is *reality* are often different. Imagine being a pilot of a two-engine airplane. The co-pilot says, “Captain, number 1 and number 2 engines are out.” You respond, “That can’t be! I need those engines on!” What you need and what reality is are two different things. The same thing is true with our work; what we *need* done and what is *possible* are two different things. To create the greatest chance of getting what we need done, it is critical to avoid creating new work that does not add value.

**FOCUSBNG ON PRODUCTIVITY ACTUALLY WORKS AGAINST US**

Can we just make developers more productive? If they just worked smarter not harder, wouldn’t they be able to get it all done? We have all tried this. It does not work. On the contrary, going for increased productivity directly usually results in *decreased* productivity. Why? Because keeping people productive means keeping them busy, even if that means starting new tasks before completing old ones. This creates multiple things being worked on simultaneously and that means induced work (think back to Figure 1).

Let’s see how this happens. Imagine a development team is working on a project and needs some information. They want to ask questions of an analyst but find they have to wait to get them
answered. This happens because the analyst, wanting to stay productive, has started working with another team. The developers just sigh, go back and decide, well, in order for them to be productive (busy), they'll have to work on something else too. Since they can't continue with their original project they start working on a new feature. Before long, everybody is doing several things at once, each waiting for the other. It gets so bad that people stop trying to talk to each other and instead start using emails to communicate. And so it goes, worse and worse.

It is a horrible situation and all too common. We have all been there. We are incredibly busy but do not achieve real value compared with our efforts. We fall into this trap because the goal of productivity does not provide us insights into achieving it. Poor decisions are made because we take actions that increase local productivity, keeping each person working, but ignore the big picture – getting value delivered to our customers.

**The Software Development Value Stream – A Necessary Component of Increasing True Productivity**

This value stream is the flow of work from its conception to its consumption. Figure 3 illustrates this, showing the roles of the customer, business, management, and development team.

---

**Figure 3: A Conceptual View of the Software Development Value Stream**

The diagram illustrates that we must go to our customers (internal or external) to see what they need. Business sponsors need to select, size, and prioritize those features that are going to be created. By creating the smallest features that are still useful from a customer's perspective and viable for the business to deploy, product managers can provide work assignments that teams can quickly build and deploy. Teams, for their part, must learn how to develop in smaller steps than

---

Note: Features of this type are often called Minimal Marketable or Minimal Viable Features.
they may have been used to. Management’s job is to facilitate how teams interpret the
business/customer need, provide visibility to the business of what is being accomplished and to
improve the organization so that impediments to the value stream are removed. This combination
of business, management, and teams is required for agility at the enterprise level.

**Cycle Time: The Key Metric**

The value stream provides us with a better metric to guide our decisions than productivity levels,
which may actually work against us. One way to lower induced work is to remove the delays that
cause it. Reducing delays will reduce the time it takes for our work to flow from conception to
consumption. We call this the value stream’s cycle time. Shortening our cycle time will necessarily
shorten the delays between the work being done. This makes cycle time the key metric to focus on
to see how efficient our process is. The value stream highlights where induced work is occurring by
making the delays between our work visible. These delays show up as bottlenecks and delays.
Looking at cycle time helps you understand when features are too big or when there are too many
in the pipeline. Properly managing the size and number in your portfolio will have a significant
impact on your cycle time.

**Starting an Agile Transition without Considering Product Portfolio Management**

The common mantra in the Agile world today is to start a pilot project focused on creating team
agility. The intention is to scale this process throughout the entire organization after one or two
teams have learned it. In many organizations, however, the real problem isn’t so much the teams’
process as much as it is too much work being handed to the teams, which keeps them working in a
haphazard and inefficient fashion. Oddly enough, this team-pilot approach often has great initial
success but is not able to scale to the organization. This occurs because the team selected for the
pilot projects is told to work on one project. They achieve much greater productivity, mostly
because they have created the team to be focused on one project, are often co-located, and have all
of the skills they need to do the project on one team. We’ve seen that these factors have just as
much impact on a team’s efficiency as do Agile methods.

Of course, they may not truly understand this and likely will attribute it to their new process. As
they add more agile teams, the rest of the organization actually gets worse and worse. Eventually,
the local improvements of the agile teams are more than balanced out by the increasing burden
placed on the other teams. Some point to this as evidence that agility works since the agile teams
are doing well – even if the organization is not improving. Ironically, the more the teams as a whole
are being overloaded (that is, the more the need for proper portfolio management) the better the
pilots will appear compared to the other teams. This is one of the key reasons agile pilots work but
then agility at the enterprise level fails.

Ken Schwaber, co-creator of Scrum, has said, “I estimate that 75% of those organizations using
Scrum will not succeed in getting the benefits that they hope for from it” (Schwaber). While this
number has been generally agreed to among software consultants, the reason why has not. I was
initially going to say “hotly debated”, but in fact, most Agilists haven’t investigated the cause for this
other than the immediate assumption that the failure occurs due to lack of motivation or discipline
within the organization.
Our experience is that team-focused agile methods don’t provide the insights required to remove the organizational impediments that they face. A more holistic, value stream oriented view geared toward shortening cycle time provides these insights. Scaling often fails because the root cause of the impediments in the value stream are not observed, let alone dealt with. Agility can be achieved across an organization when the entire value stream is considered – not merely for a lucky team or two selected for the pilot project.

**IMPROVED PREDICTABILITY AS WELL**

So where does predictability come in? Predictions are notorious difficult to accurately make (it is easy to make bad predictions). The difficulty is the high degree of variation that exists in software development. Let’s examine this problem.

\[
\text{Time To Completion (TTC)} = \frac{\text{Amount of Work (AOW)}}{\text{Rate of Work Completion (ROWC)}}
\]

AOW is related to the size and complexity of the work.

ROWC is related to the number of people and their rate of completion.

When estimation is typically done, we focus on size and complexity of the work, thinking that this is where the big variation is. While size may vary, complexity within a domain probably doesn’t vary by more than 2-3 times (I’d suggest typically more like 50%). However, rate of completion can vary by a factor of 10! Why? Because if you work on five things at once versus one, you have a factor of five right there. Add induced work and you have a factor of 10 or more. We have all seen this – give too much work to a team and nothing seems to happen – yet we keep trying to do it.

Hence, managing the rate of work being worked on is essential to both increase productivity as well as increase predictability. Kanban/Scrumban gives us the final piece of the holy grail of software – being able to predict when you’ll be done. Those of our clients who have actually limited WIP throughout the value stream by using product portfolio management coupled with Kanban/Scrumban have seen this predictability. They can tell when a team is not functioning properly by seeing that its rate of completion is off compared to other groups in the company. Interestingly enough, when this happens it is typically because lack of visibility into the process. This should not be surprising, given that the core of Kanban has often been stated as limiting WIP with visibility into the process. If either is off, you will have problems.

We've all heard “work smarter not harder” as if that tells you something. The answer is “work without delay” or, as David Anderson likes to say, “Stop starting and start finishing!”
REFERENCES
CONVERTING A SCRUM TEAM TO KANBAN
Mattias Skarin

ABSTRACT
In 2009 I met a team in trouble. They were working huge amounts of overtime and caught in the evil code-code-don’t-ask loop. Their mission was to replace a core business system at a key client. With the deadline two and a half months away, the client putting all future engagements on ice, the challenge became to surface the right problems and apply countermeasures that would have an effect in time.

This paper tells how we used Kanban as a methodical approach to surface and solve problems. Kanban helped us:

- Surface the impediments preventing flow of value
- Bring together line managers, project managers, and developers to overcome them
- Step-by-step shift the focus from coding to high quality deliveries
- Maintaining a discipline of quality, even in times of high pressure
- Bring surrounding parties (such as the client developers) onboard with the changes we made

The delivery was made in time and the client came back to the company with their next project. At the time of go-live, the team was no longer working overtime. During the period, the velocity of the team increased by a factor of 1.9.
CONVERTING A SCRUM TEAM TO KANBAN

INTRODUCTION
The team was in trouble. They were six months into their eight-month project when the client threatened to pull the plug. This meant saying good bye to a key client. What had initially run well – a pilot delivered on time creating a happy client – had turned into a growing pain of overtime, rework, and technical debt.

Scope adjustments had been made along the way, but what was now due was the bare minimum the client needed to run his business. There was no fat left to cut out. The system to be developed was to run our client’s core business. And they were very nervous to miss the goal.

On the positive side, (come on, there is always a positive side :) the team had very committed team members, a weekly dialog with the customer (not pleasant, but still), a positive project manager/product owner, and strong backing from management.

When I first met the team, I asked what the biggest problems were. The answers were:

As a Project Manager:

- Deliver something with quality
- Team overstressed
- Estimations totally wrong, up to five times off for new stuff

As a Team:

- Switching too much between tasks
- Doing too many things at the same time

As a Client:

- The team is not up to this task
The answers varied greatly based on whom you asked. I concluded that there was no help here in figuring out what to fix first.

**TEAM SETUP**

In reality there were two teams. One based in Great Britain (client side) and one in France (our side).

The main bulk of development was made in the French team (our side) with some extensions developed by the client. Our product owner and his Project Manager met once a week to revise prioritization and review demos.

The French team had been using Scrum for a couple of months and the Project Manager was a certified Scrum master. It was not hard to find problem indicators. The team worked massive amounts of overtime, sometimes coding to 1.00am in the morning. The sprint burndowns displayed a worrying pattern:
When asked about how it worked out for them, the developers’ comment was, “We are not sure Scrum is helping”.

**HOW KANBAN GOT INTRODUCED**
The team really got introduced to a Kanban board by their CEO. He noticed what was going on at another team I was helping out. They had been running Kanban for a couple of weeks. He asked if he could introduce a similar board to this team and I said “sure”. The next week their first Kanban board was up and running.

![Figure 1](image)

**Figure 1: The first Kanban board team used. Work in progress was applied to the Dev, Merge, CI and Product owner (PO) test columns. A fast lane existed for urgent items (see top) with the limitation of only allowing one such item at a time.**

Nothing else in the existing work procedures was really changed. The only news was the introduction of a work in process limit and increased visibility in the value stream.

**AFTER TWO WEEKS OF KANBAN**
The teams continued their sprints for two weeks using this Kanban board. Below is the sprint burndown after two weeks.
It hinted to me that the team really could produce working software if conditions were right. At about this time I got the opportunity to work full time with the team.

**Deciding What to Fix First**

Two weeks into the process, the Kanban board displayed some interesting information.

- Stories got “stuck” in the test phase
- Stories would enter the sprint end up half done, then re-enter in later sprints

It turned out that there were several stories proving too hard to complete within a sprint. Unfortunately they were also mandatory to complete the project, which meant they could leave the sprint to re-enter later, since they did not leave the project scope. This had a demoralizing effect on the team, watching the same problems resurface over and over again, always unsatisfactorily solved. The team, often under heavy stress, reverted to the first possible solution in order to make their sprints.

I also noted that the team and Project Manager were tracking two different things. While the Project Manager was tracing completion of project tasks, the team was tracking sprints. No common view existed on the project progress.

<table>
<thead>
<tr>
<th>Project</th>
<th>Sprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project manager with task completion focus</td>
<td>Team with sprint focus</td>
</tr>
</tbody>
</table>

| 60% | 30% |

**Finding Time to Do Necessary Improvements**
While there were several things we could try out, we had a more urgent problem first: finding time to do things. The developers’ schedules were filled to late evening trying to get the software done. So, we needed to find slack.

**Simply Estimation**

The simplification of estimates started as a way to solve a more urgent problem. I needed some time to talk with the team but they were busy that day. They had promised delivery of story estimates to the client and had the day blocked off for it. I proposed to the Project Manager: “If we can finish the estimation faster than expected, can I get free time with the team?” She was happy to try something different so she said, “sure”. I asked them to have lunch with me and bring the stories along. During the meal, the team passed around the stories and completed the estimation using a simple T-shirt sizing scheme. By the time of coffee, all stories had been estimated and I had my free time with the team.

The estimation scheme we used was very simple:

- Small: 1 day or less
- Medium: 1 week or less
- Large: Anything bigger

**Abolishing Sprints**

We did not see the value derived from our sprints. For example, they continuously seemed to be broken by outside events. So, we decided to try stop sprints, instead focusing on achieving high quality and a continuous flow.

One of the things we kept was a weekly release cadence. If a piece of work was completed with high quality, it would be passed into the release. If not, it would have to wait until next week. This enabled the Project Manager to mitigate stories “close to completion” and shift them to the next week’s release instead of moving the release for one more day.

The value we got from these actions was less thrashing; stories were now allowed to stay in work until finished.

**Addressing Quality and Development Unbalance**

It was not rare that the Kanban board would display the following pattern:
Stories would aggregate in the test column. We were quick to code but not as fast in testing and correcting. It was our Project Manager doing the testing, so in reality we had a part time tester.

**Building in Quality**
How to fix the unbalance? We started by introducing TDD. The intention was to shift parts of the quality work load on to the developers and introduce fewer bugs.

We ran four TDD sessions to teach the team test driven development on the fly. Because of the tight schedule, we had no other choice than to run the training off hours. In agreement with the team, we conducted a set of morning sessions when the team agreed to come in early four days to do TDD workshops.

Right after the TDD workshops, the team addressed a series of refactorings in their code. It turned out that the team had known about them for a long time, but now felt they had the necessary tools to deal with them.

**How We Dealt with Test Frameworks and Technical Hurdles**
Not all parts of the code could be developed using TDD. For example, there was GUI code not supporting TDD. Although we had ideas of how to fix this, implementing the necessary infrastructure was a time consuming technical hurdle and not something fixed in a day. So we made a compromise:

- Design for test were we can
- Do manual testing were we could not write automated test cases
**Figure 3:** The second step of the Kanban board. Test design was introduced as before coding. Team also changed the last steps to build and package to better support release coordination with the client's team.

*How We Got to a Releasable State*

A new problem quickly surfaced:
While the quality seemed to improve, our releases were frequently interrupted by late commits. To mitigate this we decided to change the branch policies.

We pushed to get this implemented in both teams; when in place it helped us maintain a releasable state and mitigate problems while still small.

**Small Signs of Improvement**

Even if the pressure still was intense, small signs of improvement surfaced. Comments like, “Why do we have things like this in our code!” could be heard and the team members started to take individual initiatives in refactoring and fixing quality problems.

About this time the Project Manager returned from a client meeting and told me, “You know what? Even if they are still stressed out, they now trust me when I say we are going to deliver something”.

Small indications that we were on the right track.

**How We Got Continuous Improvement Going**

So far we had mostly been on the defensive, fighting off problems as they surfaced. Also, they had support from an outside coach (me). However, we needed to strengthen the team’s capabilities to act proactively on their own.

About this time the team started to take control of the board, adding policies and refining work states.
**Figure 4:** Updates to the Kanban board. A triggering mechanism tells when partial or full builds are needed. Testing is separated into unit testing and functional testing to help focus manual testing.

I trained the team in how to discover problems early using the board, in doing root cause analysis, and in team problem solving. The team started to run weekly continuous improvement sessions together with their Project Manager. Each session focused on one simple thing: fix one problem every week.

Root cause analysis helped us move from individual problem perception to unified understanding of cause and effect.
It was fascinating to see how issues started to surface. An example: the problem in the picture above “Editor delivered with low usability” (UI component missing prominent feature). In this case the Project Manager had known about the problem but considered it useless to surface since there was no obvious way to fix it. Once surfaced, the problem was addressed immediately by starting to train the team on understanding the code base.

**How We Addressed Cross Team Problems**

A problem that surfaced during the continuous improvement sessions was “how do we get a working cooperation with and transfer know-how to the client’s team?” To address this, lead developers took initiative to arrange a weekly telephone conference aimed at addressing common problems. We also started a developer exchange program, where developers from the client’s team would spend time with us and we would work with them to learn their context.
During the last couple of weeks the managers and the team worked closely together to fix problems that might prevent the release. For example, our CEO ensured availability of a senior developer to speed up fixes in core modules as we got closer to the release.

Yes, the team made the deadline. While this seemed like an unreachable goal two months before, I remember passing the finish line more like every day work than something extraordinary. The official acknowledgement arrived one week after the release when the client gave a green light for the next project.

We all know it is not impossible to meet a tough deadline. Really - all you need to do is work massive amounts of overtime. So - what made me really happy was a comment from the Project Manager right after the release:

“You know what? The team no longer works overtime.”
A Look Back
I find it hard to identify “one single thing” that made this work. It was more the combinations of many small things accelerating each other. Where Kanban really helped was by showing us where the problems were (and weren’t). Fixing the problems was up to us.

The good thing was that this information was “cheap”. All we needed to do was to put a Kanban board on the wall and start to use it. No other changes were required. We kept our Jira tracking system, our project structure, and our way of work. But, as soon as a problem was identified, we then turned our full attention to fixing it.

So were there other solutions?
Wise people I have met have told me that more than one solution always exists to any problem.

“So it looks like you were thrashing, why not work in longer sprints?”
That is also an option. Our problem was we had not enough trust selling this to the client. I am not sure a message like “we are going to work in longer sprints and keep away while we do” would have produced a positive reaction.

“And what about your definition of done?”
It is true we were working in a “scrum butt” way. But, we were not the only party in the project. We did not control the client’s deployment procedures for example. Changing the definition of done required having the client with us, accepting the consequences. In reality we did something similar by providing a longer visibility downstream into the value chain.

Feedback From a Team Member

“The hardest part is to train yourself to embrace a new mentality. To realize that if a task is coded, this does not mean the task is completed.

Kanban ‘forced’ us to think about quality. When you have a column called ‘function test’ on the board it’s a little bit hard to ignore it :) It forced us to stop the coding, coding, coding chain.

I think, the best lesson we learned was always think about quality”.

What About Data?
**Velocity / Throughput**
The average throughput in May was 7.25 story points per week and in September 13.50 sp/week (red line).

**Velocity / Man-day**
If we look at the throughput per man-day, the May average was 0.64 and in September 1.04 (green line - vertical axis in graph scaled in percentage).

**WHERE ARE WE TODAY?**
I still work with the team today. The client is still with the team.

The team has pushed on and implemented most of the testing frameworks they wanted to put in. The reason I mention this is to show that you can overcome technical hurdles and project management sells. In the beginning we could not envision the moment we would get the necessary time to put these fixes into place - but now they are there.

The challenges today are much related to transferring know-how to the client’s IT organization outside its development team.

**KANBAN GOING WILD!**
Late into the project a girl from H&R approached me and asked if I had any ideas that could help decrease the stress she experienced. I was skeptical at first but figured out that frequent reprioritization of things she had started to work on was one of the culprits. We discussed some options, and then I drew a simple Kanban on the glass wall of her cubicle. I asked her to tell her instructed stakeholders to put their requests in priority in the “in queue”, but to leave any started work alone.

One week later I passed by her office and noted there were Kanban systems all over the walls. A Kanban system had been started by all the staff around her – Economy, Marketing, and IT.
FIGURE 7: KANBAN THREATENS TO TAKE OVER THE WORLD.

I hope there is something here to learn; I learned a lot.
FEATURE BITS: ENABLING FLOW WITHIN AND ACROSS TEAMS

Erik C. Sowa & Robert Y. Loh

ABSTRACT

“Feature Bits” is a technique for using latent code in SaaS applications. Latent code allows us to separate the act of rolling code to production from the act of releasing features to customers. The flow of product development work is enabled by eliminating many of the collision points that would otherwise require intensive coordination and big-bang deployments. We review the business context for this technique, the solution design, and the coding patterns we use to implement it. We review some usage observations and statistics based on 1.5 years of experience.

Keywords: Latent code, flow, continuous delivery.
**Feature Bits: Enabling Flow Within and Across Teams**

**Business Context**
Lyris HQ is an online marketing solution for small and medium businesses. Our product is based on several previous products or “point applications” developed by different companies with different technology stacks and different cultures. At any time we have 80 people in 8-10 teams building product that is rolled to our production servers every three weeks.

When we started this work, features that spanned development teams and features within a single team that spanned multiple code rolls presented us with frequent coordination problems that slowed us down to unacceptable rates of change.

For example, consider a team replacing a complex report over multiple iterations. In the classic approach, the new version of the report would be kept in a development branch until it was complete, then merged into the production branch for release to all customers simultaneously. Making the decision to release it is all-or-nothing, and any one of several stakeholders can ask to delay the release until his or her perceived needs are addressed.

For another example, consider several teams implementing an architectural change to the way their parts of the application work with common application services. In the classic approach, releasing this change to production is again all-or-nothing; if any one of the teams is not ready to release, no team can release, and other changes that might be ready to roll can get backed up behind the logjam.

In both of these examples, the classic approach uses branching in the code repository to isolate changes that are not ready to be deployed. Large merges and the associated integration testing was another coordination problem that slowed down the delivery of value.

In August 2008, we began implementing “Feature Bits” to mitigate these problems.

**Latent Code Technique: Feature Bits**
Developing and operating a Software-as-a-Service (SaaS) application presents special opportunities and challenges. We can see and measure what our customers are doing at any time. We can update the application, whether to add features or fix problems, at any time. Unlike shrink-wrapped software, there is no physical driver to hold changes until the batch is big enough to justify the overhead of a physical release. Companies in this space are doing very good work towards “continuous delivery” (Humble & Farley, 2010); that work relies on strong teamwork between development and operations (Edwards, 2010a; Edwards, 2010b) and has been featured in conferences recently (Allspaw & Hammond, 2009). Some of the busiest SaaS sites are updated many times during each business day without adversely affecting user experience; in those companies deployment or “rolling code to production” is by necessity simple and stress free. We now have the technical capability to deploy changes faster than our customers can absorb them. We also have the need to test major changes at production scale, and with production-quality data, before exposing them to all customers. It seems that we need a complementary technique to balance the continuous delivery capability.
In this paper, we assert that the capability to deploy latent code is a complementary technique. Decoupling the act of rolling code to production from the act of releasing features to customers addresses these needs. Based on a suggestion by Sean G. Ryan, who used a similar technique at Digital River and ExactTarget (personal communication, 2008), and with technical leadership provided by Robert Loh, we implemented “Feature Bits” to effect this decoupling. The overarching goal was to allow incomplete features, and complete features which customers are not ready to consume, to stay in the code base even while we continued to perform our frequent rolls to production, and to enable testing that code in production before the features are released to customers. We have achieved that goal and realized several other benefits along the way. We call our solution “feature bits” because each “feature” has a “bit” that controls its visibility at various levels of granularity. Other companies have come up with similar techniques (Hammond, 2010; Popescu & King, 2010); a common theme of these techniques is to move the “branching” decision from development time to run time.

**Usage Observations and Statistics**

Feature bits started in August 2008 as an innovation coming out of engineering. We used it to decouple the workstreams of multiple teams working on the new Flex-based user interface, and the coordination points were very coarse, e.g. replacing major navigation links to the classic application with links to similar or improved functionality in the Flex application. For several months, we operated with only 10-20 feature bits administered by engineering.

Meanwhile, as our product management team learned that they could control when a feature was released to customers, feature bits worked their way into release planning. One consequence was that product owners took over administration. Another consequence was that the number of active feature bits proliferated. By December 2009 we had over 80 active feature bits, and we were not following up on removing feature bits that were no longer useful. So we raised the priority of removing feature bits and ended up with only 40 active bits at the end of the month. We also made some small changes that made obsolete feature bits more visible. At the end of March 2010 we had 48 active feature bits in production.

The biggest innovation from product management in 2010 was to use feature bits to batch changes into "UI releases". One intent of this was to give downstream teams, e.g. support and training, time to absorb the changes and prepare for the impact on their work with customers. Another was to prepare marketing material to help our customers anticipate the changes and value them appropriately.

Let’s take a closer look at how we made feature bits work.

**Implementation**

**Technical Context**

In the back end of our system, we have a federation of “point services” which defer to a “core service” for authentication. This core service also acts as the SSO portal, and is hosted on the domain directly associated with our product; our user experience starts here. Additionally, all point services are registered in the core service.
**Figure 1: Overview of our Ecosystem**

There is an Org-Person hierarchy (Persons have accounts in a given Org), as well as Org-SubOrg hierarchy (Orgs may have Sub-orgs, with some associated entitlement and cascade rules). The result is that a given Person may have multiple accounts, one in each Org to which they have explicitly been granted rights.

**Figure 2: Org / Person ERD**
The primary front end of the system is a Flex application. The first UI layer of the system is a thin loader app, launched from the core service. The first thing this app does is request bootstrap information for the current user session from the core service. This bootstrap info includes the list of point services for which this user has rights.

The Flex panels in the system are coupled to back-end services which provide the actual functionality. The loader app understands enough about a user’s panel request to be able to instantiate panels for a given point service. Panels are typically bundled together in a flex module SWF file, which is served from the point service whose API those panels consume. The loader app finds and loads the associated module SWF, which exposes a factory interface for creating specific panels. During the lifecycle of this process, the point service’s “module factory SWF” is itself given a chance to bootstrap with the same package of data the loader app initialized from.

Requirements

- Three levels of granularity for each feature: global, user, and org
  - Global is the rule for everyone without an explicit bit
  - Org level overrides specify the state of a feature for all users within an org. Note: Org level overrides DO NOT cascade up/down the org hierarchy, i.e. it doesn’t matter what bits are on for an org’s parent
  - User level overrides specify the state of a feature for a specific user account
- The absence of a given feature, or any bits for that feature, means that feature is “off”
- Ability to check a given feature from the GUI tier
- Ability to check a given feature from the services tier
- Ensure a consistent list of features across our shared environments, as code is promoted from one to the next (dev to qa to production)
- Allow each “service” to own the list of features relevant to its stack
- Simple process to get a new feature into the system
- Given the diversity in our point services, keep the implementation and checks simple
- Administration capabilities:
  - Review a list of all features in the system
  - Turn a feature on/off globally
  - Turn a feature on/off for a specific org or set of orgs
  - Turn a feature on/off for a specific user account in an org
  - Review the bits in place for a given feature

Core Infrastructure

Our core service was the natural home for the API and related data. It allowed us to piggyback on the bootstrap mechanism to drive the session-based features available to the user in the Flex tier. Additionally, it was a natural home for the API to allow any of our point services to perform relevant checks for feature bits.

Table structure

A Feature has a name and a description. A Feature Bit is associated with a Feature, and optionally
associated with an Org, or a User within an Org. A row in the Feature Bits table implies an override to the default rule of “off” (which may simply be an explicit bit specifying “off”).

![Feature Bits ERD Diagram]

**Figure 3: Feature Bits ERD**

**Querying**

There is one main query that drives the pattern. It is complex but it nicely isolates the implementation of the required logic. Also, although this query is relatively complicated with its nested sub queries, the Feature Bits table should ideally never expand beyond more than a few hundred rows, so performance impact is relatively minimal. We implement the following using named JPA queries, since our stack uses hibernate and jpa for basic data persistence.

Finding all features for my account (provide :user and :org). This query is used to generate the feature list for the bootstrap data, as well as our service API call.
SELECT f FROM Feature f WHERE
  ( EXISTS (SELECT fbg FROM FeatureBit fbg WHERE
    fbg.feature=f AND fbg.org=null AND fbg.user=null AND
    fbg.isEnabled=true)
    AND NOT EXISTS (SELECT fbgo FROM FeatureBit fbgo WHERE
      fbgo.feature=f AND fbgo.org=:org AND fbgo.user=null AND
      fbgo.isEnabled=false)
    AND NOT EXISTS (SELECT fbgu FROM FeatureBit fbgu WHERE
      fbgu.feature=f AND fbgu.org=:org AND fbgu.user=:user AND
      fbgu.isEnabled=false)
  ) OR
  ( EXISTS (SELECT fbo FROM FeatureBit fbo WHERE
    fbo.feature=f AND fbo.org=:org AND fbo.user=null AND
    fbo.isEnabled=true)
    AND NOT EXISTS (SELECT fbou FROM FeatureBit fbou WHERE
      fbou.feature=f AND fbou.org=:org AND fbou.user=:user AND
      fbou.isEnabled=false)
  ) OR
  ( EXISTS (SELECT fbu FROM FeatureBit fbu WHERE
    fbu.feature=f AND fbu.org=:org AND fbu.user=:user AND
    fbu.isEnabled=true)
  )

Breakdown of this query: find the features where: a) the global bit is on, and there is no org override for the bit to be off, and there is no account override for the bit to be off, or b) my org has override for the bit to be on, and there is no account override for the bit to be off, or c) my account has an override for the bit to be on.

**Flex Tier**

For checking feature bits from the Flex tier, we decided to drive “the list of active features” at a session level, allowing us to piggyback on top of the bootstrap mechanism. We ship the list of features available to a user in the bootstrap data. Then we added API to our config object as the mechanism by which to check a given feature. Since this code is in our company’s common Flex libraries, it allowed all our front-end development teams to painlessly implement feature bit protections.

The check is simple; embedded in the bootstrap XML is a section similar to the following:

```xml
<features>
  <feature name="hq.remote_panel_load" />
  <feature name="el.enable_asset_library" />
  ...etc...
</features>
```

So, a simplified xpath implementation of the “is a feature on?” check looks like this:

```java
public function isFeatureOn(featureName:String):Boolean {
  var nodes:XMLList = xml.features.feature.(@name==featureName);
```
Services Tier
The core service provides a single RESTful API, which is designed to be used in conjunction with caching mechanisms at the point service. Put simply, you can ask for the list of features that are “on” for either an org or a specific account.

${core-url}/accounts/featureBits?userUid=&orgUid=&

Calling this API will return an XML payload similar to the one described above. In our case, we have a bit of standard API wrapping that comes along with it.

There is a reason we don’t have an API like:

${core-url}/accounts/checkFeatureBitForAccount
to check a specific feature or set of features against an account. Put simply, we want to discourage many micro checks, and instead encourage the “grab the bigger list, cache it, and look into it yourself for granular checks” approach. The query to check for the list of active features is already pretty costly; specifically checking for an actual feature only adds a layer of complexity and cost. Additionally, a service is much more likely to call an API like this in high volume if this approach is not encouraged: Imagine an asynchronous batch processing job that wants to “skip the new data processing steps” for those customers who aren’t set to see that new data. Instead of having point services making a call against each feature they’re interested in, we’d rather they ask for and cache the list of features for a given account, and refresh on a reasonable interval (e.g. job runs once per day, reasonable to ask once at start of job and have each check within the process [if there are multiple] reference the same data). Once you start to have multiple bits across different jobs, the benefits behind batching and caching the results can increase exponentially.

Discovery
Context
There were several design constraints we associated with discovery of new features. The following list summarizes our concerns:

- Correctness: Manual data entry is prone to typo or cut-and-paste errors; consequences include your feature never being enabled.
- Consistency: We have a series of target environments; builds of the application transition from development to QA and finally to production. We needed to be able to “bootstrap” or otherwise “sync” the features list in a consistent way across each of these environments.
- Decoupling: We have multiple code bases with isolated deployment vehicles. If the “list of features” were centralized in a single app (say our core app), then all other point services would be dependent on a roll of that code to get a new feature into the list. We needed the features relevant to a particular point service to be decoupled from any other code base.
- Security: The production network is isolated from any development or QA environments.
Solution

Each point service publishes a list of features. This takes the form of a simple XML feed served from a known relative endpoint (per point service). Because new features appear in the system on roll boundaries, this couples the registry of features recognized by a point service with the code that the related feature bits will control.

Example XML:

```xml
<features>
  <feature name="EL.enable_new_overview" description="Enables the new overview panel" />
</features>
```

This file (or service) is maintained in the point service's code base. When a developer needs to add a new feature, he or she simply adds an entry to this file and updates it in source control. For our needs, this covers the majority of use cases.

If a particular feature crosses point services, the teams can coordinate to decide who will "own" the feature entry. Roll order should depend on release plans and technical dependencies; ideally it won’t matter, since a feature not registered in the system is equivalent to a feature being “off”. However, there may be reasons why a particular service might want to own the feature: They are creating a new API to be consumed by other services, or their engine needs to pre-process a large chunk of data, and wants to roll with a lead time in order to get this processing done, or simply because they are scheduled to roll in a particular order.

The core app provides an admin feature to “sync features” across the system. Triggering this function will cause the core service to spawn a background thread to query each registered point service for their features XML, and consolidate a list of features to be added to the system.

Administration

Feature administration is consolidated in the core service. There are several reasons for this:

- The core service is the authority for all authentication,
- The core service owns the bootstrap process for our flex app,
- The core service owns the other system level admin functions.

The interface for Feature Bit administration provides two basic views, one of which has three filters:

- Global, Org, and Account showing the list of all features, and their relevant bit status.
- Feature, showing all bits under a feature and their implied override state.
**Figure 4:** Account level permissions across all features. Global and Org are basically different filters on the data set.

**Figure 5:** Feature level view. Drill down on all bits in the system for a given feature.
As described above in “discovery”, the list of features is acquired by querying all attached services for their list of features. This provides a roll-agnostic mechanism to sync the features list with all of our original design constraints.

There are several use cases for the interface:

- Post deployment, and likely prior to smoke testing, Ops or QA will sync features and turn on new features for some internal orgs to enable testing.
- Product owner wants to turn a feature on for everyone at some date decoupled from the code roll date.
- Product owner wants to find a feature, and turn it on for a specific customer or set of customers.
- A new feature is on for everyone, but breaks for a specific customer, perhaps due to the state of that customer's data supporting the feature. Advanced support can turn the feature off temporarily as a workaround while a fix is effected.

**STRATEGIES FOR PROTECTING CODE**

**Generalized and Abstracted**
A nice consolidated way to control the release of a new feature is by focusing on the launch point(s) for the new feature. If a new feature is only available from the top-level navigation system, one can devise a system that eliminates the need for developers to have feature-bit related code anywhere in the repository at all.

In our case, we use a dynamically instantiated array, with an object per top-level category (and an array of link items underneath) to drive the definition of our navigation bar. Parameterization for links include the point service responsible for this panel request, and the target panel; we extended the definition to include optional information on associated featurebits. Thus, we can easily control links in the navigation bar based on the state of a feature bit. Early on, our focus was on transitioning customers from links to the legacy HTML gui to our new flex equivalents, so this solution involved “falling back” on an (optional) link to an HTML page when the feature bit was off, and hiding the link when no URL is provided.

Example JSON of a module-link definition with no URL fallback:

```json
{ app: "EmailLabs", type: "panel-featurebit", featurebit: "el.enable_sms_messages", panelClass: "smssubscribers", }
```

**Basic Checks**
The basic pattern for protecting code with a feature bit is simple. Within the Flex app, centralized library functions allow you to check status of a given feature for the user in the current session as follows:

```java
if( Registry.instance().configData.isFeatureOn( featureName ) ) {
  // new implementation ...
} else {
```

However, when expressed as such simple code, the tendency will be to abuse the simplicity. Thus, we strongly encourage their use via patterns.

**Patterns**

Ideally, developers implement feature-bit controlled logic in an open-closed fashion. The worst-case scenario has the developer sprinkling checks as above in numerous places throughout the code for a given feature: Doing so will complicate maintenance of, and eventually removing, the control mechanism (and retiring the old code, if relevant).

If the target system is built using object-oriented design patterns, there are many opportunities for creative and clean ways to iterate on new feature development with the protection of feature bits, while guaranteeing the integrity of the existing code and contracts.

For example:

**Use with Strategy pattern to redefine behavior:**

```java
if( Registry.instance().configData.isFeatureOn("service.new_analyzer") ) {
    service.analyzer = new Analyzer2();
} else {
    service.analyzer = new Analyzer();
}
```

**Use with Factory pattern to decide which version of a panel to instantiate:**

```java
public function createMainDisplay():DisplayObject {
    if(Registry.instance().configData.isFeatureOn("service.panel_redesign") ) {
        return new panel2(); // which extends panel
    } else {
        return new panel(); // which extends DisplayObject
    }
}
```

**Use with Chain of Responsibility pattern to enable new (API hook-points):**

```java
if( Registry.instance().configData.isFeatureOn("service.enable_trickle_reporting") ) {
    userActionLogger = userActionLogger.setNext( new TrickleReportNotifier( .. ) );
}
```

**LIFECYCLE OVERVIEW**

**Development**

- A story, or some other set of new behaviors, is identified and designated to be wrapped
A name is picked, and recorded in local point service's “features” registry
Code is wrapped with a check for the feature bit
The functionality is tested based on intended state(s) of the feature bit between now and the next planned roll
Code is released. At this point, all of the code protected by new feature bits is disabled.

Administration
• Features are sync'ed in the core service
• Bits for the feature are turned on at appropriate granular levels
• The feature is eventually turned on globally.

Maintenance
• When bugs are fixed, both forks of code must be considered and maintained appropriately
• Code related to the feature bit is retired
• The feature, and all associated bits, are removed from the production environment.

DEVELOPMENT
Assess the need for a feature bit early; protecting code with a feature bit exerts design pressures on the solution. Also, the need to accommodate both sets of code being live in production at the same time may impact design / -abilities (test, scale, use-, etc.)

Apply some simple rules behind the naming, so as to avoid the possibility of collision. Since we have multiple domains in our product, we used "[domain-acronym].” as the required leading text to any feature-bit name to avoid any cross-team naming collisions. A similar team or service identifier would be the simplest marker, or you may choose to use a named vertical slice of your stack, or component names; any top level concept that everyone can agree on.

QA needs to understand the “feature-bit plan” for a given feature, both at the time of roll, and if there is a plan to twiddle feature bits in between rolls. If the intention is to turn on a feature some amount of time after code is released to production, QA will need to run acceptance and smoke tests to exercise both areas of code.

Once the feature has been released to production, all new “protected features” will be disabled or off by default.

ADMINISTRATION
After each roll, an administrator must instruct the core service to sync its list of features. Often, there is a period of time where only a few select users and/or organizations will have the new feature enabled for them. Eventually, the feature will be turned on globally for all relevant parties, but for some time only your “beta users” or “test orgs” will be exposed to the new feature. In some cases (e.g. for the purpose of “decoupled rolls”), you may proceed directly to turning on the switch globally.

MAINTENANCE
Sometimes you can eliminate the need to work on a bug (e.g. the new feature is to be turned on globally soon and addresses the problem in a different way).

Mostly, though, you will have to “fix a bug” in both areas of the code, or otherwise effect the fix in a way which both sides can consume in a reasonable way.

Plan for feature-bit retirement by creating backlog items to remove the feature-bit, including details about any sins against coding that were committed at the time, and how to handle old code.

When retiring the related code make sure appropriate measures are taken for “old paths” (e.g. deprecation, prune from source control, etc.).

Remove the features from your environments a roll or three after the work to retire the feature bits from the code. This ensures that the data is available in case of the emergency need for a roll-back, and that the list of “protected features” in production remains manageable.

**LESSONS LEARNED**

**Design pressure is good**
Feature bits work best when the need to use the technique is identified early and it is applied deliberately. As with test-driven development, this technique does apply pressure to the design. Fewer places to wrap code for a given feature is better, and that pressure leads to modular designs with low coupling and strong cohesion. The feature bit itself is temporary by design; some transient ugliness in the code is an acceptable price to pay as long as it is cleaned up when the feature bit is removed. However, the design of the feature itself survives long after the feature bit is gone.

**Manage the Lifecycle**
Feature bits should have a short lifetime and be retired aggressively. When retiring the related code, ensure that appropriate measures are taken for “old paths” (e.g. deprecation, prune from source control, etc.) Failing to eliminate a feature bit after the feature is completely rolled out leaves you with unnecessary multiple execution paths to manage and test. The combinatorics of multiple feature bits add up quickly to wasted effort.

**Code Must be Production Quality**
Feature bits should not be used to hide untested code. Neither the technical definition of done nor the severity of bugs reported against the work should be softened because, e.g. “the feature bit is not scheduled to be enabled in production yet”. Code behind feature bits is subject to the same quality requirements as any code. The point of testing in production is to work at scale and with real customer data, not to replace the testing that developers can perform while developing the feature.

**Decoupling of Rolls Drives the Default State**
The default state of a feature bit should correspond to the application behavior before the change, especially when the definition of the feature bit and its application are in parts of the application
that roll separately. A mistake here can inadvertently couple those rolls. Don’t have the feature bit “enable” a pre-existing feature being removed; instead, “disable” it.

**Naming Convention Matters**
A naming convention for feature bits is extremely useful in practice. Feature bits are not just for developers – product owners, support staff, and operations staff all need to understand the configuration of the application and good names facilitate effective communication across roles.

**Do Not Overload**
Although feature bits is a simple construct, resist the temptation to overload it for other purposes. For example, the same coding technique can be used to manage application entitlements. But because those are permanent rather than transient, they are administered very differently.

**Manage Customer-Facing Releases**
Should your product team have the desire to batch up several changes into one “UI release” to customers, you may find yourself in the position of managing several feature bits at once. This is probably better than insisting that multiple features be coordinated with a single bit. Finer-grained feature bits help developers do their job of testing their part of the work but do not give visibility into the coarse-grained UI release. As with individual feature bits, identify the need to manage a collection of bits early and be explicit about the expected settings of those bits to inform integration and acceptance testing at the release level. Grouping feature bits into a higher-level construct could be a useful enhancement to the technique but try that coordination manually first.

**Limit the Overhead**
There is an associated tax of development and testing time that comes with using feature bits. If the intention is to turn on a set of features some amount of time after code is released to production, you will need to run acceptance and smoke tests to exercise both states of the application. Over time this can grow more complicated in several ways. For one, multiple feature sets may have different release cycles, necessitating the need to test in all intended states of the product; e.g. as released with no bits on, with bits for feature set A on, then with bits for both feature sets B and C, co-existing with A,... etc. For another, certain features may involve “inverse dependencies” - e.g. the feature refactor hasn’t been wired to work with the new on-screen validation system, so if you have the feature bit for one on then you have to have the other off. Ideally, your product team will coordinate to minimize the complexity of this matrix of the features that are still in the development phase, or otherwise still being quarantined.

**Beta and split testing**
Given our account-centric definition of feature bits, the question of beta-testing features with real customers naturally arises. We have yet to try this. The need and desire to do this would have to be identified early enough to identify and recruit appropriate customers. Similar considerations apply to A-B or split testing, although it is the measurement rather than recruitment that needs to be thought out in advance. Both of these uses put pressure on the transient nature of feature bits.
REFERENCES


FEEDING THE AGILE BEAST:
IMPROVING ECONOMIC OUTCOMES THROUGH EFFECTIVE
ALIGNMENT AND PRIORITIZATION

Dennis E. Stevens & Dean C. Stevens

ABSTRACT

Agile methods of software development are helping teams develop software faster and better than ever before. But even companies that are effectively applying agile techniques are not delivering value faster. Consequently, businesses are not realizing the dramatic improvement in time to value that is required to justify their investments into improve tools, methods, and infrastructure. Not only must organizations get better at developing software, they must get better at scoping, prioritizing, and communicating product vision to gain benefit from their investment.

This paper will explain how product firms can make themselves more agile in their analysis and prioritization approach based on “business capabilities” to identify where to focus their energies and resources. Following this strategy, leading organizations right now are realizing significant improvements time to value.
FEEDING THE AGILE BEAST:
IMPROVING ECONOMIC OUTCOMES THROUGH EFFECTIVE ALIGNMENT AND PRIORITIZATION

**AGILE BETTER, FASTER, CHEAPER DEVELOPMENT**
Agile software development is helping teams deliver software better, faster, and cheaper. Emerging about 15 years ago, agile became a recognized meme in 2001 when a group of people practicing improved methods of software development got together and formalized the agile Manifesto. The agile Manifesto espoused a set of values and principles that this group had found to be desirable in helping teams achieve high levels of productivity.

Agile arose as a set of lightweight methodologies in response to pressures associated with demands for faster delivery of technology. Over the last 15 years a perfect storm of factors has combined to bring these emerging methodologies mainstream. The rise of the internet and mobile networks has resulted in ubiquitous networks (wired and wireless) being available to deploy technology to end users in every setting. Along with the rise of ubiquitous networks, massive inexpensive computing power is now available. New software and infrastructure delivery methods are becoming common place in “the cloud”. Combining these factors with new development, automation, and testing tools makes frequent delivery possible at a very low cost.

We can show empirically that teams that follow a disciplined approach to agile development are able to deliver software better, faster, and cheaper. This arises from a combination of outcomes that manifest in agile development, primarily fit with customer desire, continuous flow of value, transparency, and ongoing learning, and continuous improvement that arises from what Alistair Cockburn has called the collaborative game.

**AGILE DOESN’T EXPLICITLY ADDRESS BUSINESS VALUE**
We expect dramatic results. But all too often, the business does not realize the expected benefit. A big reason is a lack of focus on “business value”. Delivering product with low business value, even if we do it well and fast, will always be a disappointment. While this may be obvious, agile tools don’t help us get to business value as well as we hope. Or we trust that someone else is thinking about business value. We place a great deal of trust that the product owner is a superstar who has a clear understanding of how the product will deliver business value. It can be challenging to establish a shared understanding of business value and performance to ensure work is sufficiently scoped and prioritized so that development efforts are optimally focused on value.

Business value is created by building product that will provide the highest economic return (over time) to the business. If we focus on building things with agile methods, we are controlling costs. If we build things that customers demand with agile methods, we deliver business value. Agile without a focus on business value is cost control. Moving faster is not sufficient – we have to build the right things. Let’s consider a few agile themes that we hope address business value.

**Fit with Customer Desire**
One of the common themes in agile development methodologies is improving the fit of the solution with customer desire. Fit with customer desire is the result of close customer interaction fueled by
continuous delivery, iterative development, and progressive elaboration. With continuous delivery the customer can see what is being developed and provide guidance on the next most important improvement from the customer’s perspective. In iterative development, we plan to go back and make improvements to code that has been delivered to the customer. Progressive elaboration allows the team to apply feedback on the plan and solution as the team learns from what has actually been presented to the customer. This makes sure that the team builds what the customer wants.

But giving the customer what he or she wants does not necessarily translate to business value. For example, if it costs more to deliver the solution than our business can recoup, it actually destroys value. Even continuing refinement of valuable features eventually reaches diminishing returns. Whether the customer wants it or wants it better, at some point we have invested enough. Another example: infrastructure and refactoring are necessary, but our customer may not value these efforts as highly as a new feature. So fit with customer desire, while necessary, is not enough.

**Continuous Flow**

Agile has enabled continuous flow, or at least incremental delivery, of development tasks in a number of ways. First, low cost deployment, automated testing tools, and powerful development environments make it easy for teams to deploy code frequently. Process changes that limit WIP, support progressive elaboration, focus on quality and achieving “done-done”, maintain a continuous state of readiness, and enable refactoring allow teams to demonstrate high quality working code on a frequent basis.

While flow methods enable us to deliver high quality code quickly, they do not necessarily deliver business value. Working on features that are easy promotes flow but likely does little to reduce risk, promote learning, or increase revenue.

**Transparency**

Transparency is critical to agile methods and supports strong teams and collaboration. Incremental development, visible management tools, measures of cycle time and/or velocity all contribute to transparency. Transparency helps the development team identify work that is blocked and serves as a trigger to address obstacles to flow. It also allows teams to hold each other accountable since problems become obvious more quickly. This does little to help us identify whether we are working on the right thing.

**Collaborative Game**

The collaborative game is the set of ceremonies and supporting processes that facilitate improvement and learning – not only regarding the product but about the team’s ability to deliver the product. These ceremonies result in shared understanding and self organization. Again, this helps us get better as a team, but we need a clear roadmap of where we are going to get to business value. Without that, we are a high performance sports car without a map.
GETTING TO BUSINESS VALUE

Business value is not always clearly understood. There are multiple stakeholders in a business and they all have different interests and concerns. But business value is pretty straightforward. A project or product development effort that increases or protects revenue, reduces cost, or positions the organization for the future in alignment with an organization’s strategy delivers business value.

Technology is no longer the obstacle. The bargain of agile is to expend the least resource to accomplish the next business objective. This allows you to operate at a decision cycle faster and cheaper than your competition can achieve.

(Technology + Adoption) * Business Need = Business Value

The deal is to get focused on delivering what makes the most sense to the business. Early in a product development initiative, that means driving down risk -- then focusing on growing business
value. Just getting code developed isn’t enough. You have to be able to take the product from concept through completed development through adoption to the point of realizing business value.

This is hard enough to do in small development teams and organizations. Different people have different perspectives about what is important and why. Building a stable view of an emerging product is very difficult. Not everyone recognizes different types of risks. We look to the product owner for guidance but realize the daunting task of translating business needs to technical requirements. And the situation gets dramatically more challenging with large multi-team efforts, with multiple products, on complex architectures, all while trying to coordinate up and down stream in the Enterprise.

What if there was an agile way to ensure that the purpose and value proposition of each feature/story is communicated to development / QA and verified in customer acceptance.

**The How Trap**

At the root of this conversational divide, is something that has been called ‘the “how” trap’ and it affects us all. People often become so attached to how they do something (like sending a fax), that there description of their work often masks “what” they are doing (communicating the status of something is more likely to be “what” is being done, and the fax is the “how”). The “whats” that make up a product are what we also call business capabilities. This is hard to do because we typically communicate needs as how’s, not as what’s. We don’t “see” the value in the requirements. Look at the Fed Ex logo. Inside that logo is a symbol that is very meaningful. Until you see the symbol it is invisible. Once you see it, you will never look at the logo again without seeing this symbol.
We have found that describing the various business capabilities in a product to be an excellent first step in getting to a far clearer, more objective view of how customers will use the product and this work is fast and most people enjoy it. From there, adding pieces of information about the areas that are most and least valuable, performance requirements, and maturity or risk can lead to highly objective and efficient discussions about work prioritization, especially when business value is an element of the conversation.

Most of us have had the “why are we going this way?” conversation in the car, and that is the result of people getting attached to “how” they get to a favorite destination when it rarely matters “how” you accomplish the outcome, or objective, of getting there on time. It’s not that people are stupid, it’s simply that we often overlay “how” we do things in a way that starts to mask “what” the actual objective is, making it hard to think of any other way to achieve it.

Nowhere is ‘the “how” trap’ more common than in the workplace, and examples are the best way to illustrate that. If you are trying to collect business requirements for a specific part of an organization and, not knowing anything at all about the organization, you decide to walk up to a person at the fax machine sending a fax and ask them “what” they are doing. The response you are most likely to get is “I am sending a fax” and in that situation most people are likely to have follow up questions like “Is sending a fax a necessary part of the work you do? Do you have to send a fax to successfully complete your work?” and the answers are most likely to be “yes” which will lead the person to capture “send fax” as a requirement.

But it isn’t. The requirement, “what” they are doing, in this case is something more along the lines of “communicate status” or “confirm order” and “how” they are doing it is with the fax machine. So if you then go back to the person having disentangled the “what” from the “how” and ask if it matters “how” it gets accomplished, the worker will typically see that it doesn’t matter, and which has already transformed the conversation about requirements. The “whats” that make up the requirements are what we call business capabilities, and capturing the business capabilities is an effective and efficient way to get people out of ‘the “how” trap’. That is an important first step in getting to the thermal diagrams we call heat maps.

**OVERCOMING THE HOW TRAP**
So, what’s the first step out of the “how trap”? A product can be made up of hundreds of features/stories or product capabilities. Identify the capabilities with the highest risk and the highest business value.

First, look at the features and strip away the “how” verbs. The graphic below from a product that includes functionality to create insurance quotes helps illustrate identification of “how” verbs.

As an aside, notice that the verb “Automate” at the left maps to the right with “None” - that is because automate is neither a “how” verb nor a “what” verb. It is a secondary description of a “how” verb and should be used with caution in these discussions.

From here, the next step would be to document the view of the business capabilities. For example, here are the “whats” for the work that makes up creation of an insurance quote:
This by itself isn’t going to “wow” many people in an organization, but when you ask people about the business value of each block of work, including the parent “Create Quote” for an insurance quote, and then also ask how to define how each is performing, you can color each business capability, where the shades of red (pink and red) are flags for your attention (high value, and poor performance in this case) and the shades of green suggest the opposite (low value, performing well).

This is where the conversation starts to get significantly more objective and interesting. Now you can objectively ask which “child” business capabilities will cause the parent to perform better, and in this case, where the parent has a pink “Border” color that indicates medium-high business value,

just because the child “Create Certificate” is high value and the lowest performing, you may be able to ignore that problem because there are more important (high business value, low performing) parent capabilities. In defining this assessment framework you drive a shared understanding of what is most important to business and you can document it so that it becomes part of the priority decision making process. This is where part of the big impact of doing business capabilities analysis comes in, which opens the door to more objective business prioritization.

CAPABILITY ANALYSIS
Capability Analysis shares agile outcomes as it helps the organization more effectively scope, prioritize, and communicate requirements.

Fit with Business Strategy: Definition and assessment of capabilities is based on close stakeholder interaction and supports progressive elaboration.

Flow Across the Enterprise: Coordination around capabilities and focus on delivering value at the capability level.

Visibility of Strategic Prioritization: Visible management tools, measures of business value, suitability, and risk. Feedback lets us improve our approach.
Collaboration with the Business: Business and technology communities have overcome the How Trap. We can retain insight and decision information and set context for everyone – alignment arises from the resulting learning and self organization.

Clarify the Strategy (Context and Outcomes)
First, we must clarify context and outcomes for the product. The business probably has a lot of documentation around this but it is likely too much for most in the organization to consume. What is needed is a way to focus and communicate the product vision. We have successfully used the Lencioni’s Thematic Goal Model that provides focusing objectives that effectively shares the product vision. Facilitating shared agreement on this simple tool results in organizational clarity.
**Identify the Capabilities**

In the section, Overcoming the “How Trap”, we introduce a way to describe what the product *does* instead of “how” it is done by removing the “how” verbs. This may feel clumsy at first but really does focus the conversation on what and frees us of our preconceptions of how.

Developing our capabilities is very much like developing story maps (see the illustration below). Identify the big chunks of the product first. Then describe the smaller chunks for each bigger chunk. But, describe it in terms of outcomes. It takes some practice because we live in a process world and operational details.

Working through these descriptions, with outcomes in mind, with management and end users of the product will result is greater clarification and shared understanding. This exercise begins to build the bridge across the conversational divide between these groups. It doesn't take much time and is valuable in itself.

---

**Assess the Model**

So how do we color the model? In the first several sections, we explained that an understanding of business value was necessary to prioritize product development work. By developing the list of business capabilities, we now have a framework to discuss and assess the work. We also require an understanding of the risk and required performance of each business capability.

This can actually be done rather quickly by asking a few questions about each one. This is the approach we recommend initially. The framework also supports a more rigorous assessment.

*Business Value*
These few questions, asked for each capability, will quickly get to a fairly accurate assessment of business value. Is this capability:

- directly aligned with Strategic Themes and Focusing Objectives?
- mission critical?
- core to differentiating strategy?
- key to company's brand?

Blitz QFD, Kano Analysis, and the Business Value Game are some methods for a more rigorous assessment. These more rigorous assessment tools require metrics most organizations lack, a high level of maturity, and tend to be more opaque to the overall community. This simple Business Value assessment usually provides sufficient relative ranking.

Risk
Risk is the second characteristic that we must assess. In the section “Getting to Business Value”, we recommended addressing high risk elements initially.

- Is there technical risk associated with this capability (vendor stability, new technology)?
- Is there a compliance requirement or other business risk associated with this capability?
- Does changing this capability introduce a lot of dependent changes or risk?
- Are we aware of the Design, Development, and Delivery Risk?
- Is there a significant cost of delay with the delivery of this capability?

A FEMA can be used to consider risk more thoroughly. The heat map itself doesn’t replace a rigorous risk approach, but it does highlight where risk impacts the product feature roadmap.

Performance
Performance, current or required, is another key attribute we must assess. We must understand performance to guide iterations. How good is good enough?

- How is this capability performing today?
- How important is this capability to the product customer?
- Do we have insight into how to improve performance? This should be done at the performance attribute. For each capability, in the context of the current strategy, what is the status of the performance of this capability, what level of performance is tolerable for the strategy, and what is the goal?

Now we have colored the map and further developed a common understanding of where we agree and where we have differences. This identifies areas that demand further conversation.
Prioritize, Scope, and Communicate

From this assessment, you can prioritize those capabilities that are most important to achieving the objectives. Within each capability that is prioritized you can do a more detailed drill down into features and stories. Not all features and stories that can be conceived are of equal value to delivering the performance needed to deliver on the business value. Define Scope as specific changes in capabilities that are tied to the business objectives and in a testable fashion. Armed with this understanding at the “what” level developers and testers have a very clear understanding of what is needed. Now, with the delivery team, you can evaluate approaches to improvements based on cost and value. Only drill into those areas that are the next most important areas to focus (red business value – red performance) based on the current product strategy.

Prioritize remaining work:

- Risk First (New Technologies, Key Enabling, End-to-End Slice)
- Value Second and Only Associated Enabling (Focusing Theme)
- Portfolio Management (Risk and Value – Invest in a mix across themes and feature sets)

Focus, Deliver, and Learn

As the product is developed and feedback is gathered from the customer, it is likely that your understanding of the business value framework, the risks, or the performance gaps will evolve. One of the powerful aspects of this model is that you can rapidly reassess the backlog based on what is learned during the development of the product. As one strategic theme is sufficiently achieved, the backlog can be reprioritized against the next strategic theme.
Capability Analysis supports organizational learning resulting in a shared view across the business. This creates a clear context that empowers members of the team to improve locally in a fashion that is consistent with the larger plan.
**GUIDANCE FOR ORGANIZATIONAL DESIGN**

An important additional benefit of Capability Analysis is the support it provides for Organizational Design. The capability model will break down the product into features and components. As the capability model is decomposed, each feature set is broken down into a subset of prioritized features. Grouping the capability model at the high level outcome approach is very useful in defining how teams can best be organized to deliver the project. In larger, more complex organizations, the design of the delivery organization itself becomes a constraint. Treating shared resources and dependent teams as independent units, you can design your development “system” from first principles to serve customer demand. Effectively, this results in “service oriented enterprise architecture” for enterprise / organization business processes. Each unit exhibits services with SLAs. Use capability analysis to identify service boundaries and necessary levels of service, desired improvements, and out sourcing opportunities, including required SLA from vendor if/when required. This reduces coupling between organizational units that will result in high transaction costs and latency in delivery. Aligning technology platforms and the delivery organization with the product strategy and business model creates opportunities for much more effective delivery of software and a clearer focus on an organization design that is aligned with realizing the product strategy.

**SUMMARY**

Organizations that are looking for an improved method to scope, prioritize, and communicate to agile teams will benefit from Capability Analysis. Aligning the outcome based model with specific performance gaps results in clear and testable requirements. Assessing this model in the context of business value and risk helps the team prioritize work based on optimizing delivery of business value. The facilitated approach to defining the capabilities and the assessment framework, combined with the visible tool of the heat map results in dramatically improved communication and a shared understanding of what the team needs to deliver to meet the needs of the customer.
REFERENCES


SCRUM, XP, AND BEYOND:
One Development Team’s Experience Adding Kanban to the Mix
Frank X. Vega

With acknowledgement and thanks to: Manny Segarra, Michael McMahon, and Chad Bruns

ABSTRACT
As software development teams strive to effectively deliver value for their organizations, each must address their specific business needs along with similar (core) challenges they all share. To address some of these challenges, like many of you, I endeavored to increase my knowledge and understanding of lean-agile principles through hands-on experience gained from applying them in software development processes and practices.

This report captures what I hope for you will be key insights from my experience with a development team as we learned of Kanban (queuing systems) concepts and applied them to our existing Scrum/XP software development process. We first worked with these concepts on two small projects with four team members and then applied what we learned on a main-line project with an 11-14 member team over a 16 month period, which is the source for this experience report.
SCRAM, XP, AND BEYOND:  
ONE DEVELOPMENT TEAM’S EXPERIENCE ADDING KANBAN TO THE MIX

SETTING THE STAGE
The software development group consisted of 20-30 people within a 4,000+ employee manufacturing-type company purchased months earlier by a larger organization. In seven years of employment there, I saw acquisitions, selling off of divisions or whole businesses, restructurings, and several changes in upper management. The eventual home, in this sea of change, for this software development group was a new entity formed when several companies under the flagship were combined into one.

Prior to my coming onboard, half of the software development group left for a new entity as part of a joint venture with an external partner. Essentially, I signed on to “backfill”, but was sold on the opportunity to help lead change in a development environment where 6-8 month (or longer) development phases were standard. These were followed by months of test and fix cycles, followed by one or more patch distributions eventually leading to a stabilized application weeks (or months) after release to the field.

OUR FIRST STEP - 12 WEEK ITERATION
The first two years there, we built on RUP training that had occurred prior to my employment. We specifically emphasized the iterative and incremental development (IID) aspects of the Iterative Unified Process (IUP) and from this came the first key process change: a 12 week iteration cycle.

Several team members had learned some of Little’s Law, Theory of Constraints (TOC), Lean, Scrum, XP, etc. Yet none of us had in-depth knowledge or experience with these. But, in hindsight, by establishing the 12-week iteration, what was back then a “leap of faith”, I believe we took our “first steps” toward limiting work in progress (WIP), eliminating waste, delivering quicker, failing (learning) faster, creating knowledge, reducing lead/cycle times, and establishing a cadence (“heartbeat”) and flow.

FAST FORWARD - A TASTE OF LEAN-AGILE
Improvements to software architecture and engineering practices were also started. This included improving unit testing to address “bottlenecks” created by disproportionate dependence on script/record (GUI dependent) test tools. We later learned these “large batch” testing processes downstream from our “small batch” development processes were contributing greatly to our long development lead and cycle times. This impeded incremental feature delivery (flow) and delayed feedback, further decreasing quality.

However, these testing improvement efforts also highlighted deficient object-oriented programming and low-level coding practices, which resulted in external training on design patterns (see Shalloway, Net Objectives). Fortunately this training recognized the importance of taking a holistic view and included looking at XP practices and “lean concepts” applied to software development. This began an “amplification” of lean-agile software development learning and three years later, we were applying a fairly common form and mix of Scrum and XP practices:
Cross functional teams (typically 6-10 with most co-located)

- Daily scrums
- Project Board
- Story/Task Board
- Application/Release/Iteration Story backlogs
- Two week time-boxed iteration
- Point-sizing
- Velocity tracking
- Release burn-down
- Iteration burn-down
- Pairing – preferred, but accepted reviews within a short feedback timeframe (hours, day)
- Embedded product owners, business analysts, subject-matter experts (SMEs)
- Test Driven Development (TDD) – not all the time; at minimum established acceptance criteria
- Automated unit tests and acceptance tests
- Continuous integration/daily builds

By now, the testing bottleneck prior to release of an application’s next version had been reduced from months to weeks, even down to days in most cases. Fire-fighting and patches after a release disappeared. The 12 week iteration shrunk to two, though getting down from four was the most “anxious” change for some. By this time too, a number of prioritized stories were consistently completed every two-week iteration and “demo-ed” to stakeholders for feedback. However, more improvement was needed.

**Problems Remained**

There was a roughly “estimated” two year backlog with varying projects for the development group, and a business need for several applications to collaborate and integrate with each other. Yet, each team was dedicated to an on-going development project and operated in a “silo-ed” manner.

The application story backlogs that each team maintained consistently had 60-100+ stories with 15-20% queued for months (6-12). It was common for one team to spend 1-2 stressful days of a two-week iteration re-prioritizing existing stories, point sizing and prioritizing new stories, and essentially managing and grooming their unwieldy story backlog.

The focus on “velocity” also played out in unconstructive ways. “Pulling” stories based on current velocity but with aim to improve always felt like a “Push.” Frequently as burn-downs showed issues near mid-iteration, stories were de-scoped, including incomplete ones. Surprisingly, but not in retrospect, with less WIP, the remaining stories got completed quickly. Then, a de-scoped story or two would be “re-pulled” to fill the last bit of remaining iteration time. On another team, the back-and-forth flow of stories during iterations was not observed. However, while meeting goals for improving velocity and unit test code coverage, their technical debt increased over time. This was debt that got addressed only after changes were made to these processes.
In early 2007, I read an article on lean-agile portfolio (project) management that described how reducing WIP, increasing visibility, and focusing on lead and cycle times could help address our “starving” secondary projects and “silo-ed” development style. My understanding was still high-level at this time, but it was enough to start advocating for trying a lean portfolio project management approach.

Later that year at the Agile 2007 Conference, a member of our development group attended an open-space session led by David Anderson discussing “Kanban.” This led to us understanding more about limiting WIP. Since we had already observed ourselves how completed stories would spike each time we de-scoped at mid-iteration, this definitely piqued our interest. Also of great interest was hearing that “decoupling scope from schedule” might help us reduce the “push” to complete stories prematurely at iteration end, along with the negative effects this creates and that we were experiencing.

Shortly afterwards, several of us explored these concepts in depth via a conference call with David, who graciously offered to follow-up with our colleague. During our call David suggested that we also work to determine an average size item in our backlog. This suggestion shifted our focus to analysis and creating backlog items of similar size (to the average). Doing so would help us decrease variability in the time to complete backlog items and reduce efforts to forecast completion times as effectively, if not more, than attempting to (accurately) estimate each unique item.

This discussion with David gave me a better understanding of how reducing WIP, increasing visibility, and addressing constraints and flow could be consistently applied to managing our iteration, application, and broader portfolio backlogs and up and down the organizational level value stream for greater benefit.

**TIME TO TRY SOMETHING DIFFERENT**

Ultimately, with continuing pressure on the development group to address project backlog items beyond the three “mainline” applications, local management eventually agreed to let us try something different. As a result, key changes were introduced into our development process with most “evolving” over weeks and months into the following:

1. Consolidated “backlogs” (inventories of work requests).
   a. A single prioritized MMF/Epic level portfolio backlog combining all project and application work requests into one list, limited to 16 weeks of work for the group.
   b. Development teams stopped using application/release level backlogs with large numbers of stories; these transformed into small background tracking lists for product owners.
   c. A goal to deliver 6-9 similarly sized MMFs spread across three releases per year (at this point our releases were still approximately one per year).

2. Decoupled intervals used for planning/prioritizing, reviewing, releasing, and retrospectives.
   a. Development teams stopped pulling stories every two weeks and went to pulling from a short (8-12 MMFs) portfolio backlog when they freed-up (finished an MMF).
A team forming policy was developed to favor stable teams but allowed dynamic forming as needed.

b. Upon pulling an MMF, the team identified development goals and user acceptance criteria. A prioritized, JIT (just-in-time) Story Backlog was created with typically 10-15 stories, just enough to validate preliminary analysis and “ball-park” MMF sizing.

c. Demos (reveals) were conducted every two weeks to show completed work to extended business stakeholders; however, revealed stories had already been reviewed, again JIT, with MMF product owners/stakeholders as a step to Done-Done.

d. Field releases occurred according to business needs, usually governed by marketing. Frequent internal releases occurred after a significant feature or two were completed or “in the can.”

e. Teams conducted formal “retros” every two weeks but also employed a “stop-the-line” mentality allowing a problem to be addressed JIT when and if it made sense to do so.

3. Development team applied “Kanban” concepts formally on the MMF and story level work.
   a. Established a limit WIP of “one” MMF per development team.
   b. Established WIP limits for the story value stream phases: the JIT Story Backlog (prioritized but dynamic), the ToDo (on-deck, buffer) phase, the Doing (agile/XP development) phase, and the Done-Done phase representing potentially shippable value.
      i. Limit for the JIT Story Backlog was set at a “soft” 15 stories (meaning, typically it was less); as work progressed on the pulled MMF, new stories were added JIT with less of a concern for “discovering” every story for the MMF up-front.
      ii. Limit for the ToDo phase was set at a “soft” six stories (again, typically it was less) to ensure a buffer of ready stories.
Note: the team quickly established a flow of approximately one story finished per day which remained fairly consistent; this meant the JIT Story Backlog and ToDo limits were representative of a two week buffer, and a six day buffer, respectively.
      iii. Limit for the Doing phase was set according to the number of team members divided by two. This supported our use of pairing on stories in process or when addressing “blocks” placed on any (as we continued using XP practices).
      iv. Done-Done visualized the stories completed for the MMF; but there was no specific limit and it was simply cleared each time an MMF was completed.
   c. Established story pulling guidelines focused on limiting WIP and minimizing story lead and cycle times.
      i. Guidelines for pulling stories into Doing (see Scotland, Kanban, Flow, Cadence):
         1. Nothing to work on?
            
            *Ask if you can help someone else (on any WIP).*
2. Don’t have the right skills for that?
   *Work to remove any existing bottleneck.*

3. Don’t have the right skills for that (or no bottlenecks currently)?
   *Pull story from ToDo.*

4. Don’t have the right skills for that?
   *See PO about lower priority work in Backlog.*

5. Don’t have the right skills for that?
   *Find other interesting work or BUILD YOUR SKILLS.*

ii. Development team started with a physical board to map and track the story level value stream, switched to a software tool, and then to a combination of both.

Note: the board provided visibility at the story level while the software tool provided visibility at the story and task level. However, “daily tasks” were entered in the software tool only, to communicate with remote team members about what was happening that day on a WIP story.

Note: Small white cards in Doing column designated team members working on respective stories; the story at the top is “blocked” (indicated by a bright pink card next to it), and the bottom story card without a white name card next to it is an on-going non-functional process reminder. White name cards at the bottom of the “Backlog” column indicate team members “out” for the
day, or “distracted” (not working on the WIP MMF stories at that moment), or SMEs not currently needed for the day.

d. Established a “low-effort” self-generating system to collect real-time data and compute MMF and story metrics for lead and cycle times and flow.

e. Stopped using point sizing, velocity (metric), burn-down charts, and fixed two week iterations. Employed iterative development, incremental delivery, time-boxes, and spikes, all with a focus on establishing predictable value delivery (flow) and reducing lead and cycle time averages and variations by improving quality and reducing waste.

f. Fully consumed the JIT Story Backlog as part of completing an MMF, including any “field bugs” found and “injected” into the current MMF’s JIT Story Backlog. This did not preclude adding “discoveries” to the Portfolio Backlog as an MMF for future consideration.

WHAT HAPPENED?
The changes we made to incorporate Kanban concepts, along with the outcomes, evolved over weeks and months. Here are the significant results that occurred at the development team level:

1. Smoother continuous planning
   a. Portfolio (MMF) level planning
      i. Initially, a portfolio planning and resource check meeting occurred every two weeks with all development team members present. After the changes, this was reduced to a few team representatives meeting with product owners on average every 37 days (our cadence) as an MMF neared completion.
      ii. The portfolio planning meeting also went from a half-hour down to a 5–10 minute review of the “short” portfolio backlog and “pulling” the next prioritized “ready” MMF matching the team’s capability.
   b. Story level planning
      i. After each Portfolio planning meeting, a JIT Story Backlog was created during a 2-4 hour meeting, occurring again on average every 37 days (median 33.5) each time an MMF was pulled. This replaced spending 1-2 days every two weeks to “iteration plan” and groom backlogs with 60-100 stories.
      ii. Time previously spent by the development team grooming the story backlog transformed (disappeared) into a JIT (ongoing) incremental activity integrated as part of the day-to-day work of completing the MMF.

2. Manageable Backlogs and Fresh Stories
   a. The Portfolio (MMF) backlog became a dynamic list of MMFs not in process, and it was prioritized as needed. However, in terms of work effort, it was maintained relatively stable to represent roughly only 16 weeks of work requests for the development group.
b. As work began on an MMF, stories were pulled into the Doing phase, quickly lowering the JIT Story Backlog well below its “soft” limit of 15. Then, as work progressed, stories were added JIT to this backlog, maintaining its count often only at four to eight.

c. Final count of Done-Done stories for a completed MMF ranged from 30-60 (average 32, median 24, standard deviation 30). However, 50% to 75% of the stories were created “as we worked” on the MMF. That is, most of the stories were created after the MMF was pulled and still after the first 10-15 stories were developed to validate “size.” No time-boxed two week iterations and no more de-scoping then re-pulling (“yo-yo stories”) as was observed prior to these process changes.

d. Since the entire JIT Story Backlog was consumed as part of completing an MMF, again on average every 37 days, including any field bugs “injected” into the backlog, a story (or field bug) was never more than a few weeks old prior to being worked on.

3. Collaboration Increased
   a. Pairing before the changes was typically one pair per story, but after the process changes it became common to see one or two of the stories in the Doing phase each being “swarmed” on by two pairs (even three pairs at times).
   b. The daily scrum also became frequently supplemented now by spontaneous huddles occurring almost daily between two pairs, or even the whole team.
   c. Blocks and Impediments
      i. Efforts to resolve “blocked issues” (impediments) were more responsive when there was a focus on minimizing lead and cycle times and limiting WIP stories. Definitely better than what we observed when using fixed two-week iterations, velocity, and committing to completing a specific set of “pulled” story points.
      ii. When “blocked issues” needed resources from outside the team, as per our guidelines, a blocked pair moved to “swarming” on WIP stories. They would attempt to join with another pair on WIP, or split up and help on two separate WIP stories, before considering pulling a new story or other valued work (ex. refactorings, process improvements identified in retrospectives, training, etc.)
      iii. When possible, blocked stories were modified so that any valued part that could move forward did, and the blocked part “rolled back” for future consideration.

4. Value Delivery Increased, Quality Increased, and Technical Debt Decreased
   a. Over the 16 month reporting period, 18 MMFs were completed; seven in the first eight months, and 11 in the second eight months.
   Note: At times multiple MMFs were worked on concurrently.
   b. In the first seven MMFs completed, 29 field bugs were deemed necessary to inject into the JIT Story Backlogs. Only five field bugs were submitted in the last 11 completed MMFs. The inference here is the changes reduced the bugs that escaped to the field, and decreased their severity, frequency, or both.
c. While completing 18 MMFs during this 16 month period, the development team was able to address several issues related to problematic architecture, testing, and technical debt. These issues had accumulated previously and were addressed only after these changes in process. They were significant enough to require intermittent team effort spread over several months to address. Note: The “development team” maintained this issues/technical debt backlog. A “story” from this backlog which most directly supported the delivery of the current MMF WIP was also pulled. Since this story was not time dependent and often done as stages of improvements, this "pair" in an unplanned event (emergency) could jump back on main MMF stories as necessary. This "slacker pair" represented intentional system slack, and by the debt addressed and flow achieved, it was apparently effective.

d. System improvement suggestions came from retrospectives; with each MMF pulled, the team also picked a retrospective improvement item and “sized” an appropriate effort that become part of the specific goals for the MMF as a "doing things right" part alongside the "MMF value to deliver" part, both developed with product owner and management’s input. More often, this story took lower priority while we worked on the MMF but as part of our definition of "Done-Done" for the MMF these stories had to be completed too.

5. Improved Ability to Project Completion
   a. Using actual lead times, cycle times, and flow metrics computed for MMFs and stories, the development team was able to “forecast” MMF completion dates, four to eight weeks out. We were more accurate (a business week or less), had more precision (no missed field release commitments), and did this with less effort than we had been able to do with task break down (time) or point sizing methods. Note: During the 16 month reporting period, there were two special cause issues that resulted in significant lead and cycle time increases and a need for overtime efforts to meet field release commitments.

Conclusions
I will conclude with the “two minute” version of this experience integrating Kanban (queuing systems) concepts into our software development processes.

Adding visibility to lead times, cycle times, and flow notably changed and improved the team’s existing combination of Scrum and XP development process and practices by:

- Establishing a consistent, smoother, and more predictable delivery of value (field software releases) than we had done before.
- Creating fewer field bugs over time, while simultaneously reducing both the level of prior technical debt and creation of new technical debt; this was something not observed previously by our teams.
- Changing the perspective of “value” to completing and delivering quality work faster and away from maintaining large accumulations of unfinished WIP inventories.
Using self-generating data to provide visibility into the effects of increasing WIP on lead times, cycle times, and flow (of value delivery). Attaining a deeper understanding of “pull” vs. “push” and the effects each have on the team’s software development capacity.

Increased understanding of how upstream and downstream processes impact the software development process (ex. delays in decisions, information, customer and stakeholder feedback, approvals, etc.)

As part of these process changes, we also computed and charted cycle times on MMFs as they were completed over the 16 month period. By analyzing this visual display we learned a great deal about how flow principles played out in our software development process over this time.

The latest completed MMF was charted to the left as earlier ones scrolled right like a heartbeat monitor. When multiple MMFs were worked on simultaneously, they were numbered with repeating letters (N1-N3, E1-E2). This chart version also shows the running MMF cycle time mean along with plus/minus one standard deviation.

MMF E1 and E2 represent the first time we pulled and worked on two MMFs simultaneously. After 42 days, both were completed for an average cycle time of 21 days apiece. Clearly these two independent MMFs could have been completed serially and each under the cycle time mean (28 days) that existed when we pulled them. If we had done that, one potentially would have been available nearly three weeks earlier than it was by us working on them concurrently.

During MMF F a special business event occurred which significantly impacted capacity. MMF F’s cycle time was nearly triple the running cycle time mean prior to pulling MMF F. It was also nearly five times the running standard deviation above the running cycle time mean. What contributed to this?

Other data we tracked across the 16 month period showed each team member spent on average six hours per week on “non-development work” (15% of their 40 hour week). However, during this special cause event, while MMF F was in process, each team member on average spent 10 hours per week on “non-development work” (25% of their 40 hour week). Inspecting the development work itself also showed that during this period eight of the 55 stories (15%) completed were “special event” stories, additional WIP, not related to MMF F. These “special” stories came with fixed
completion dates and required overtime hours by several on the team. It clearly showed in our context that increasing “non-development work” by 10% and introducing non-related additional WIP which consumed all “slack” (plus required overtime) had a significant impact on the team’s capacity. In retrospect, this event was an unidentified class of service injected into our software development process.

In contrast, MMF K, L, and M were small planned features yet each of these were pulled serially and like few cars than run on the freeways at night each flew through development with cycle times nearly a standard deviation (20-22 days) under the running cycle time mean (32-35 days) when pulled.

Unfortunately, another special (cause) business event followed shortly and with it an attempt to “pull” (or more accurately “push”) what turned out to be multiple MMFs (N1, N2, and N3). Once again, the principles of flow clearly showed us the consequences, and the development team needed a significant amount of overtime over several weeks to restore the flow. Another lesson learned.

Special acknowledgement and thanks to: Manny Segarra, Michael McMahon, and Chad Bruns for all the assistance and support they provided me during this period and especially for the courage to learn by “doing something different.”