Kanban

Successful Evolutionary Change for Technology Organizations

David J. Anderson
dja@agilemanagement.net

Revision 1.07.2 Jan 8th 2010

Table of Contents

Kanban *Successful Evolutionary Change for Technology Organizations* **1**

Table of Contents **2**

Section One Introduction **9**

Chapter 1 Solving an Agile Manager's Dilemma **10**

My Search for Sustainable Pace 10

My Search for Successful Change Management 11

From Drum-Buffer-Rope to Kanban 14

Emergence of the Kanban Method 15

Kanban's Community Adoption 15

The Value of Kanban is Counter-Intuitive 16

Takeaways 17

Chapter 2 What is the Kanban Method? **18**

The Imperial Palace Gardens 18

What is a Kanban system? 19

Kanban Applied in Software Development 20

Why use a kanban system? 21

A Definition of the Kanban Method 21

Kanban 22

Kanban as a Permission Giver 22

Takeaways 24

Section Two Benefits of Kanban **25**

Chapter 3 A Recipe for Success **26**

Implementing the Recipe 27

Focus on Quality 28

Reduce Work-in-Progress, Release Often 29

Balance Demand against Throughput 34

Prioritize 35

Reduce Variability and Improve the Process 36

Recipe for Success and Kanban 37

Takeaways 38

Chapter 4 From Worst to Best in Five Quarters **39**

Visualize the Wokflow 40
Factors Affecting Performance........ 40
Make Process Policies Explicit........ 42
Estimation was a Waste 42
Limit Work-in-progress 42
Establishing an Input Cadence 43
Striking a New Bargain 44
Implementing Changes 45
Adjusting Policies 45
Looking for Further Improvements 46
Results 47
Takeaways 50
Chapter 5 A Continuous Improvement Culture........51
Kaizen Culture 51
Kanban accelerates organizational maturity and capability 52
Case Study: Corbis Application Development 52
Background and Culture........ 53
The need for a Software Maintenance Function 54
Small Projects for Maintenance wasn't Working........ 55
Implementing Change........ 55
Primary Effects of the Changes 56
Unanticipated Effects of Introducing Kanban........ 56
Sociological Change........ 56
Viral Spread of Collaboration 58
Bargaining........ 58
Democracy 58
Down with Democracy........ 59
Collaboration 59
Cultural Change is perhaps the biggest benefit of Kanban 60
Takeaways 61
Section Three Implementing Kanban 62
Chapter 6 Mapping the Value Stream 63

Defining a Start and End Point for Control 63
Work Item Types 63
Anatomy of a Work Item Card 64
Drawing a Card Wall 65
Electronic Tracking 67
Setting Input & Output Boundaries 68
Coping with Concurrency 68
Coping with Unordered Activities 69
Takeaways 70
Chapter 7 Coordination in Kanban Systems 71
Visual Control & Pull 71
Electronic Tracking 72
Daily Standup Meetings 74
The After Meeting 74
Prioritization Meetings 75
Release Planning Meetings 76
Triage 76
Issue Log Review and Escalation 78
Sticky Buddies 78
Synchronizing across Geographic Locations 79
Takeaways 80
Chapter 8 Establishing a Delivery Cadence 81
Coordination Costs of Delivery 82
Transaction Costs of Delivery 83
Efficiency of Delivery 85
Agreeing a Delivery Cadence 86
Improve Efficiency to Increase Delivery Cadence 86
Making On-Demand or Ad Hoc Deliveries 87
Takeaways 90
Chapter 9 Establishing an Input Cadence 91
Coordination Costs of Prioritization 91
Agreeing a Prioritization Cadence 93

Efficiency of Prioritization 93
Transaction Costs of Prioritization 94
Improve Efficiency to Increase Prioritization Cadence 95
Making On-Demand or Ad Hoc Prioritization 96
Takeaways 98
Chapter 10 Setting Work-in-Progress Limits 99
Limits for Work Tasks 99
Limits for Queues 100
Buffer Bottlenecks 100
Input Queue Size 101
Unlimited Sections of Workflow 102
Don't Stress Your Organization 103
It's a Mistake Not to Set a WIP Limit 103
Takeaways 105
Chapter 11 Establishing Service Level Agreements 106
Typical Class of Service Definitions 106
Expedite 106
Fixed Delivery Date 107
Standard Class 108
Intangible Class 109
Policies for Class of Service 109
Expedite Policies 110
Fixed Delivery Date Policies 110
Standard Class Policies 110
Intangible Class 111
Determining a Service Delivery Target 112
Putting Classes of Service to Use 113
Takeaways 114
Chapter 12 Metrics & Management Reporting 115
Tracking WIP 115
Cycle Time 116
Due Date Performance 117

Throughput 118
Issues & Blocked Work Items 119
Flow Efficiency 120
Initial Quality 120
Failure Load 120
Takeaways 122
Chapter 13 Scaling Kanban with Two-tiered Systems 123
Hierarchical Requirements 124
Decouple Value Delivery from Work Item Variability 124
Two-tiered Card Walls 126
Introducing Swim Lanes 127
Incorporating Classes of Service 128
Takeaways 129
Chapter 14 Operations Review 130
Takeaways 135
Chapter 15 Getting Started with Kanban 136
Kanban accelerates organizational maturity and capability 136
The Primary Goal for our Kanban System 137
Secondary Goals for our Kanban System 137
Know the Goals and Articulate the Benefits 142
Steps to Get Started 143
Kanban Strikes a Different Type of Bargain 145
Striking the Kanban Bargain 146
Takeaways 152
Section Four Making Improvements 153
Chapter 16 Three Types of Improvement Opportunity 154
Bottlenecks, Waste Elimination, and Variability Reduction 154
Theory of Constraints 154
Five Focusing Steps 155
Lean, TPS & Waste Reduction 156
Deming and Six Sigma 157
Fitting Kanban to your Company Culture 158

Takeaways 159
Chapter 17 Bottlenecks & Non-instant Availability 160
Capacity Constrained Resources 161
Elevation Actions 161
Exploitation/Protection Actions 162
Subordination Actions 165
Non-instant Availability Resources 166
Exploitation/Protection Actions 168
Subordination Actions 169
Elevation Actions 170
Takeaways 172
Chapter 18 An Economic Model for Lean 173
Redefining "Waste" 173
Transaction Costs 175
Coordination Costs 176
How do you know if an Activity is a Cost? 178
Failure Load 179
Inventory Depreciation 179
Takeaways 181
Chapter 19 Sources of Variability 182
Internal Sources of Variability 183
Work Item Size 184
Work Item Type Mix 184
Class of Service Mix 186
Irregular Flow 186
Rework 187
External Sources of Variability 187
Requirements Ambiguity 187
Expedite Requests 189
Irregular Flow 190
Environment Availability 191
Other Market Factors 191

Staffing Resource Factors 192
Difficulty Scheduling Coordination Activity 194
Takeaways 195
Chapter 20 Issue Management & Escalation Policies 196
Managing Issues 197
Escalating Issues........ 198
Tracking and Reporting Issues 198
Takeaways 200
Chapter 21 An Operational Decision Framework 201
The Lean Decision Filter 201
The Agile Decision Filter 204
Takeaways 207

Section One
Introduction

Chapter 1 Solving an Agile Manager's Dilemma

In 2002, I was an embattled development manager in a remote outpost of Motorola's PCS (cell phone) division based in Seattle, Washington. My department had been part of a startup company Motorola had acquired a year earlier. We developed network server software for wireless data services such as over-the-air download and over-the-air device management. These server applications were part of integrated systems that worked hand-in-hand with client code on the cell phones as well as other elements within the telecom carriers' networks and back office infrastructure such as billing. Our deadlines were set by managers without regard to engineering complexity, risk or project size. Our code base had evolved from a startup company where many corners had been cut. One senior developer insisted on referring to our product as "a prototype." We were in desperate need of greater productivity and quality in order to meet demands from the business.

In my daily work, back in 2002, and through my writing efforts, on my earlier book[i], two main challenges were on my mind. Firstly, how did I protect my team from the incessant demands of the business and achieve what the Agile community now refers to as a "sustainable pace?" And secondly, how could I successfully scale adoption of an agile approach across an enterprise and overcome the inevitable resistance to change?

My Search for Sustainable Pace

Back in 2002, the Agile community referred to the notion of a sustainable pace as simply "the 40 hour week[ii]". The Principles behind the Agile Manifesto[iii] told us that, "Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely." My team at Sprint PCS, two years earlier had become used to me telling them that "large scale software development is a marathon, not a sprint." If team members were to keep up the pace for the long haul on an 18 month project we couldn't afford to burn them out after a month or two. The project had to be planned, budgeted, scheduled and estimated so that team members could work reasonable hours each day and avoid tiring themselves out. The challenge for me as a manager was to figure out how to achieve this goal while accommodating all the business demands.

In my first management job, in 1991, at a 5 year old startup that made video capture boards for PCs and other smaller computers, the CEO gave me the feedback that I was seen by the leadership as "very negative." I was always telling them "No" when our development capacity was already stretched to the maximum and they asked for yet more products or features. By 2002, it wasn't a new pattern. I'd spent 10 years saying "no" pushing back against the constant and fickle demands of business owners.

In general, software engineering teams and IT departments seemed to be at the mercy of other groups who would negotiate, cajole, intimidate and over-rule even the most defensible and objectively derived plans. Even plans based on thorough analysis and backed by years of historical data were vulnerable. Most teams who had neither a thorough analysis method or any historical data, were helpless at the hands of others who would push them to commit to something unknown and often completely unreasonable.

Meanwhile, the workforce has come to pretty much accept that crazy schedules and ridiculous work commitments are the norm. Software engineers are apparently not supposed to have a social life or a family life. It feels abusive, because it is! I know too many who have irreparably damaged relationships with children and other family members through commitment to work. It's hard to have sympathy for the typical software development geek, though. In my home state of Washington, in the United States, software engineers are second only to dentists as the highest paid profession. Like Ford assembly line workers in the 2nd decade of the 20th Century earning five times the average wage in the United States, no one cared about the monotony of the work, or the wellbeing of the workers, when they were so well remunerated. It's hard to imagine the emergence of organized labor in knowledge work fields like software development. It's hard to imagine anyone addressing the root causes. Richer employers have been apt to add additional health care benefits, such as massage, psychotherapy, and provide "mental health" sick days, rather than pursue the root causes of the problem. A technical writer at a well known software firm once commented to me, "there is no stigma about being on antidepressants, everyone is doing them!" In response to this abuse, software engineers tend to acquiesce to demands, collect their fancy salaries and suffer the consequences.

I wanted to break that mold. I wanted to find a "win-win" approach that allowed me to say "Yes" and still protect the team and facilitate a sustainable pace. I wanted to give back to my team – to give them back a social and family life. I wanted to bring an end to the stress-related health issues that were affecting developers in their late 20s. So I decided to take a stand and try to do something about this problem.

My Search for Successful Change Management

The second thing on my mind was how to lead change in large organizations. I'd been working as a development manager at Sprint PCS and later at Motorola. In both cases, there was a real business need to develop a more agile way of working. But in both cases I had struggled to scale agile methods across more than one or two teams.

In both cases, I didn't have sufficient positional power to simply impose change on a large number of teams. I was trying to influence change, at the request of senior leadership but without any positional power. I had been asked to influences peers to make changes in their teams to match the changes I had implemented with my own team. These other teams resisted adopting techniques that were quite clearly producing better results with my team. There were probably many facets to this resistance but the most common theme was that every other team's situation was different. The techniques used on my team would have to be modified and tailored to their specific needs. By mid-2002, I had concluded that prescriptively enforcing a software development process on a team didn't work.

It was necessary to adapt a process to the specific situation. To do this required local leadership. This was often lacking. Even with the right leadership, I doubted if significant change could happen. I doubted it because a framework and guidance for how to tailor and adapt a process to a different situation was missing. Without a framework to guide the leader, coach or process engineer any adaptation was likely to be subjectively imposed based on superstitious beliefs. This was just as likely to raise hackles and objections as the imposition of an inappropriate process template.

I had partly set out to address this issue with the book I was writing at the time, "Agile Management for Software Engineering." I was asking "why does agile development produce a better economic outcome than traditional approaches?" and I sought to use the framework of the Theory of Constraints[iv] to make the case.

While researching and writing the book, I came to realize that it was very likely that in some way, every situation was unique. Why should the constraining factor or bottleneck be in the same place on every team and every project, every time? Every team is different: different sets of skills, capabilities and experience. Every project is different: different budget, schedule, scope and risk profile. And every organization is different: a different value chain operating in a different market. It occurred to me that this might provide a clue to the resistance to change. If proposed changes in working practices and behavior did not have a perceived benefit then there would be resistance. If those changes did not affect what the team perceived as their constraint or limiting factor then they would resist. Simply put changes suggested out of context would be rejected by the workers who lived and understood the project context.

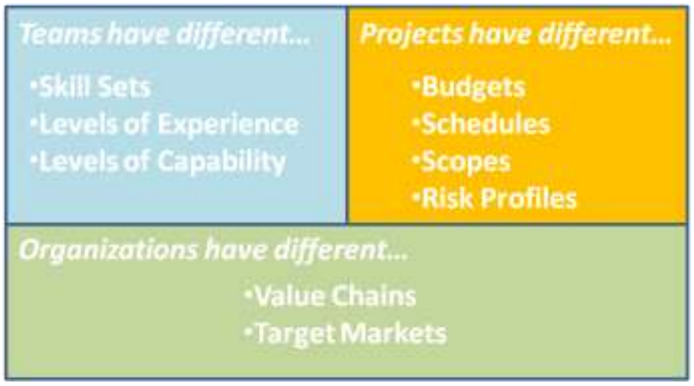


Figure 1. Why "one size fits all" development methodologies don't work

It occurred to me that it would be much better to let a new process evolve by eliminating one bottleneck after another. This is the core thesis of the Goldratt's Theory of Constraints. While recognizing that I had a lot to learn, I felt there was value in the material and I pressed ahead with the original planned manuscript. I was fully in the knowledge that my book did not provide advice on how to implement the ideas at scale as it offered little or no advice on change management.

Goldratt's approach, explained in Chapter 16, seeks to identify a bottleneck and then find ways to alleviate it until it is no longer constraining performance. Once this happens a new bottleneck will reveal itself and the cycle repeats. It's an iterative approach to systematically improving performance by identifying and removing bottlenecks.

I recognized that I could synthesize this technique with some ideas from Lean. By modeling the workflow of a software development lifecycle as a value-stream and creating a tracking and visualization system to track state changes of emerging work as it "flowed" through the system, I could see bottlenecks. An ability to identify a bottleneck is the first step in the underlying model for the Theory of Constraints. Goldratt had already developed an application of the theory for flow problems, awkwardly called "Drum-Buffer-Rope." Regardless of the awkwardness of the name, I realized that a simplified Drum-Buffer-Rope solution could be implemented for software development.

Generically, Drum-Buffer-Rope is an example of a class of solutions known as *pull* systems. As we will learn in Chapter 2, a kanban system is another example of such a *pull* system. An interesting side-effect of pull systems is that they limit work-in-progress to some agreed quantity thus preventing workers from becoming overloaded. In addition, only the workers at the bottleneck station remain fully loaded and everyone else should experience some slack time. I realized that pull systems had the potential to deliver a solution to both of my challenges. A pull system would enable me to implement process change incrementally, with (hopefully) significantly reduced resistance, and it would facilitate a sustainable pace. I resolved to try to implement a Drum-Buffer-Rope pull system at the earliest opportunity. I wanted to experiment with incremental process evolution and see whether it enabled sustainable pace and reduced resistance to change.

The opportunity to try arrived in the fall of 2004 at Microsoft and is fully documented in the case study in Chapter 3.

From Drum-Buffer-Rope to Kanban

Implementing a Drum-Buffer-Rope solution at Microsoft worked well. With very little resistance, the productivity more than tripled and lead times shrunk by 90% while risk was reduced through greatly improved predictability. By the fall of 2005 I reported the results at a conference in Barcelona[v] and again in winter of 2006. My work came to the attention of Donald Reinertsen who made a special trip to visit me at my office in Redmond, Washington. He wanted to persuade me that I had all the pieces in place to implement a full kanban system.

"Kan-ban" is a Japanese word that literally means "signal card" in English. In a manufacturing environment, this card is used to signal an upstream step in a process to produce more. The workers at each step in the process are not allowed to do work unless they are signaled with a kanban from a downstream step. While I was aware of this mechanism, I was not convinced that it was either a useful technique or a viable technique for application to knowledge work and specifically software engineering. I understood that kanban systems would enable sustainable pace. However, I was not aware of its reputation as a method for driving incremental process improvement. I was unaware that Taiichi Ohno, one of the creators of the Toyota Production System had said, "The two pillars of the Toyota production system are just-in-time and automation with a human touch, or autonomation. The tool used to operate the system is kanban." In other words, kanban is fundamental to the kaizen (or "continuous improvement") process used at Toyota. It is the mechanism that makes it work. I have come to recognize this as a complete truth through my experiences over the five years since.

Luckily, Don made a convincing argument that I should switch from Drum-Buffer-Rope implementations to a kanban system for the highly esoteric reason that a kanban system makes a more graceful recovery from an outage in the bottleneck station than Drum-Buffer-Rope. Understanding this idiosyncrasy is not important for your understanding and reading of this book.

Revisiting the final implemented solution at Microsoft, I realized that, had we conceived of it as a kanban system from the beginning then the outcome would have been identical. It was interesting to me that two different approaches would result in the same outcome. So if the resultant process was the same, I didn't feel compelled to think of it specifically as a Drum-Buffer-Rope implementation.

I developed a preference for the term "kanban," over Drum-Buffer-Rope. Kanban is used in Lean manufacturing (or the Toyota Production System). This body of knowledge has much wider adoption and acceptance than the Theory of Constraints. The term "kanban", while Japanese, is less metaphorical than Drum-Buffer-Rope. Kanban was easier to say, easier to explain and as it turned out easier to teach and implement, so it stuck.

Emergence of the Kanban Method

In September 2006, I moved from Microsoft to take over the software engineering department at Corbis, a privately held stock photography and intellectual property rights business based in downtown Seattle. Encouraged by the results from Microsoft, I decided to implement a kanban pull system at Corbis. Again the results were encouraging and led to the development of most of the ideas presented in this book. It is this expanded set of ideas that includes, workflow visualization, work item types, cadence, classes of service, specific management reporting, and operations reviews that defines the Kanban method explained in these pages.

In the remainder of this book I describe Kanban (capital K) as the evolutionary change method that utilizes a kanban pull system (small k), visualization and other tools described in this text. Kanban enables you to achieve context specific process optimization with minimal resistance to change while maintaining a sustainable pace for the workers involved.

Kanban's Community Adoption

In May 2007, Rick Garber and I presented the early results at the Lean New Product Development conference in Chicago. Later that summer, at the Agile 2007 conference in Washington D.C., I held an open space session to discuss kanban systems. About 25 people attended. Two days later, one of the attendees, Arlo Belshee, gave a lightning talk sharing his Naked Planning[vi] technique. It appeared others had been implementing pull systems too and often for similar reasons. A Yahoo! discussion group was formed and quickly grew to 100 members. At the time of writing it has over 1000 members. Several of the attendees from the open space session committed to trying kanban in their workplace, often with teams who had struggled with Scrum. The most notable of those early adopters were Karl Scotland, Aaron Sanders and Joe Arnold, all from Yahoo! who quickly took kanban to more than 10 teams on 3 continents. Another notable attendee at the open space was Kenji Hiranabe who had been developing kanban solutions in Japan. Soon afterwards he wrote two notable articles for InfoQ[vii, viii] on the topic. In the fall of 2007 Sanjiv Augustine, author of "Managing Agile Projects"[ix] and a founder of the

Agile Project Leadership Network (APLN) visited Corbis in Seattle and described kanban as the “first new agile method I’ve seen in five years.”

The following year, at Agile 2008, in Toronto, there was 6 presentations about the use of kanban solutions in different settings, including one from Joshua Kerievsky, of Industrial Logic, an Extreme Programming consulting and training firm, who showed how he had evolved similar ideas to adapt and improve Extreme Programming to his business context. That year, the Agile Alliance awarded the Gordon Pask Award to Arlo Belshee and Kenji Hiranabe who had both made a visible contribution to the emergence of Kanban or produced remarkably similar ideas.

The Value of Kanban is Counter-Intuitive

In many ways knowledge work is the antithesis of a repetitive production activity. Software development is most certainly not like manufacturing. The domains exhibit wildly different attributes. Manufacturing has low variability while much of software development is highly variable and seeks to exploit variability in design and novelty in order to drive profit. Software is by nature “soft” and can often be easily and cheaply changed while manufacturing tends to be about “hard” things that are difficult to change. It’s natural to be skeptical about the value of kanban systems in software development and other IT work. Much of what we, as a community, have learned about Kanban over the last few years is counter-intuitive. No one predicted the effect on culture or the improved cross-functional collaboration. In these pages I hope to show you that “Kanban can!” And in doing so, convince you that the simple rules of Kanban can improve productivity, reduce delivery times with improved predictability, improve customer satisfaction and in doing all this will change the culture of your organization and improve collaboration and relationships with other parts of the business.

Takeaways

- Kanban systems are from a family of approaches known as pull systems
- Eliyahu Goldratt's Drum-Buffer-Rope application of the Theory of Constraints is an alternative implementation of a pull system
- The motivation for pursuing a pull system approach was two-fold: the desire to find a systemic way to achieve a sustainable pace of work; and the desire to find an approach to introducing process changes that increased acceptance
- Kanban is the mechanism that underpins the Toyota Production System and its "kaizen" approach to continuous improvement
- The first virtual kanban system for software engineering was implemented in 2004 through 2005 at Microsoft
- Results from early Kanban implementations were very encouraging in regards to achieving sustainable pace, reduced resistance to incremental evolutionary change, and producing significant economic improvements
- The Kanban Method as an approach to change started to grow in community adoption after the Agile 2007 conference in Washington D.C. in August 2007
- Throughout this text "kanban" (small "k") will be used to refer to signal cards, and "kanban system" (small "k") will refer to a pull system implemented with (virtual) signal cards
- Kanban (capital "K") will be used to refer to the methodology of evolutionary incremental process improvement that emerged at Corbis between 2006 and 2008 and has continued to evolve in the wider Lean software development community in the years since

Chapter 2 What is the Kanban Method?

In spring of 2005, I had the good fortune to take a vacation in Tokyo, Japan at the beginning of April during the time when the blossoms appear on the cherry trees in the parks and gardens. To enjoy this spectacle I made my second ever visit to the East Gardens at the Imperial Palace in downtown Tokyo. It was here that I had the revelation that kanban wasn't only for manufacturing.

The Imperial Palace Gardens

On Saturday April 9th 2005, I entered the park via the North entrance crossing the bridge across the moat close to the Takebashi subway station. It was spring and the cherry blossom was out. Many Tokyoites were taking the opportunity on a sunny Saturday morning to enjoy the tranquility of the park and the beauty of the *sakura (cherry blossom)*.

The practice of taking a picnic under the cherry trees while the blossoms fall around you is known as *hanami (flower party).* It's an ancient tradition in Japan. It's a chance to reflect on the beauty, fragility and shortness of life. The brief life of the cherry blossom is a metaphor for our own life, and our short, beautiful and fragile existence amongst the vastness of the universe.

The cherry blossom provide contrast against the grey buildings of downtown Tokyo, its hustle and bustle, the throbbing crowds of busy people, and the noise of traffic, the cherry blossom provided stark contrast. The gardens were an oasis of tranquility and beauty at the heart of the concrete jungle. As I crossed the bridge with my family, an elderly Japanese gentleman, with a satchel over his shoulder approached us. Reaching in to his bag he produced a handful of plastic cards. He offered one to each of us, pausing briefly to decide whether my 3 month old daughter strapped to my chest required a card. He decided she did, and handed me two cards. He said nothing and as my Japanese is limited I offered him no conversation. We walked on into the gardens to look for a spot to enjoy our family picnic.

Two hours later after a pleasant morning in the sunshine, we packed up the picnic blanket and the refuse from our snacks and headed towards the exit. This time we head to the East Gate at Otemachi. As we approached the exit, we joined a line of people formed in front of a small kiosk. As the line shuffled forward I saw people returning their plastic entrance cards. I fished around in my pocket and retrieved both of the cards I'd been given. Approaching the kiosk I saw a neatly uniformed Japanese lady on the other side. Between us was a glass screen with a

semi-circular hole cut in it at counter level, very similar to any admission booth at a cinema or fun park. I slid my plastic cards across the counter top through the hole in the glass. The lady took them in her white gloved hands and stacked them in a rack with others. She bowed her head slightly and thanked me with a smile. No money changed hands. No explanation was given for why I'd been carrying around 2 white plastic admission cards since entering the park 2 hours earlier.

What was going on with these admission tickets? Why bother to issue a ticket if no money was to be collected? My first inclination was that it must be a security scheme. By counting all the returned cards, the authorities would insure that no stray visitors had been lost inside the grounds when they closed the park in the late afternoon. However, I quickly reflected on this and realized it would be a very poor security system. Who was to say that I'd been issued two cards rather than just one? Did my 3 month old count as luggage or a visitor? There seemed to be too much variability in the system. Too many opportunities for errors! If it was a security scheme then surely it would fail and produce false positives every day. [As a brief aside, such as system can not readily produce false negatives as it would require the manufacture of additional admission tickets. This is a useful common attribute of kanban systems.] Meanwhile, the troops would be out scurrying around the bushes every evening looking for lost tourists. No, it had to be something else. I realized then that the Imperial Palace Gardens was using a kanban system!

This epiphany was hugely enlightening and gave me permission to think beyond manufacturing with respect to kanban systems. It seemed likely that kanban tokens were useful in all sorts of management situations.

What is a Kanban system?

A number of kanban (or cards) equivalent to the (agreed) capacity of a system are placed in circulation. One card attaches to one piece of work. Each card acts as a signaling mechanism. A new piece of work can only be started when a free card is available. This free card is attached to a piece of work and follows it as it flows through the system. When there are no more free cards, no additional new work can be started. Any work waiting to start must queue until a card becomes available. When some work is completed, a card is detached and recycled. With a card now free a new piece of work queuing can be started.

This mechanism is known as a *pull system* because new work is pulled into the system when there is capacity to handle it, rather than being pushed into the system based on demand. A *pull system* cannot be overloaded providing the capacity as determined by the number of signal cards in circulation has been set appropriately.

In the case of the Imperial Palace Gardens, the gardens themselves are the system, the work-in-progress is the visitors to the gardens and the capacity is limited by the admission cards that are used to signal the entrance for newly arriving visitors. Admission is actually only possible when there are free admission tickets to hand out. On a normal day this is never an issue. However, on busy days such as public holidays and a Saturday during the cherry blossom season, the park is popular. When all the admission tickets are given out, new visitors must queue outside across the bridge waiting for cards to be recycled from visitors leaving. The kanban system provides a simple, cheap and easily implemented method for controlling the crowd and limiting the people inside the park. This allows the park wardens to maintain the gardens in good condition and avoid damage caused by too much foot traffic and overcrowding.

Kanban Applied in Software Development

In software development, we are using a virtual kanban system to limit work-in-progress. While "kanban" means "signal card" and there are cards used in most Kanban implementations in software development, these cards do not actually function as signals to pull more work. Instead they represent work items. Hence the term "virtual" because there is no physical signal card. The signal to pull new work is inferred from the visual quantity of work-in-progress subtracted from some indicator of the limit (or capacity). Some practitioners have implemented physical kanban using techniques such as sticky clips. Often the "signal" is generated from a software work tracking system. Sometimes, it's generated visually from observation of work orders on a physical task board. Examples of all the mechanics of kanban systems applied to IT work are given in Chapter 6.

Card walls have become a popular visual control mechanism in agile software development as shown in figure 2. Using either a cork notice board with index cards pinned to a board, or a white board with sticky notes to track a work-in-progress has become common place. It's worth observing at this early stage that despite some commentary in the community to the contrary, these card walls are not inherently kanban systems. They are merely visual control systems. They allow teams to visually observe work-in-progress and for the teams to self-organize, assign their own tasks and move work from a backlog to complete without direction from a project or line manager. However, if there is no explicit limit to work-in-progress and no signaling to pull new work through the system then it is not a kanban system. This is more fully explained in chapter 7.

[Figure 2. Insert photograph of a card wall]

Why use a kanban system?

As should become evident from the subsequent chapters, we use a kanban system to balance limit the capacity of work-in-progress on a team and to balance the demand the team against their throughput of delivered work. By doing this we can achieve a sustainable pace of development so that all individuals can achieve a personal work versus personal life balance. As you will see kanban also quickly flushes out issues that impair performance and challenges the team to focus on resolving them in order to maintain a steady flow of work. By providing visibility onto quality and process problems it makes obvious the impact of defects, bottlenecks, variability and waste on flow and throughput. The simple act of limiting work-in-progress with kanban encourages higher quality and greater performance. The combination of improved flow and better quality helps us to shorten cycle times and improve our predictability and due date performance. By establishing a regular release cadence and delivering against it consistently, kanban helps us to build trust with customers and across our value stream with other departments, suppliers and dependent downstream partners.

By doing all of this kanban assists in the cultural evolution of organizations. By exposing problems, focusing the organization on resolving them and eliminating their effects in future, kanban facilitates the emergence of a highly collaborative, high trust, highly empowered continuously improving organization.

Kanban has been shown to improve customer satisfaction through regular, dependable, high quality releases of valuable software. It has been shown to improve productivity, quality and cycle times. It has been shown to be a pivotal mechanism evolving a more agile organization through evolutionary cultural change.

The remainder of this book is dedicated to helping you to understand the use of kanban in software development and, to teaching you how to implement a kanban system to achieve these benefits with your team.

A Definition of the Kanban Method

Now, to document the approach in a book, I find it necessary to make a formal definition of the Kanban Method. As Kanban has been both emerging from practice and evolving over the last four years, I have never thought to write down a formal definition - until now. I worry that such a definition will become static and people will become dogmatic in their thinking, citing this definition as the one true source for kanban process goodness. I hope that while attempting to define Kanban, I can also encourage you to open your mind to the idea that our learning will evolve and rather than cite this section of this book, you will seek out a contemporary definition on the web at the Limited WIP Society, http://www.limitedwipsociety.org/

Kanban

There are 5 core properties to a Kanban implementation

1. Limit Work-in-Progress
2. Visualize Workflow
3. Measure & Optimize Flow
4. Make Process Policies Explicit
5. Manage Quantitatively

There at least 5 emergent properties we might expect in a Kanban implementation

1. Prioritize Work by Cost of Delay
2. Optimize Value with Classes of Service
3. Spread Risk with Capacity Allocation
4. Encourage Process Innovation
5. Use Models* to Recognize Improvement Opportunities

*Common models in use with Kanban include the Theory of Constraints, Systems Thinking, an understanding of variability through the teachings of W. Edwards Deming and the concept of "muda" (waste) from the Toyota Production System. The models used with Kanban are continually evolving and ideas from other fields such as sociology, psychology, and risk management are appearing in some implementations.

Kanban as a Permission Giver

Kanban is not a software development lifecycle methodology or an approach to project management. It requires that some process is already being followed so that Kanban can be applied to incrementally change the underlying process.

This evolutionary approach, promoting incremental change, has proved to be controversial in the agile software development community. It is controversial because it suggests that teams should not adopt a defined method or process template. An industry services and tools have developed around a small set of practices defined in two popular agile development methods. Now with Kanban, individuals and teams might be empowered to evolve their own unique process solutions that obviate the need for such services and require a new set of tools. Indeed, Kanban has encouraged a new wave of insurgent tool vendors eager to displace existing agile project management tools with something more visual and programmable that is readily tailored to a specific workflow.

In the early days of agile software development, the leaders in the community often didn't understand why their methods worked. We talked about "ecosystems"[x] and advised that implementers had to follow all of the practices or the solution was likely to fail. In recent years there has been a negative trend that extends this thinking. A few businesses have published Agile Maturity Models that seek to assess practice adoption. The Scrum community has a practice-based test often referred to as the "Nokia Test[xi]." These practice-based assessments are designed to drive conformity and are in denial of the need for context based adaptation. Kanban is giving the market permission to ignore these practice based appraisal schemes. It's actively encouraging diversity.

In 2007, several people visited my offices at Corbis to see Kanban in action. The common question from any visitor associated with the agile software development community could be paraphrased as "David, we have been around your building and we've seen seven kanban boards. Each one is different! Each team is following a different process! How can you possibly cope with this complexity?" My answer was always a dismissive "Of course! Each team's situation is different. They evolve their process to fit their context."

As more people have tried Kanban and they have realized that it helps address problems they'd been encountering with change management in their organizations. Kanban enabled their team, project or organization to exhibit better agility. We've come to recognize that Kanban is giving permission in the market to create a tailored process optimized to a specific context. Kanban is giving people permission to think for themselves. It is giving people permission to be different: different from the team across the floor; on the next floor; in the next building; and at a neighboring firm. It's giving people permission to deviate from the textbook. Best of all Kanban is providing the tools to enable an explanation and justification of why being different is better and why a choice to be different is the right choice in that context.

YES WE KANBAN
limitedwipsociety.org

To emphasize this choice, I designed a T-shirt for the Limited WIP Society, inspired by a Barack Obama campaign poster, and featuring the face of Taiichi Ohno, the creator of the kanban system at Toyota. The slogan "Yes We Kanban" is designed to emphasize that you have permission. You have permission to try Kanban. You have permission to modify your process. You have permission to be different. Your situation is unique and you deserve to develop a unique process definition tailored and optimized to your domain, your value-stream, the risks that you manage, the skills of your team, and the demands of your customers.

Takeaways

- ❖ Kanban systems can be used in any situation where there is a desire to limit a quantity of things inside a system
- ❖ The Imperial Palace Gardens in Tokyo uses a kanban system to control the size of the crowd inside the park
- ❖ The quantity of "kanban" signal cards in circulation limits work-in-progress
- ❖ New work is *pulled* into the process by a returning signal card on completion of a current work order or task
- ❖ In IT work we are (generally) using a virtual kanban system as no actual physical card is passed around to define the limit to work-in-progress
- ❖ Card walls common in agile software development are not kanban systems
- ❖ Kanban systems create a positive tension in the workplace that forces discussion of problems
- ❖ The Kanban Method (or capital K "Kanban") utilizes a kanban system to be a catalyst of change
- ❖ Kanban requires that process policies are defined explicitly
- ❖ Kanban uses tools from various fields of knowledge to encourage analysis of problems and discovery of solutions
- ❖ Kanban enables incremental process improvement through repeated discovery of issues affecting process performance
- ❖ A contemporary definition of the Kanban Method can be found online at the Limited WIP Society community website, http://www.limitedwipsociety.org/
- ❖ Kanban is acting as a permission giver in the software development profession, encouraging teams to devise context specific process solutions rather than dogmatically following a software development lifecycle process definition or template

Section Two
Benefits of Kanban

Chapter 3 A Recipe for Success

Over the 10 years or so I've been involved in managing and coaching software development teams, I've been challenged with a problem. It's the problem of what actions you should take as a manager when you inherit an existing team, especially if that team is not working in an agile fashion, has a broad spread of ability and is perhaps completely dysfunctional. Typically, I've been given management positions as a change agent. So I've been challenged with how to create positive change and make progress quickly within two or three months.

As a manager in larger organizations, I've never been given the chance to hire my own team. I've always been asked to adopt an existing team and with minimal personnel changes create a revolution in the performance of the organization. I think this situation is much more common than the situation where you get to hire a whole new team.

I gradually evolved an approach to managing this change. It's an approach based on experience and learning from failure. The failures involved trying to impose a process and workflow using positional power. Management edict tended to fail. When I asked teams to change their behavior and use an agile method such as Feature Driven Development, I met resistance. I countered with the suggestion that no one should fear for I would provide them with the training and coaching required. I got acquiescence at best not true deep institutionalization of the change. Asking people to change their behavior creates fear and lowers self-esteem as existing skills are clearly no longer valued.

I developed what I've come to call my Recipe for Success to address these issues. The Recipe for Success represents the guidelines for a new manager adopting an existing team. Following the recipe will enable improvement quickly with low levels of resistance from the team. I want to acknowledge the direct influence of Donald Reinertsen who gave me the first two steps in the recipe and the indirect influence of Eli Goldratt whose writings on the Theory of Constraints and its Five Focusing Steps greatly influenced the remainder of the recipe.

The five steps in the recipe are:

- Focus on Quality
- Reduce Work-in-Progress, Deliver Often
- Balance Demand against Throughput
- Prioritize
- Reduce Variability and Improve the Process

Implementing the Recipe

The recipe is delivered in order of execution for a technical function manager. Focus on Quality is first as it is under the sole control and influence of a manager such as a development or test manager or their supervisor with a title like Director of Engineering. Working down the list there is gradually less control and more collaboration required with other downstream and upstream organizations until we get to Prioritize. Prioritization is rightly the job of the business and not the technology organization. Hence, it should not be within the remit of a technical manager at all. Unfortunately, it is all too common for the business to abdicate responsibility leaving a technical manager to make prioritizations then blaming that same manager for making bad choices. Unfortunately Kanban doesn't provide an easy fix if you are suffering under this type of dysfunction.

Focusing on quality is easiest because it is a technical discipline that can be directed by the function manager. The other steps are harder because they depend on agreement and collaboration from other teams. They require skills in articulation, negotiation, psychology, sociology and emotional intelligence. Building consensus around the need to balance demand against throughput is non-trivial. Solving s dysfunction in roles and responsibilities such as I just described requires even greater diplomatic and negotiation skills. It makes sense then to go after things which are directly under your control and you know will have a positive effect on performance of both your team and your business.

The harder things can be enabled by developing an increased level of trust with other teams. Building and demonstrating high quality code with few defects improves trust. Releasing high quality code with regularity builds yet more trust. As the level of trust increases, the manager gains more political capital. This enables a move to the next step in the recipe. Ultimately your team will gain sufficient respect that you are able to influence product owners, your marketing team and business sponsors to change their behavior and collaborate to prioritize the most valuable work for development.

Reduce Variability and Improve the Process appears last on the list because it is difficult to do. Reducing variability requires people to change their behavior and adopt new practices and improve their capabilities. Because this is hard it should not be undertaken until a team is already performing at a more mature and greatly improved level. The first four steps in the recipe will have a very significant impact. They will deliver success for the new manager. However, to truly create a culture of innovation and continuous improvement it will be necessary to address variability in the process and its inputs. So reducing variability and improving the process is the step for extra credit: the step that delineates the truly great technical leaders from the merely competent managers.

Focus on Quality

The Agile Manifesto doesn’t say anything about quality though the Principles Behind the Manifesto[xii] do talk about craftsmanship and there is an implied focus on quality. So if quality doesn’t appear in the Manifesto why is it first on my recipe for success? Put simply, excessive defects are the biggest waste in software development. The numbers on this are quite staggering and show several orders of magnitude variation. Capers Jones[xiii] reports that in 2000, during the dot com bubble, he measured software quality on North American teams ranging from 6 defects per function point down to less than 3 per 100 function points, a range of 200 to 1. The midpoint is in the range of one defect per 0.6 to 1.0 function points. This implies that it is common for teams to spend more than 90% of their effort fixing defects. As direct evidence, late in 2007, Aaron Sanders, an early proponent of Kanban reported a team he was working with was spending 90% of available capacity on defect fixes.

Encouraging high initial quality will have a big impact on the productivity and throughput of teams with high defect rates. A 2 to 4 times throughput improvement is reasonable. With truly bad teams a 10 times improvement may be possible by focusing on quality.

Improving software quality is a well understood problem.

Both agile development and traditional approaches to quality work. They should be used in combination. Professional testers should do testing. Testers finds defects and prevents them escaping into production code. Asking developers to write unit tests and automate those tests to provide automated regression testing also has a dramatic effect. There seems to be a psychological advantage in asking developers to write the tests first. So called Test Driven Development (TDD) does seem to provide the advantage that test coverage is more complete. It is worth pointing out that, well-disciplined teams I’ve managed who wrote tests after functional coding demonstrated industry leading quality. Nevertheless, it appears evident that for the average team insisting on writing tests first before functional coding improves quality.

Code inspections improve quality! Whether it is pair programming, peer review, code walkthroughs or full Fagan inspections, code inspections work. They help to improve both external quality and internal code quality. Code inspections are best done often and in small batches. I encourage teams to inspect code every day for at least 30 minutes.

Collaborative analysis and design improve quality! When teams are asked to work together to analyze problems and design solutions the quality output is higher. I encourage teams to hold collaborative team analysis and design modeling sessions.

Design modeling should be done in small batches every day. Scott Ambler has called this Agile Modeling[xiv].

The use of design patterns improves quality! Design patterns capture known solutions to known problems. Design patterns ensure that more information is available earlier in the lifecycle and design defects are eliminated.

The use of modern development tools improves quality! Many modern tools include functions to perform static and dynamic code analysis. This should be switched on and tuned for each project. These analysis tools can prevent programmers from making elementary mistakes and introducing well understood problems such as security flaws.

More exotic modern development tools such as Software Product Lines (or Software Factories) and Domain Specific Languages reduce defects. Software factories can be used to encapsulate design patterns as code fragments reducing the defect insertion potential from entering code. They can also be used to automate away repetitive coding tasks again reducing the defect insertion potential of entering code. The use of software factories also reduces the demands on code inspections as factory code doesn't need to be re-inspected. It has a known quality.

Some of these latter suggestions really fall in the category of reducing variability in the process. The use of software factories and perhaps even design patterns is asking developers to change their behavior. The big bang for the buck comes from using professional testers, writing tests first, automating regression testing and code inspections. And one more thing…

Reducing the quantity of design-in-progress boosts software quality!

Reduce Work-in-Progress, Release Often

In 2004 I was with two teams at Motorola. Both teams were developing network server-side code for cell phone applications. One team was working on the over-the-air (OTA) download server for ringtones, games and other applications and data. The other team was developing the server for over-the-air device management (OTA DM). Both teams were using the Feature Driven Development (FDD) methodology. Both teams were approximately the same size about 8 developers, one architect, up to 5 testers, and a project manager. The team was responsible for its own analysis and design working with the marketing team. In addition there was a user experience design and user documentation (technical writing) team who provided services to both project teams.

Figure 9 shows a cumulative flow diagram for the project work by the OTA download team. A cumulative flow diagram is an area graph that depicts the quantity of work in a given state. The states shown on this chart are: "inventory" which means backlog or queuing to be started; "started" which implies the requirements for a feature have been explained to the developers; "designed" which means specifically that a UML sequence diagram had been developed for the feature; "coded" which means the methods on the sequence diagram have been implemented; "complete" which means that all unit tests are passing and that the code has been peer reviewed and the team lead developers has accepted the code and promoted it for testing.

The first line on the graph shows the number of features in scope for the project. The scope arrived in two batches from the business owners. The second line shows the number of features started. The third line shows the number designed. The fourth shows the number developed and the fifth line shows the number completed and ready for testing.

The height between the 2nd line and 5th line on any given day shows the quantity of work-in-progress, while the horizontal distance between the 2nd line and the 5th line shows the average cycle time from starting a feature on that day until it was finished. It is important to note that the horizontal distance is an average cycle time and not a specific cycle time for a specific feature. The cumulative flow diagram does not track specific features. The 55th feature started might be the 30th feature completed. There is no association between a line on the y-axis and a specific feature in the backlog.

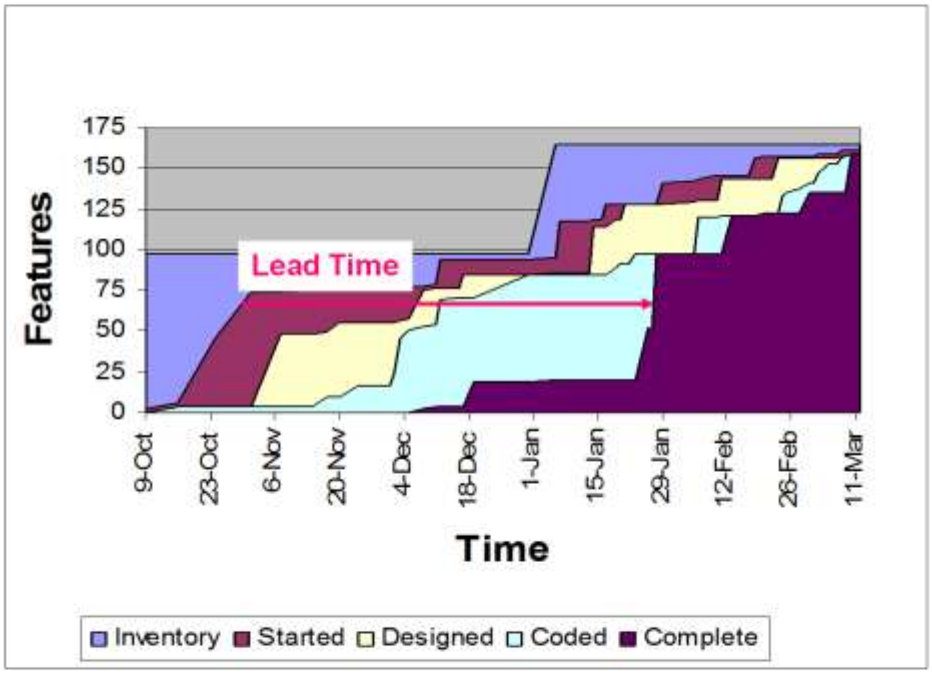


Figure 9. Cumulative flow diagram OTA Download team fall 2003, winter 2004

The OTA download server team lacked the discipline or perhaps lacked buy-in to use the FDD method. They did not work collaboratively as FDD demands. They handed out large batches of features to individual developers. Typically they had 10 features per developer in progress at any given time. The OTA DM team followed the method to the textbook definition. They were working very collaboratively. They were developing unit tests for 100% of the functionality. And most importantly they were working on small batches of features at a time, typically 5 to 10 features in progress for the whole team at any given time. As a benchmark, a feature in FDD appears to represent about 1.6 to 2.0 function points of code.

The OTA download server team had an average cycle time per unit from starting a feature to completing it for handoff from the team in Seattle, Washington to integration test in Champagne, Illinois of around 3 months, as shown in Figure 9. The OTA DM team had an average cycle time per unit in the range of 5 to 10 days, illustrated in Figure 10. The difference in initial quality measured as escaped defects leaking into system or integration test was greater than 30 times between the two teams. The OTA DM team produced industry leading initial quality of 2 or 3 defects per 100 features, while the OTA download server team produced industry average performance of around 2 defects per feature.

We can see from examining the charts that the quantity of work-in-progress is directly related to the cycle time. Figure 10 clearly shows that average cycle time falls as the quantity of work-in-progress falls. At peak the average cycle time is 12 days. Later in project with less work-in-progress the average cycle time falls to as little as 4 days.

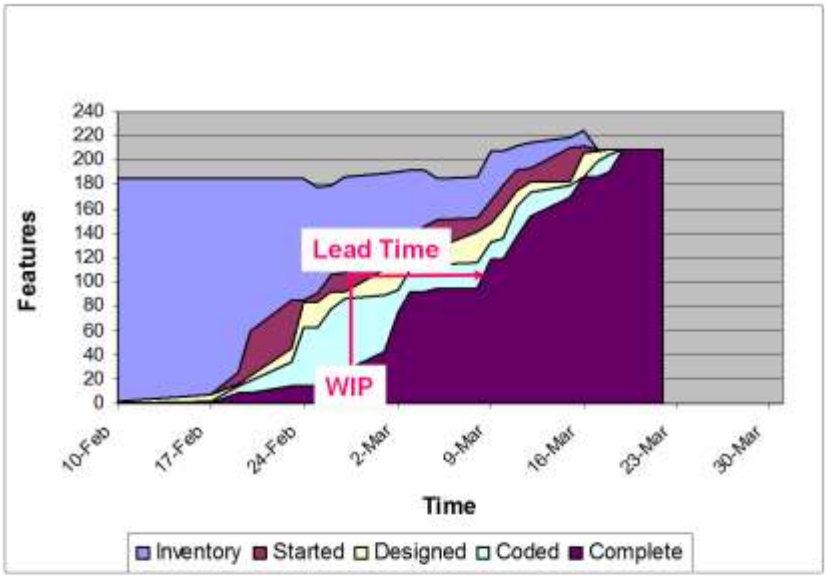


Figure 10. Cumulative Flow Diagram from OTA DM team winter 2004

There is causation between quantity of work-in-progress and average cycle time and the relationship is linear. In the manufacturing industry this relationship is known as Little's Law. The evidence from these two teams at Motorola suggests that there is a correlation between increased cycle time and poorer quality. It seems that longer cycle times are associated with significantly poorer quality. In fact an approximately 6.5x increase in average cycle time resulted in a greater than 30x increase in initial defects. Longer average cycle times are a result of greater work-in-progress. Hence, the management leverage point for improving quality is to reduce the quantity of work-in-progress. Since uncovering this evidence I have managed work-in-progress ("WIP") to control quality and have become convinced of the relationship between the quantity of WIP and initial quality. However, at the time of writing there is no scientific evidence to back up this empirically observed result.

Reducing work-in-progress or shortening the length of an iteration will have a significant impact on initial quality. It appears that the relationship between quantity of work-in-progress and initial quality is a non-linear relationship, that is to say, that defects will rise disproportionately to the increased quantity of WIP. Therefore, it makes sense that 2 week iterations are better than 4 week iterations and that 1 week iterations are better than 2 weeks. Shorter iterations will drive higher quality.

However, following the logic of the evidence presented it makes even more sense to simply limit WIP using a kanban system. If we know that managing WIP will improve quality, why not introduce explicit policy to limit WIP thus freeing managers to focus on other activities?

Because of the close interaction between work-in-progress and quality, it follows that step 2 of the recipe should be implemented together or soon after step 1.

I intervened with the OTA download team during the week of Christmas 2003 and suggested to the team lead that there was too much work-in-progress, the cycle time was too long and that very few things were actually being completed. I shared my concern that quality would be poor as a result of this. He took my advice to heart and in January 2004 made some changes to how the team was working. The result is reduced WIP in 2004 and a demonstrably shorter cycle time. This change, however, came too late to prevent the team from creating a large number of defects.

While the diagram suggests that the project completed around mid-March 2004, the team continued working on the software until mid-July that year. Half of the OTA DM team was pulled off their project to help fix the defects. In July 2004 the general manager of the business unit declared the product complete despite continued concerns about quality. The product was handed off to the field engineering team. During deployment, as many as 50% of customers cancelled delivery, due to quality concerns. While the field engineering team maintained good personal relationships with the engineering team for this product they had a low level professional respect and a lack of trust in the ability of team. Their opinion was that the product was of poor quality and the team was incapable of delivering anything better.

Ironically, if you had walked into that building in Seattle's SODO neighborhood, and asked the developers, "Who's the smartest guy around here?" They would have pointed to someone from the OTA download team. If you'd asked them, "Who has the most experience? ", again the same answer. If you'd looked at the resumes you would have seen that the average experience on that OTA download team exceeded that of the OTA DM team by 3 years. On paper, everything suggested that the OTA download team was better. And to this day some of these individuals believe that they were better despite all the quantitative evidence otherwise. I know from my managerial and coaching experience with this team that some of the OTA DM team members suffered from low professional self-esteem and worried that they weren't as talented as some of the truly smart folks on the other team. However, the OTA DM produced 5.5 times greater productivity with more than 30 times better initial quality. The right process, with good discipline, strong management and good leadership all made a difference. What this example really demonstrates is that you don't need the best people to produce world class results. One of the core beliefs in the agile community, what I refer to as "craftsmanship snobbery," suggests that all you need for success in agile development is a small team of really good people. However, in this case, a team with a spread of talent was able to produce world class results.

Reducing WIP shortens cycle time. Shorter cycle times mean that it is possible to release working code more often. More frequent releases build trust with external teams, particularly the marketing team or business sponsors. Trust is a hard thing to define. Sociologists call it social capital. What they've learned is that trust is event driven and small gestures or events made often enhance trust more than larger gestures made occasionally.

When I teach this in classes, I like to ask women in the class, what they think after they go on a first date with a guy. I suggest that they had a nice time and then he doesn't call them for two weeks. He then turns up on their doorstep with a bunch of flowers and an apology. I ask them to compare this to a guy who takes the time to type a text message on his way home that evening saying, "I had a great time tonight. I really want to see you again. Call you tomorrow?" and then follows up by actually calling the next day. Small gestures often cost nothing but build more trust than large expensive gestures provided occasionally.

And so it is with software development. Small, frequent, releases of high quality, builds more trust with partner teams than larger release less often.

Small releases show that the software development team can deliver and is committed to providing value. They build trust with the marketing team or business sponsors. High quality in the released code builds trust with downstream partners such as operations, technical support, field engineering, and sales.

It's quite easy to speculate why small batches of code improve quality. Complexity in knowledge work problems grows non-linearly with the quantity of work-in-progress. Meanwhile our human brains struggle to cope with all this complexity. So much of knowledge transfer and information discovery in software development is tacit in nature and created during collaborative working sessions, face-to-face. The information is verbal and visual but in a casual format like a sketch on a whiteboard. Our minds have a limited capacity to store all this tacit knowledge and it degrades while we store it. We fail to recall precise details and mistakes get made. If a team is collocated and always available to each other then this loss of memory can be corrected through repeat discussion or tapping the shared memory of a set of people. So agile teams collocated in a shared workspace are more likely to retain tacit knowledge for longer. Regardless of this, tacit knowledge depreciates with time. So, shorter cycle times are essential for tacit knowledge processes. We know that reducing work-in-progress is directly related to reducing cycle times. Hence, we can infer that there will be lower tacit knowledge depreciation when we have less work-in-progress and as a result, higher quality.

So in summary, reducing work-in-progress improves quality and enables more frequent releases. More frequent releases of higher quality code improve trust with external teams.

Balance Demand against Throughput

Balancing demand against throughput implies that we will control the rate we accept new requirements into our software development pipe to the rate at which we can deliver working code. When we do this, we are effectively fixing our work-in-progress to a given size. As work is delivered, we will pull new work (or requirements) from the people creating demand. So any discussion about prioritization and commitment to new work can only happen in the context of delivering some existing work.

The effect of this change is quite profound. The throughput of your process will be constrained by a bottleneck. It's very likely you do not know where that bottleneck is. In fact, if you speak to everyone in the value stream, they will probably claim to be completely overloaded. However, once you balance demand against throughput and limit the work-in-progress within your value stream, magic will happen. Only the bottleneck resources will remain fully loaded. Very quickly other workers in the value stream will find they have slack capacity. Meanwhile, those working in the bottleneck will be busy but not swamped. For the first time, perhaps in years, the team will no longer be overloaded and many people will be experiencing something very rare in their career, the feeling of having time on their hands. The stress will be lifted off the organization and people will be able to focus on doing their jobs with precision and quality. They'll be able to take a pride in their work and will enjoy the experience all the more. Those with time on their hands will start to apply that time to improving their own circumstances. They may tidy up their work space, or take some training. They will start to apply themselves to bettering their skills, their tools and how they interact with others upstream and downstream. As time passes and one small improvement leads to another, the team will be seen to be continuously improving. The culture will have changed. The slack capacity created by the act of limiting work-in-progress and only pulling new work as capacity is available will enable improvement no one thought was possible.

You need slack to enable continuous improvement. You need to balance demand against throughput and limit the quantity of work-in-progress to enable slack.

Intuitively, people believe they have to eliminate slack. So after limiting work-in-progress by balancing demand against throughput, the tendency is to "balance the line" by adjusting resources so that everyone is efficiently fully utilized. While this may look efficient and satisfy typical 20th Century management accounting practices, it will impede the creation of an improvement culture. You need slack to enable continuous improvement. In order to have slack, you must have an unbalanced value stream with a bottleneck resource. Optimizing for utilization is not desirable.

Prioritize

If the first three steps in the recipe have been implemented things will be running smoothly. High quality code should be arriving frequently. Development lead times should be relatively short as work-in-progress is limited. New work should only be pulled in development as capacity is freed up on completion of existing work. At this point, management attention can turn to optimizing the value delivered rather than merely the quantity of code delivered. There is little point in paying attention to prioritization when there is no predictability in delivery. Why waste effort trying to order the input when there is no dependability in the order of delivery? Until this is fixed, it is a better use of management time to focus on improving the ability to deliver and the predictability of delivery. You should turn your thoughts to ordering the priority of the input when you know you can actually deliver things in approximately the order they were requested.

Prioritization should not be controlled by the engineering organization and hence is not under the control of the engineering management. Improving prioritization requires the product owner, business sponsor, or marketing department to change their behavior. At best the engineering management can only seek to influence how prioritization is done.

In order to have political and social capital to influence change, a level of trust must have been established. Without the capability to deliver high quality code regularly, there can be no trust and hence little possibility to influence prioritization and optimize the value being delivered from the software team.

Recently, it's become popular in the Agile community to talk about business value optimization and how the production rate of working code (called the "velocity" of software development) is not an important metric. This is because business value delivered is the true measure of success. While this may ultimately be true, I feel it is important not to lose sight of the capability maturity ladder that a team must climb. Most organizations are incapable of measuring and reporting business value delivered. They must first build capability in basic skills before they try greater challenges.

This is how I feel a team should mature. First learn to build high quality code. Then reduce the work-in-progress, shorten cycle times and release often. Then balance demand against throughput, limit work-in-progress and create slack to free up bandwidth to enable improvements. Then with a smoothly functioning and optimizing software development capability, improve prioritization to optimize value delivery. Hoping for business value optimization is wishful thinking. Take actions to get to this level of maturity incrementally – follow the Recipe for Success.

Reduce Variability and Improve the Process

Both the affects of variability and how to reduce it within a process are advanced topics. Reducing variability in software development requires knowledge workers to change the way they are working, to learn new techniques and to change their personal behavior. All of this is hard. It is therefore not for beginners or for immature organizations.

Variability results in more work-in-progress, and longer cycle times. This is explained more fully in Chapter 19. Variability creates a greater need for slack in non-bottleneck resources in order to cope with the ebb and flow of work as its effects manifest themselves on the flow of work through the value stream. A full understanding of why this is true requires some background in statistical process control and queuing theory. These topics are beyond the scope of this book. Personally I like the work of Donald Wheeler and Donald Reinertsen on variability and queuing.

For now, it will suffice that you can take it on trust that variability in size of requirements, and effort expended on analysis, design, coding, testing, build integration and delivery will adversely affect the throughput of your process and the costs of running your software development value stream.

There are three main types of opportunity for improvement. Variability in processes (work item definition, analysis techniques, design techniques, coding techniques, testing techniques, communication techniques) represent the first category. Bottlenecks in the shape of capacity constrained resources and non-instantly available resources are the second category. And waste, in the form of defects and rework, transaction costs and coordination costs, is the third category. Opportunities for improvement, how to recognize them, and how to implement them are fully discussed in chapters 16 through 20.

Recipe for Success and Kanban

Kanban enables all 5 steps in the Recipe for Success. Kanban delivers on the Recipe for Success and the Recipe for Success delivers on its promise for the manager implementing it. In turn, the Recipe for Success provides a basis for why Kanban is such a valuable technique.

Takeaways

- Kanban delivers all aspects of the Recipe for Success
- The Recipe for Success explains why Kanban has value
- Poor quality can represent the largest waste in software development
- Reducing work-in-progress improves quality
- Improved quality improves trust with downstream partners such as operations
- Releasing frequently improves trust with upstream partners such as marketing
- Demand can be balanced against throughput with a pull system
- Pull systems expose the bottleneck and create slack in non-bottlenecks
- Good quality prioritization maximizes the value delivered by a well functioning software development value chain
- Prioritization is of little value without good initial quality and predictability of delivery
- Making changes to reduce variability requires slack
- Reduced variability reduces the need for slack
- Reducing variability enables a resource balancing (and potentially reduced headcount)
- Reducing variability reduces resource requirements
- Reducing variability allows reduced kanban tokens, less WIP, and results in reduced average cycle time
- Slack enables improvement opportunities
- Process improvement leads to great productivity and greater predictability

Chapter 4 From Worst to Best in Five Quarters

In October 2004, Dragos Dumitriu was a program manager at Microsoft. He had recently taken charge of a department that had a reputation as the worst in Microsoft's IT division. The photo shows the team pictured in Hyderabad, India in late 2004 with Dragos shown just left of center.

The title "Program Manager" at Microsoft is more readily interpreted elsewhere as project manager but unusually includes responsibility for analysis and architecture. A program manager is assigned to some initiative, project or product and has responsibility for a feature or set of features. A program manager will recruit resources from functions such as development and test in order to complete the work. In Dragos' case he was responsible for the software maintenance for the XIT business unit. This team, based at a CMMI model level 5 rated vendor, developed minor upgrades and fixed production bugs for about 80 cross functional IT applications used by Microsoft staff throughout the World. Dragos was based in Redmond, Washington. At this time, I was also working for Microsoft on the corporate campus in Redmond.

Dragos had volunteered to take charge of a team that had the worst reputation for customer service within Microsoft. His job as change agent, determined to fix the long lead times and poor expectation setting, was hindered by the political climate. Several of his predecessors in this position were still colleagues working on other projects within the same business unit and worried that were he to improve the performance, it would make them look bad in comparison.

The programmers and testers working for the vendor were following the Software Engineering Institute's Personal Software Process/Team Software Process (PSP/TSP) methodology. This was mandated contractually by Microsoft. Jon De Vaan, who at that time reported directly to Bill Gates, was a big fan of Watts Humphrey of the Software Engineering Institute. As head of Engineering Excellence at Microsoft he was in a position to mandate processes used within the IT department and by their vendors. This meant that changing the software development lifecycle method in use was not an available option.

Dragos realized that neither the PSP/TSP method or the CMMI rating of the vendor were likely to be the root cause of their problems. In fact, the team produced pretty much what was asked of them and with very high quality. Nevertheless, they had a 5 month lead time on change requests and this along with their backlog of requests was growing uncontrollably. The perception was of a team that was badly organized and managed. As a result, senior management was not of a disposition to provide additional money to fix the problem.

So the constraints on change were political, fiscal and company policy related.

Visualize the Wokflow

I asked Dragos to sketch the workflow. He drew a simple stick man drawing describing the lifecycle of a change request and as he did so we discussed the problems. Figure 3 is a facsimile of what he drew. The PM stick figure represents Dragos.

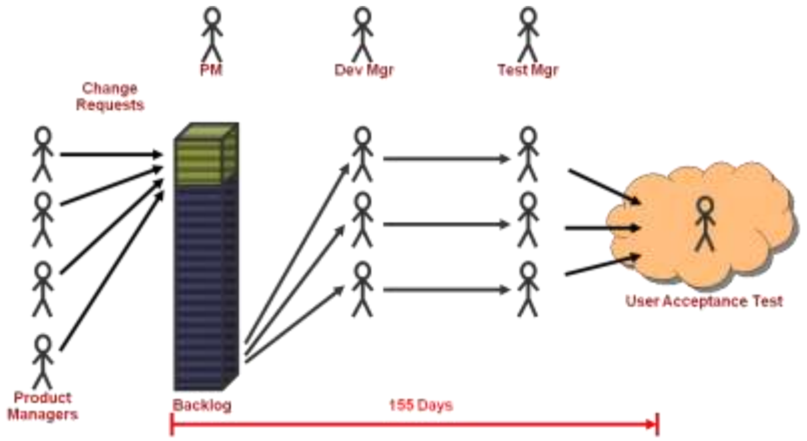


Figure 3 XIT Sustaining Engineering - Initial Workflow showing Lead Time

Requests were arriving uncontrollably. Four product managers represented and controlled budgets for a number of customers who owned applications maintained by XIT. They were adding new requests, including escaped production defects (discovered in the field). These defects were not created by the maintenance team but by the application development project teams. The application development teams were generally broken up 1 month after the release of a new system and the source code handed off to the maintenance team.

Factors Affecting Performance

When each request arrived Dragos would send it to India for an estimate. There was a policy that estimates had to be made and returned to the business owners within 48 hours. This would facilitate them making some return on investment (ROI) calculation and deciding whether to proceed with the request. Once per month, Dragos would meet with the product managers and other stakeholders and they would reprioritize the backlog and create a project plan from the requests.

At this time, the monthly throughput of requests was around 7. The backlog had 80 or more items in it and was growing. This meant that 70 or more requests were being reprioritized and rescheduled each month and that requests were taking over 4 months to process on average. This was the root cause of the dissatisfaction. These requests were small and the constant reprioritization meant that requestors were being constantly disappointed.

The requests were tracked in tool a called Product Studio. An updated version of this tool was later released publicly as Team Foundation Server Work Item Tracking. The XIT maintenance team fell into a category of organization I see often in my teaching and consulting work – they had lots of data but they were not using this data. Dragos began to mine the data and discovered that an average request involved 11 days of engineering. However, lead times of between 125 and 155 days were typical. More than 90% of the lead time was queuing or other forms of waste.

The estimates for new incoming work were consuming a lot of effort. We decided to analyze this using some guess work. Despite being referred to as "rough order of magnitude" (ROM) estimates, the customer expectation was actually for a very accurate estimate and team members had learned to take great care over the preparation of an estimate. Each one was taking about 1 day for each developer and tester. We quickly calculated that the estimation effort was consuming around 33% of capacity and on a bad month it could be as much as 40%. This capacity was allocated in preference to working on coding and testing. Estimation of new requests was also apt to randomize plans made for that month.

In addition to change requests, the team had a second type of work, known as production text changes (PTCs) that were generally graphical or textual in nature or involved modifying values in tables or XML files. These changes did not require a developer and were often made by business owners, product managers or the program manager but did require a formal test pass. So they affected the testers.

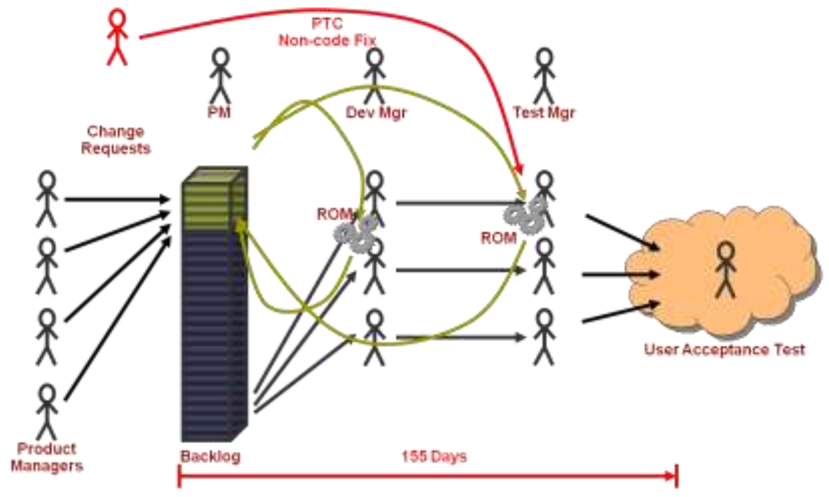


Figure 4 Workflow showing ROM Estimates & PTC input

PTCs arrived without much warning and were, by tradition, expedited over all other work or estimation effort. PTCs tended to arrive in sporadic batches. They would randomize any plans made for that month.

Make Process Policies Explicit

The team was following the required process that included many bad policy decisions made by managers at various levels. It is important to think of a process as a set of policies that govern behavior. These policies are under the control of the management. For example, the policy on use of PSP/TSP was set at the executive vice president level, one rung below Bill Gates, and this policy would be hard or impossible to change. However, many other policies such as the policy to prioritize estimates over actual coding and testing, were developed locally and were under the collaborative authority of the immediate managers. It is possible that the policies made sense at the time they were implemented but circumstances had changed and no attempt had been made to review and update the policies that governed the team's operation.

Estimation was a Waste

After some discussion with his colleagues and manager, Dragos decided to enact two initial management changes. In the first instance, he would stop estimating. He wanted to recover the capacity wasted by estimation activity and use it to develop and test software. Eliminating the randomization of the schedule caused by estimating would also improve predictability and the combination would hopefully have a great impact on customer satisfaction.

However, removing estimation was problematic. It would affect the ROI calculations and customers might be worried that bad prioritization choices were made. In addition, estimates were used to facilitate inter-departmental cost accounting and budget transfers. Estimates were also used to implement a governance policy. Only small requests were allowed through system maintenance. Larger requests, those exceeding 15 days of development or testing had to be submitted to a major project initiative and go through the formal program management office (PMO) portfolio management governance process. We will revisit these issues momentarily.

Limit Work-in-progress

The other change he decided to make was to limit work-in-progress and pull work from an input queue as current work was completed. He chose to limit WIP in development to one request per developer and use a similar rule for testers. He inserted a queue between development and test in order to receive the PTCs and to smooth the flow of work between development and test. This approach of using a buffer to smooth out variability in size and effort is discussed in Chapter 19.

Note: This is a policy choice. One change request per developer at any given time is a policy. It can be modified later. Thinking of a process as a set of policies is a key element of the Kanban Method.

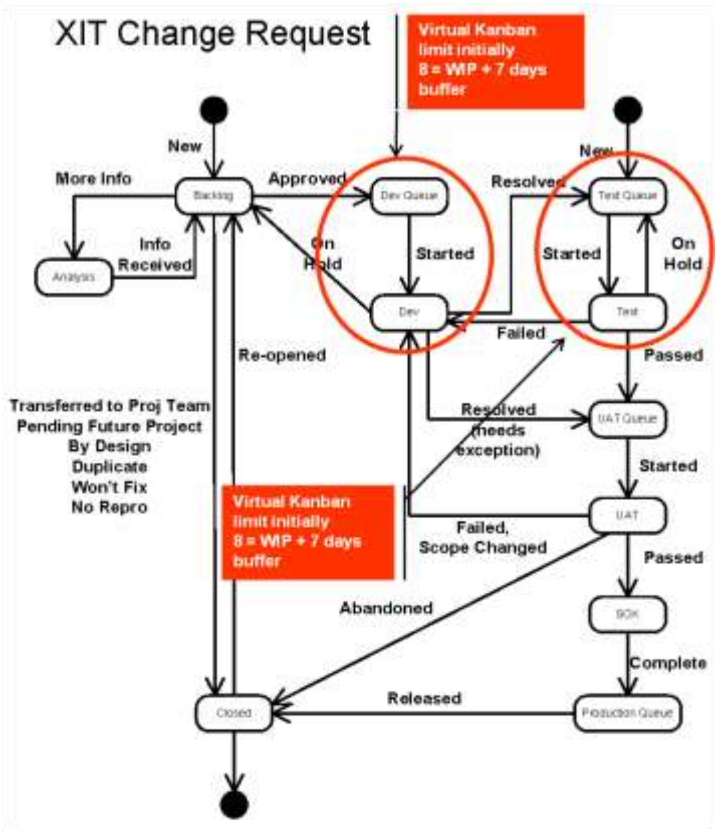


Figure 5 Statechart modeling desired workflow with WIP limits

Note: Cadence is a concept in the Kanban Method that determines the rhythm of a type of event. Prioritization, delivery, retrospectives and any recurring event can have its own cadence.

Establishing an Input Cadence

In order to facilitate this decision to limit WIP and institute a pull system, he had to think about the cadence for interacting with the product managers. He thought that a weekly meeting would be feasible as the topic of the meeting would be the simple replenishment of the input queue from the backlog. In a typical week there might be 3 slots free in the queue. So the discussion would center around the question, “Which 3 items from the backlog would you most like delivered next?”

He wanted to offer a "guaranteed" delivery time of 25 days from acceptance into the input queue. This 25 days was considerably greater than the 11 days of average engineering time required to complete the job. The statistical outliers required around 30 days but he anticipated very few of them. 25 days sounded attractive, especially compared to the existing lead time of around 140 days. He expected to hit that target with regularity, building trust with product managers and their customers as he went.

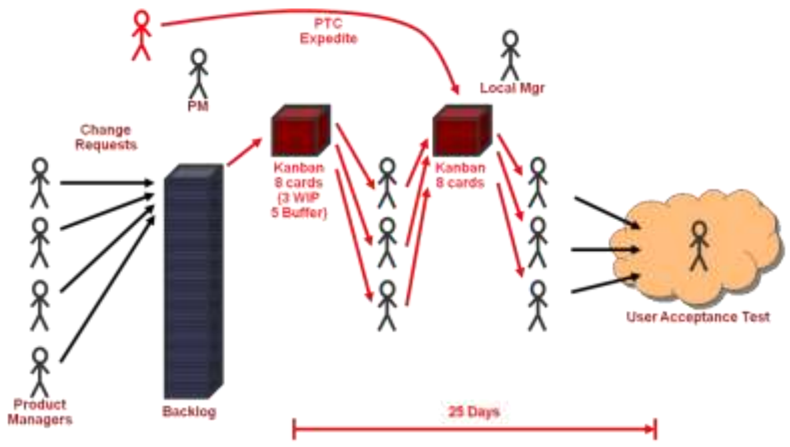


Figure 6 Workflow with Kanban WIP limits and Queues

Striking a New Bargain

Dragos took an offer to the product managers. He asked them to accept that he would meet once per week to discuss prioritization, that he would limit work-in-progress, and that his team would stop estimating. In exchange for this he would guarantee delivery within 25 days and that he'd report due date performance against this metric.

So the customers were being asked to give up ROI calculations and inter-department budget transfers based on estimated effort per request. In exchange, they were being offered unprecedented short delivery times and reliability. To get around the accounting issues they were asked to accept that all requests were on average 11 days of engineering. They were asked to accept that costs were essentially fixed. They were being asked to ignore the cost accounting paradigm on which their inter-department budget transfers were based.

To justify this it was explained that the vendor had a 12 month contract and it was invoiced monthly. The vendor allocated people against the contract and those people were paid for regardless of whether they were working or not. The budget to fund this from the four product managers was simply a fixed amount allocated proportionately. Dragos guaranteed that each of the product managers would get a fair allocation of capacity. This would free them from tracking individual requests. If they could accept that their dollars bought them capacity and that capacity was guaranteed then hopefully they could ignore their bias for unit based costing and budget transfers. Some simple rules were created to decide who should pick to refill the queue, in order that capacity was allocated fairly. A simple weighted round-robin scheme was sufficient to achieve this.

Implementing Changes

While the product managers and many of Dragos' management colleagues in the XIT unit remained skeptical, the consensus was that he should give these changes a try. After all, things were bad and getting worse. These changes could surely not make it any worse than it already was! Someone had to try something and Dragos was expected to implement some changes.

So the changes were enacted.

It began to work. Requests were processed and released to production. Delivery times on new commitments were met within the 25 day promise. The weekly meeting worked smoothly and the queue was replenished each week. Trust began to build with the product managers.

Adjusting Policies

You may wonder, how prioritization was facilitated if there was no longer an ROI calculation performed. It turned out that it wasn't needed. If something was important and valuable then it was selected for the input queue from the backlog and if it wasn't, then it wasn't selected. Some time later it was recognized that a new policy was needed. Any item older than 6 months was purged from backlog. If it wasn't important enough to be selected within 6 months of arrival then it was assumed it wasn't important at all. If it truly was important then it could be resubmitted.

Note: This is a common theme in the Kanban Method. The combination of explicit policies, transparency and visualization, empowers individual team members to make their own decisions and to manage risks themselves. Management comes to trust in the system because they understand that the process is made of policies. The policies are designed to manage risk and deliver customer expectations. The policies are explicit, work is tracked transparently, and all team members understand the policies and how to use them.

What about the governance policy to prevent large items sneaking through maintenance when they should be submitted to a major project? This was solved this by accepting that some might sneak through. The historical data told us that these were less than 2% of requests. So he instructed the developers to be alert and if a new request they started appeared to be large and they estimated that it was greater than 15 days of effort then they should alert their local manager. The risk and cost of doing this was less than one half of one percent of available capacity. It was a great tradeoff. By dropping estimates the team gained more than 33% of capacity at the risk of less than 1% of that capacity. This new policy empowered developers to manage risk and to speak up when needed!

The first two changes were left to settle in for 6 months. A few minor changes were made during this period. As I mentioned, a backlog purge policy was added. The weekly meeting with product owners also disappeared. The process was running so smoothly that Dragos' had the Product Studio tool modified to to send him an email when a slot became available in the input queue. He would then alert the product owners via email, who would decide amongst themselves who should pick next. A choice would be made and a request from the backlog replenished into the queue within 2 hours of it becoming available.

Looking for Further Improvements

Dragos began looking for further improvement. He'd been looking at historical data for tester productivity from his team and comparing with other teams within XIT supplied by the same vendor. He suspected that the testers were not heavily loaded and had a lot of slack capacity. By implication the developers were a significant bottleneck. He decided to visit the team in India. On his return he instructed the vendor to make a headcount allocation change. He reduced the test team from 3 to 2 and added another developer. This resulted in a near linear increase in productivity with the throughput for that quarter rising from 45 to 56.

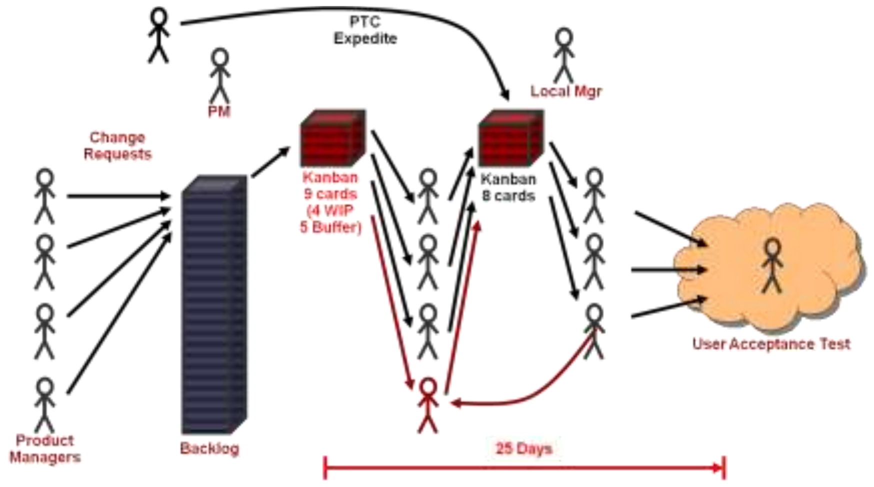


Figure 7 Resource Leveling

Microsoft's fiscal year was ending. The significant improvement in productivity and the consistency of delivery from the XIT Sustained Engineering (software maintenance) team was being noticed by senior managers. Finally, there was management trust in Dragos and the techniques he was employing. The department was given enough money for 2 more people. One developer and one additional tester were added in July 2005. The results were significant.

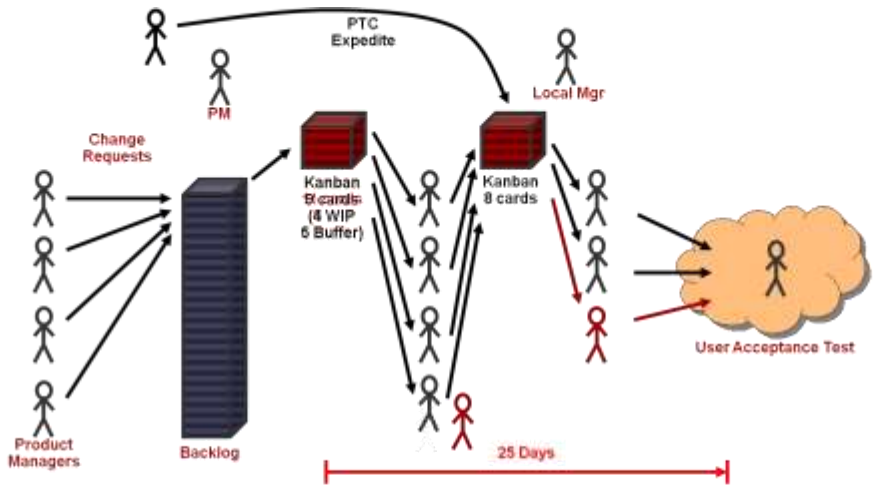


Figure 8 Adding additional resources

Results

The additional capacity was enough to increase throughput beyond demand. This resulted in the backlog being eliminated entirely on November 22nd 2005. By this time the team had reduced the lead time to an average of 14 days against an 11 day engineering time. The due date performance on the 25 day delivery time target was 98%. The throughput of requests had risen by more than three times while lead times had dropped by more than 90% with reliability improving almost as much. No changes were made to the software development or testing process. The people working in Hyderabad were unaware of any significant change. The PSP/TSP method was unchanged and all the corporate governance, process and vendor contract requirements were met in full. The team was awarded the Engineering Excellence Award for the 2nd half of 2005. Dragos was rewarded with additional responsibilities and the day-to-day management of the team was handed off to the local line manager in India who relocated to Washington.

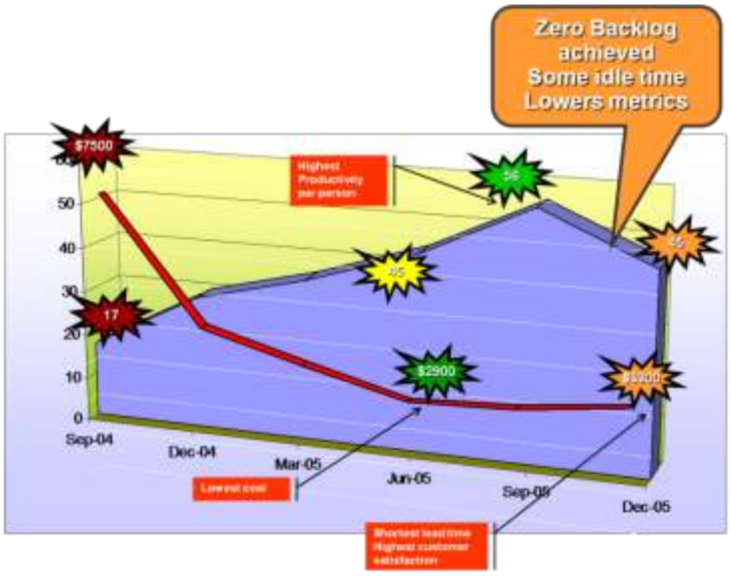


Figure 9 Quarterly Throughput overlaid with Unit Cost

These improvements came about in part because of the incredible managerial skill of Dragos Dumitriu but the basic elements of mapping a value-stream, analyzing flow, setting WIP limits and implementing a pull system, were key enablers. Without the flow paradigm and the kanban approach of limiting WIP, the change in performance would not have been possible. Kanban enabled incremental changes with low political risk, and low resistance to change.

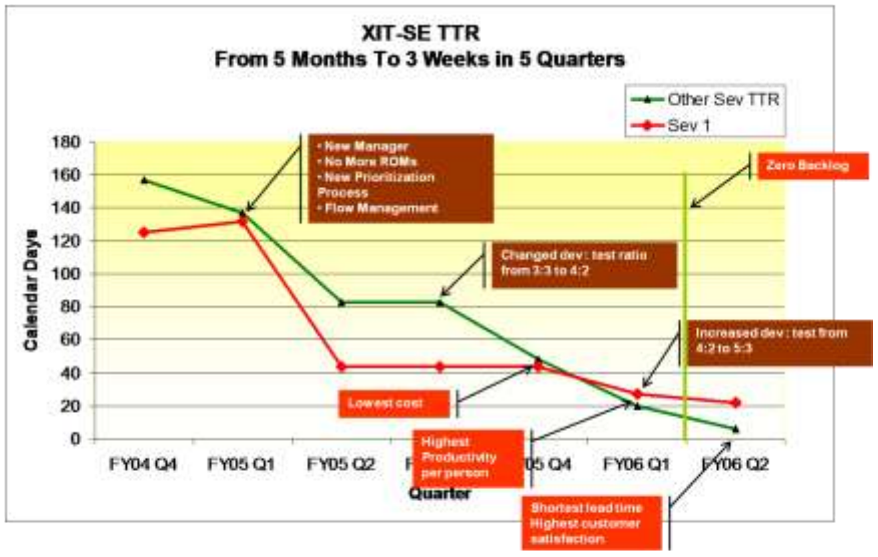


Figure 10 XIT Sustaining Time to Resolve shown in Microsoft Fiscal Years

The XIT case shows the use of a WIP limited pull system on a distributed project using offshore resources facilitated with an electronic tracking tool. There was no visual board and many of the more sophisticated features of the Kanban method described in this book were yet to emerge. Nevertheless, what manager could ignore the possibility that a greater than 3x productivity improvement with a 90% lead time reduction with significantly improved predictability was possible with minimal political risk and resistance to change?

Takeaways

- The first Kanban system was implemented with the XIT Sustained Engineering software maintenance team at Microsoft starting in 2004
- The first Kanban system was implemented with an electronic tracking tool
- The first Kanban system was implemented with an offshore team at a CMMI model level 5 appraised vendor in Hyderabad, India
- The workflow should be sketched and visualized
- The process should be described as an explicit set of policies
- Kanban enables incremental changes
- Kanban enables change with reduced political risk
- Kanban enables change with minimal resistance
- Kanban will reveal opportunities for improvement that do not involve complex changes to engineering methods
- The first Kanban system produced a greater than 3x productivity boost, a 90% lead time reduction and a similar improvement in predictability
- Significant improvements are possible by managing bottlenecks, eliminating waste and reducing variability that affects customer expectations and satisfaction
- Changes can take time to take full effect. This first case study took 15 months to enact in full.

Chapter 5 A Continuous Improvement Culture

In Japanese the word “kaizen” literally means “continuous improvement.” A workplace culture where the entire workforce is focused on continually improving quality, productivity and customer satisfaction is known as a “kaizen culture.” Very few businesses have actually achieved such a culture. Companies like Toyota, where the employee participation in their improvement program is close to 100% and on average each employee gets one suggestion implemented as part of on-going improvement every year, are very rare.

In the software development world, the Software Engineering Institute (SEI) of Carnegie Mellon University has defined the highest level of their Capability Maturity Model Integration (CMMI) as Optimizing. Optimizing implies that the quality and performance of the organization is continuously being refined. While not explicitly stated, as the CMMI says little about culture, achieving optimizing behavior as an organization is more likely to happen with a kaizen culture.

Kaizen Culture

If we are to understand why it is so hard to achieve a kaizen culture, we must first understand a little bit more about what such a culture would look like. Once we know that we can discuss why we might want to achieve such a culture and what the benefits might be.

In kaizen culture the workforce is empowered. Individuals feel free to take action. They feel free to do the right thing. They spontaneously swarm on problems, discuss options and implement fixes and improvements. In a kaizen culture, the workforce is without fear. The underlying norm is for management to be tolerant of failure if experimentation and innovation was in the name of process and performance improvement. In a kaizen culture, individuals are free (within some limits) to self-organize around the work they do and how they do it. Visual controls and signals are in evidence and work tasks are generally volunteered rather than assigned by a superior. A kaizen culture involves a high level of collaboration and a collegiate atmosphere where everyone looks out for the performance of the team and the business above themselves. A kaizen culture focuses on systems level thinking while making local improvements that enhance the overall performance.

A kaizen culture has a high level of social capital. It is a highly trusting culture where individuals, regardless of their position in the decision making hierarchy of the business, respect each other and each person’s contribution. High trust cultures tend to have flatter structures than lower trust cultures. It is the degree of empowerment that enables a flatter structure to work effectively. Hence a achieving a kaizen culture may enable elimination of wasteful layers of management and reduction of coordination costs as a result.

Many aspects of a kaizen culture are in opposition to established cultural and social norms in modern Western culture. In the West, we are brought up and educated to be competitive. Our school systems encourage competition in academic pursuits and sports. Even our team sports tend to encourage the development of heroes and teams built around one or two exceptionally talented players. The social norm is to focus on the individual first and to rely on exceptional individuals to deliver victory or to save us from peril. It is little wonder that we struggle in the workplace to encourage collegiate behavior and systems level thinking and cooperation.

Kanban accelerates organizational maturity and capability

The Kanban method is designed to minimize the initial impact of changes and reduce resistance to adopting change. Adopting Kanban should change the culture of your 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 SEI refers to this as a capability at Organizational Innovation and Deployment (OID)[xv] within the CMMI model. It has been shown[xvi] that organizations that achieve this high level of capability in change management can adopt Agile development 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[xvii] that Kanban merely optimizes something that needed to be changed. However, there is now considerable empirical evidence[xviii] that Kanban accelerates the achievement of high levels of organizational maturity and capability in core high maturity process areas such as Causal Analysis and Resolution (CAR) and Organizational Innovation and Deployment.

When choosing to use Kanban as a method to drive change in your organization, you are subscribing to the view that it is better to optimize what already exists because doing so is easier and faster and will meet 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.

Case Study: Corbis Application Development

When I introduced a kanban system at Corbis in 2006, I did so for many of the mechanical benefits that were demonstrated with Microsoft XIT in 2004 (see Chapter 3). The initial application was the same – IT systems maintenance. I was not anticipating a significant cultural shift or a significant shift in organizational maturity. I did not expect what we now know as the Kanban Method to evolve from this work.

While writing this book in 2009, it is now established that Kanban is a natural fit for IT maintenance work. At that time it wasn't clear but the form of a kanban system seemed to fit well with the functional problems of maintenance work. I didn't go to Corbis with the intent of "doing Kanban." I did go there with the intent of improving the customer satisfaction with the software development team. It was a happy coincidence that the first problem to be addressed was a lack of predictability and delivery from IT software maintenance.

Background and Culture

Corbis was at that time privately owned by Bill Gates, with a workforce of approximately 1300 people worldwide. It controlled the digital rights to many fascinating works of art as well as representing 3000 professional photographers, licensing their work for use by publishers and advertisers. It was the 2nd largest stock photography company in the world. There were some other lines of business, the most notable of which was the rights licensing business that controlled the licensing rights to the image and name of personalities and celebrities including some who are no longer with us, such as Albert Einstein and Marilyn Monroe. The IT department consisted of about 110 people split between software engineering and network operations and systems maintenance. From time to time the workforce was augmented with contract staff working on major projects. At its peak, in 2007, the software engineering department was 105 people including 35 contingent staff in Seattle and another 30 at a vendor in Chennai, India. Most of the testing was performed by this team in Chennai. In addition, there was a very traditional approach to project management with everything planned in a dependency tree of tasks and rolled up by an emerging and maturing program management office. It was a conservative company, in a relatively conservative and slow moving industry using conservative, traditional approaches to project management and the software engineering lifecycle.

The IT department maintained a diverse set of approximately 30 systems. Some were pretty typical accounting and human resource systems; others were exotic and at times esoteric applications for the digital rights management industry. The range of technologies and software platforms and languages was wide. The workforce was incredibly loyal to the company and many people in the IT department had been with the company more than 8 years with some having 15 years of service. Not bad for a company that was about 17 years old. The existing process was a traditional waterfall style software development lifecycle (SDLC) that had been institutionalized over the years with the creation of a business analysis department, a systems analysis department, a development department and an offshore testing department. Within these departments there were many specialists, such as analysts whose background was accounting whose specialty was finance. Some developers were also specialists, for example, J.D. Edwards programmers.

None of this was ideal. However, it was what it was. Things were the way they were. When I joined the company there was some expectation and trepidation that I would impose an Agile method and force the workforce through my positional power to change their behavior. While this may have worked, it would have been brutal and the impact during the change would have been severe. I was afraid of making things worse. I was afraid that projects would grind to a halt while new training was provided and staff adapted to new ways of working. I was also afraid of losing key members of personnel and aware of the fragility in the workforce due to the excessive levels of specialization. I chose to introduce a kanban system, get the systems maintenance work back on track and to see what happened from there.

The need for a Software Maintenance Function

Software maintenance (or "RRT" for "Rapid Response Team" as it was known internally) had been funded by the executive committee with an additional 10% budget for the software engineering department. This equated to 5 additional people in 2007. Some time before my arrival those 5 people had been hired. Due to the diverse nature of the systems involved and the existing high degree of specialization in the team it was decided that a dedicated team of 5 people to do maintenance work would not be a good solution. So, 5 additional people were added to the general pool of resources: 1 project manager; 1 analyst; 1 developer; and 2 testers. This introduced an additional complication that it was necessary, from a governance perspective, to show that the additional 5 people were actually doing maintenance work and hadn't simply been sucked into the major project portfolio. However, on any given day, those five people could be any of the approximately 55 people on the team.

One solution to this problem would have been to have everyone complete complex timesheets and to add an administrative burden to show that 10% of the team hours were being spent on maintenance activities. This would have been highly intrusive but it is typical of how middle managers respond to such a challenge. Another approach was to introduce a kanban system.

There had been an expectation set that a maintenance team would enable Corbis to make incremental releases to IT systems every two weeks. Major projects had typically involved major system updates and new systems releases once every three months. But as the business matured and the nature of these systems became more complex, this cadence of quarterly major releases had become intermittent. In addition, some of the existing systems were effectively end-of-life and really due for complete replacement. Legacy system replacement is a major challenge and typically involves long projects with many staff until a parity of functionality is reached and the old system can be turned off as the new system is brought online. [This approach to legacy system replacement is really far from optimal but it is typical.]

So the maintenance releases were the one tool available to Corbis IT to enable some form of business agility.

Small Projects for Maintenance wasn't Working

The existing system to deliver maintenance releases, the system that was broken, was to schedule a series of short two week long projects. This might have been recognizable as Agile Software Development using 2 week iterations but it wasn't. When I first arrived, the negotiation of scope for a 2 week release cycle was taking about 3 weeks. So the front end transaction costs of a release were greater than the value-added work. It was taking about 6 weeks to get a two week release out the door.

Implementing Change

It was clear before making any changes that the status quo was unacceptable. The current system was unable to deliver the required level of business agility. System maintenance gave us an ideal opportunity to introduce change. Maintenance work was generally not mission critical. It was nevertheless highly visible as the business had a direct input to the prioritization and the choices were highly tactical and important to short-term business goals. So system maintenance was something everyone cared about and wanted to work effectively. And finally, there was a compelling reason to make changes. The existing system was dissatisfying everyone. The developers, testers and analysts were all dissatisfied with the time wasted negotiating scope while the business was hugely dissatisfied with the results.

We designed a kanban system with scheduled bi-weekly releases, planned for 1pm every 2nd Wednesday, and with scheduled prioritization meetings with the business, set at 10am every Monday. So the prioritization cadence was weekly, while the release cadence was bi-weekly. The choice of cadence was determined through collaborative discussions with up-stream and down-stream partners and based on the transaction and coordination costs of the activities. A few other changes were made. We introduced an Engineering Ready (input) queue with a WIP limit of 5 items, and then added WIP limits throughout the lifecycle of analysis, development, build, and system test. Acceptance test, staging and ready for production were left unlimited as they were not considered capacity constrained and were to some extent outside our immediate political control.

Primary Effects of the Changes

The effects of introducing a kanban system were at one level unsurprising while at another they were quite remarkable. We started to make releases every two weeks. After about three iterations, these were happening without incident. The quality was good and there was little to no need for emergency fixes when the new code went live in production. The overheads of scheduling and planning releases had dropped dramatically and the bickering between the development teams and the program management office had almost completely disappeared. So kanban had delivered on its basic promise. We were putting out high quality releases, very regularly with a minimum of management overhead. Transaction and coordination costs of a release had been drastically reduced. The team was getting more work done and delivering that work to the customer more often.

It was the secondary effects that were all the more remarkable.

Unanticipated Effects of Introducing Kanban

On the development team, we introduced the physical card wall using sticky notes on a white board in January 2007. We started to hold a morning standup meeting around the board for 15 minutes at 9.30am each day. The physical board had a huge psychological effect in comparison to the electronic tracking tool. By attending the standup each day, team members were exposed to a sort of time-lapse photography of the flow of work across the board. Blocked work items were marked with pink tickets and the team became much more focused on issue resolution and maintaining flow. The productivity jumped dramatically.

With the flow of work now visible on the board, I started to pay attention to the workings of the process. As a result, changes were made on the board. My team of managers came to understand the changes I was making and why I was making them and by March, they were making the changes themselves. In turn, their team members, the individual developers, testers and analysts, started to see and understand how things worked. By early summer, anyone on the team felt empowered to suggest a change and we'd come to observe the spontaneous affiliation of (often cross-functional) groups of individuals who would discuss process problems and challenges and make changes as they saw appropriate. Typically, they would inform the management chain after the fact. What had emerged over approximately 6 months was a kaizen culture in the software engineering team. Team members felt empowered. Fear had been removed. They took a pride in their professionalism and their achievements and they wanted to do better.

Sociological Change

Since the Corbis experience, there have been other similar reports from the field. Rob Hathaway of Indigo Blue was the first to truly replicate these results with the

IT group at IPC Media in London. The fact that others have been able to replicate the sociological affects of Kanban observed at Corbis makes me believe that there is a causation and that it was neither a coincidence or a direct effect of my personal involvement.

I've thought a lot about what made these sociological changes happen. Agile methods have offered us transparency on work-in-progress for a decade and yet teams following the Kanban method appear to achieve a kaizen culture faster and more effectively than typical Agile software development teams. Often teams adding Kanban to their existing Agile methods find a significant improvement in social capital amongst team members. Why could this be?

My conclusion is that Kanban provides transparency into the work but also the process (or workflow.) It provides visibility into how the work is passed from one group to another. Kanban enables every stakeholder to see the affects or their actions or inactions. If an item is blocked and someone is capable of unblocking it, Kanban provides visibility into this. Perhaps there is an ambiguous requirement. The subject matter expert who can resolve the ambiguity might expect to receive an email with a request for a meeting. After a follow up call, they arrange a meeting to suit their calendar perhaps 3 weeks out. With Kanban and the visibility it provides, the subject matter expert realizes the effect of inaction and prioritizes the meeting, perhaps re-arranging their calendar to make a meeting this week rather than delay for a further two weeks.

In addition to the visibility into process flow, work-in-progress limits also force challenging interactions to happen sooner and more often. It isn't easy to ignore a blocked item and simply work on something else. This "stop the line" aspect of Kanban seems to encourage swarming behavior across the value stream. When people from different functions and with different job titles swarm on a problem and collaborate to find a solution, maintaining the flow of work and improving system level performance, the level of social capital and team trust is increased. With higher levels of trust engendered through improved collaboration, fear is eliminated from the organization.

Work-in-progress limits coupled to classes of services (explained in Chapter 11) also empower individuals to make scheduling decisions on their own without management supervision or direction. Empowerment improves the level of social capital by demonstrating that superiors trust subordinates to make high quality decisions on their own. Managers are freed up from supervising individual contributors and can focus their mental energy on other things such as process performance, risk management, staff development and improved customer and employee satisfaction.

Kanban within the team greatly enhances the level of social capital. The improved levels of trust and the elimination of fear encourage collaborative innovation and problem solving. The net effect is the rapid emergence of a kaizen culture.

Viral Spread of Collaboration

However, it was the results beyond the software engineering department at Corbis that were the most remarkable. How the viral spread of Kanban improved collaboration around the company is worth reporting and analyzing.

Each Monday morning at 10am, Diana Kolomiyets, the project manager responsible for coordinating the IT systems maintenance releases would convene the RRT prioritization board meeting. The business attendees were typically vice presidents. They ran a business unit and reported to a senior vice president or c-level officer of the company. Put another way, a vice president reported to an executive committee member. Corbis was still small enough that it made sense for such a high ranking manager to attend the weekly meeting. Equally, the tactical choices being made were sufficiently important that they really needed the direction of a vice president to influence a good choice.

Usually, on the Friday prior to the meeting each attendee received an email. It would state something like, ”We anticipate that there will be two slots free in the queue next week. Please examine your backlog items and select candidates for discussion at Monday’s meeting.”

Bargaining

In the first few weeks of the new process, some of the attendees would come with an expectation of negotiating. They might say, “I know there is only 1 slot free, but I have 2 small ones, can you just do them both?” This bargaining was rarely tolerated. The other members of the prioritization board ensured that everyone played by the rules. They might reply, “How do I know they are small? Should I take you at your word?” or counter with “I’ve got two small ones too. Why shouldn’t I get my favorites selected?” I refer to this as “The Bargaining Period” as it indicates the style of negotiation that took place at the prioritization meeting.

Democracy

After about 6 weeks, and coincidentally around the same time that the development team introduced the use of the physical white board, the prioritization board introduced a democratic voting system. They spontaneously volunteered this as they’d become tired of bickering. The bargaining at the meeting was wasting time. It took a few iterations to refine the voting system but it settled down to a system where each attendee got one vote for each slot free in the queue that week. At the beginning of the meeting, each member would propose a small number of candidates for selection. As time went by, proposing requests got more sophisticated: some people came with Powerpoint slides; others with spreadsheets that laid out a business case. Later we heard that some members were lobbying their colleagues by taking them to lunch. Deals were being done, "If I vote for your choice this week, you will vote for my choice next week." Underlying the new democratic system of prioritization, the level of collaboration between business units at the vice president level was growing. Though we didn't realize it at the time, the level of social capital across the whole firm was growing. When leaders of business units, start collaborating, so, it seems, do the people within their organizations. They follow the lead from their leader. Collaborative behavior coupled with visibility and transparency breeds more collaborative behavior. I refer to this period at "The Democracy Period."

Down with Democracy

Democracy was all very well but after a further 4 months, it seemed that democracy had failed to elect the best candidate. A considerable effort was expended implementing an e-commerce feature for the Eastern European market. The business case had been stellar but its candidature was suspect from the beginning and some had questioned the quality of the data in the business case. After several attempts this feature had been selected and was duly implemented. It was one of the larger features processed through the RRT system and many people got involved and noticed it. Two months after launch our Director of Business Intelligence did some data mining on the revenue generated. It was a fraction of the original business case and the estimated payback period against the effort expended was calculated at 19 years. Due to the transparency that Kanban offered us, a large number of stakeholders became aware of this and there was discussion about how precious capacity had been wasted on this choice when a better choice might have been made instead. That was the end of the Democracy Period.

Collaboration

What replaced it was quite remarkable. Bear in mind that the prioritization board were mostly vice president level employees and officers of the company. They had broad visibility into aspects of the business that many of us were unaware of. So at the beginning of the meeting, they started to ask, "Diana, what is the current lead time for delivery?" She might reply, "Currently we are averaging 44 days into production." So a simple question was asked, "What is the most important tactical business initiative in this company 44 days out from now?" There might be some discussion but typically there was swift agreement. "Oh, that'll be our European marketing campaign launching at the conference in Cannes." "Great! What items in the backlog are required to support the Cannes event?" A quick search might produce a list of six items. "So there are three slots free this week. Let's pick three from six and we'll get to the others next week." There was very little debate. There was no bargaining or negotiation. The meeting was over in about 20 minutes. I've come to refer to this period as "The Collaboration Period." It represents the highest level of social capital and trust between business units that was achieved during my time as Senior Director for Software Engineering at Corbis.

Cultural Change is perhaps the biggest benefit of Kanban

It was very interesting to see this emerge and to see how it affected the wider company as employees followed the lead of their vice presidents and started to collaborate more with their colleagues from other business units. This change was sufficiently profound that the recently appointed chief executive, Gary Shenk, called me to his office to ask if I had any explanation. He told me that he had observed a new level of collaboration and collegiate spirit in the senior ranks of the company and those formerly antagonistic business units seemed to be getting along a lot better. He suggested that the RRT process had something to do with it and asked if I had any explanation for it. While I am sure that I wasn't as articulate as I am now writing this chapter two years later, I convinced him that our kanban system had greatly enhanced collaboration and the level of social capital amongst everyone involved.

The cultural side-effects of what we now recognize to be capital "K", Kanban were quite unexpected and in many ways counter-intuitive. He asked me "Why aren't we doing all our major projects this way?" Why indeed? So we set about implementing Kanban in the major project portfolio. We did this because Kanban had enabled a kaizen culture and that cultural change was so desirable that changing the many mechanics of prioritization, scheduling, reporting, and delivery that result from implementing Kanban was considered a price worth paying.

Takeaways

- Kaizen means 'continuous improvement'
- A kaizen culture is one where individuals feel empowered, act without fear, affiliate spontaneously, collaborate, and innovate
- A kaizen culture has a high degree of social capital and trust between individuals regardless of their level in the corporate hierarchy
- Kanban provides both transparency on the work and the process through which the work flows.
- Transparency of process allows all stakeholders to see the effects of their actions or inactions
- Individuals are more likely to give of their time and collaborate when they can see the effect it will have
- Kanban WIP limits enable "stop the line" behavior
- Kanban WIP limits encourage swarming to resolve problems
- Increased collaboration from swarming on problems and interaction with external stakeholders raises the level of social capital on the team and the trust between team members
- Kanban WIP limits and Classes of Service empower individuals to pull work and make prioritization and scheduling decisions without supervision or direction from a superior
- Increased levels of empowerment increase social capital and trust between workers and managers
- Collaborative behavior can spread virally
- Individuals will take their lead from senior managers. Collegiate, collaborative behavior amongst senior leaders will affect the behavior of the whole workforce

Section Three
Implementing Kanban

Chapter 6 Mapping the Value Stream

Kanban is an approach that drives change by optimizing your existing process. The essence of starting with Kanban is to change as little as possible. You must resist the temptation to change workflow, job titles, roles and responsibilities, and specific working practices. Everything from which the team members and other partners, participants and stakeholders derive their self-esteem, professional pride and ego should be left unchanged. The main target of change will be the quantity of WIP and the interface to and interaction with the upstream and downstream processes. So you must work with the team to map the value stream as it exists. Try not to change it or invent it in an idealistic fashion.

In some political situations, there may be an official process that is not being followed. When you attempt to map the value stream the team will insist that you re-document the official process and not the actual process as used. You must resist this and insist that the team map the process they actually use. Without this it will be impossible to use a card wall as a process visualization tool because team members can only use the card wall if it reflects what they actually do.

Defining a Start and End Point for Control

It is necessary to decide where to start and end process visualization and in doing so define the interface points with upstream and downstream partners. It's important to handle this early stage Kanban implementation sensitively as making poor choices at this stage may invoke a failure. Successful teams have tended to stick to adopting workflow visualization with cards and limiting WIP within their own political sphere of control and negotiating the new way of interacting with immediate upstream and downstream partners. For example, if you control the engineering or software development function and have control or influence over analysis, design, testing, and coding then map this value stream and negotiate new styles of interaction with the business who provide requirements, prioritization, and portfolio management, upstream, and with system operations, or production maintenance functions downstream. By drawing the boundary this way, you are only asking your own team to adopt the behavior of limiting WIP. You are not asking upstream or downstream teams to change the way they do their jobs. You are not asking them to limit WIP and implement a pull system. However, you are asking them to interact differently with you – to interact in a way that is compatible with the pull system you want to implement.

Work Item Types

Once you have selected the point in the workflow or value stream identify the types of work that arrive at that point and any others that exist within the workflow that will need to be limited. For example, bugs are likely a type of work that exists within the workflow. You may also identify other types of development centric work such as refactoring, systems maintenance and infrastructure upgrades and related rework. For incoming work you may have types like User Story or Use Case or Functional Requirement or Feature. In some cases, the incoming types might be hierarchical such as Epic, a collection of user stories.

Typical work item types seen on teams adopting Kanban have included but are not limited to: Requirement; Feature; User Story; Use Case; Change Request; Production Defect; Maintenance; Refactoring; Bug; Improvement Suggestion; and Blocking Issue.

Anatomy of a Work Item Card

Each visual card representing a discrete piece of customer-valued work has several pieces of information on it. The design of the card is important. The information on the cards must facilitate the pull system and empower individuals to make their own pull decisions. The information on a ticket may vary by work item type or by class or service (discussed in chapter 11).

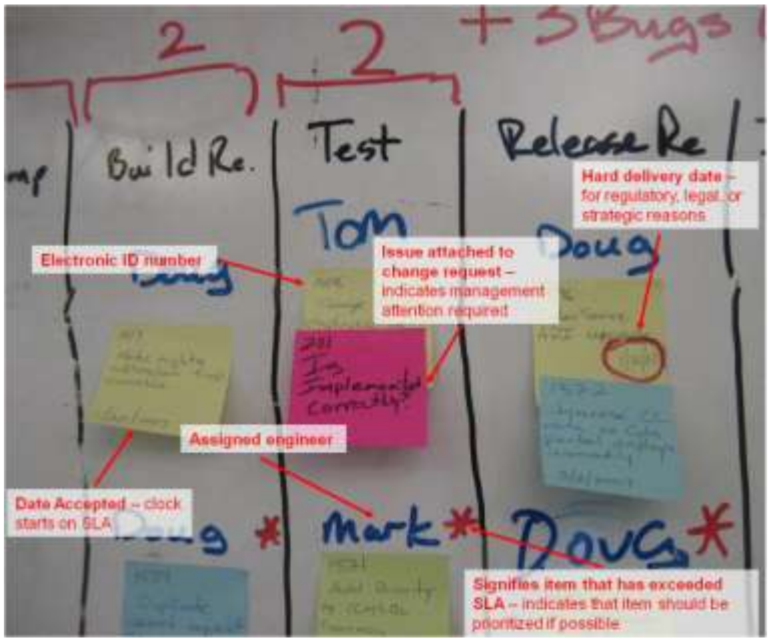

+ 3 Bugs
Build Re.
Test
Release Re
Hard delivery date – for regulatory, legal, or strategic reasons
Electronic ID number
Issue attached to change request – indicates management attention required
Assigned engineer
Date Accepted – clock starts on SLA
Signifies item that has exceeded SLA – indicates that item should be prioritized if possible
Tom
Doug
Mark

Figure 11. Close up of card wall showing anatomy of work item cards

In the example in the picture, the number in the top left is the electronic tracking number used to uniquely identify the item and to link it to the electronic version of the tracking system. The title of the item is written in the middle. The day the ticket entered the system is written in the bottom left. This serves a double purpose. It facilitates first-in, first-out (FIFO) queuing for the standard class of service and it allows team members to see how many days have expired against the service level agreement (described in chapter 11). For fixed delivery date class of service items, the required delivery date is written in the bottom left of the ticket.

In this early example from 2007, some other information is shown off-ticket. A star designates that the item is late against the target cycle time in the service level agreement. In some more recent implementations this is achieved by attaching a sticker to the upper right hand corner of the ticket. The name of the assigned person is also written above the ticket. The assigned person will change as the ticket flows across the board. Hence, it is undesirable to write it on the ticket. However, more recent implementations have features small name tags stuck to the items, the use of magnets (where the white board is magnetic), and the use of stickers or magnets that feature avatars of team members. Characters from South Park are a popular choice of avatar. Any mechanism that allows team members and immediate management to see at a glance who is working on what, will suffice.

As a general rule the design of the ticket used to represent an individual piece of work should have sufficient information to facilitate project management decisions such as which item to pull next without the intervention or direction of a manager. The idea is to empower team members with transparency of process, project goals and objectives and risk information. As you discover more about classes of service and service level agreements, you'll discover that Kanban facilitates a powerful self-organizing risk management mechanism. Equally, Kanban, by empowering team members to make their own scheduling and prioritization decisions, shows respect for individuals and a trust in the system (or process design.) A well designed work item card is a key enabler of high trust culture and a Lean organization.

Drawing a Card Wall

It's typical to draw card walls to show the activities that happen to the work rather than specific functions of job descriptions. Often there is a strong overlap between a function and an activity, for example, analysts perform analysis. However, the convention with Kanban in software projects over the last few years has been to model the work and not the workers, the functions or the handoffs between functions.

Start by drawing columns on the board to represent the activities performed in the order they are performed. When drawing columns initially, it makes sense to draw these with a marker. However, through usage the lines will be erased. During the first few weeks you may find that you make a few changes to the workflow and continuing to use an erasable marker makes sense. However, there will be a point you will want something more permanent. Very narrow rolls of vinyl tape are available from office supply stores, specifically designed for precision work on whiteboards. At Corbis it became commonplace to delineate columns and rows on a card wall using this tape. This practice is now widely adopted with teams using various grades and widths of tape to mark rows and columns.

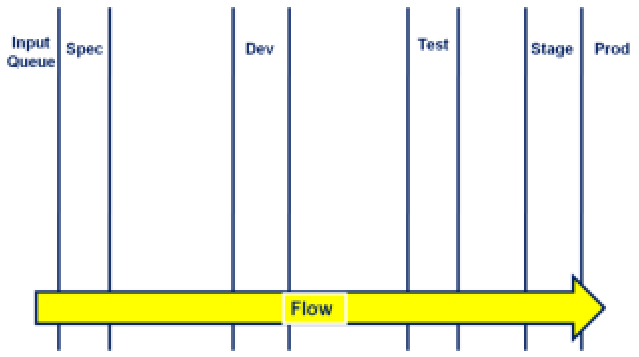


Figure 12. Outline workflow on card wall (flow left to right)

The convention has been to model flow from left to right though some teams have adopted a bottom to top approach.

[Insert picture from Motley Fool or SEP]

Note that for activity steps it is necessary to model the in-progress and complete and by convention this is done by splitting the column.

Next, add the input queue and any downstream delivery steps that you wish to visualize as shown in figure 12.

Now add any buffers or queues that you believe are necessary. There are some differing schools of thought on this and it is really an advanced topic. A full discussion of where to put buffers and how to size them is beyond the scope of this book. It will suffice to describe two popular approaches.

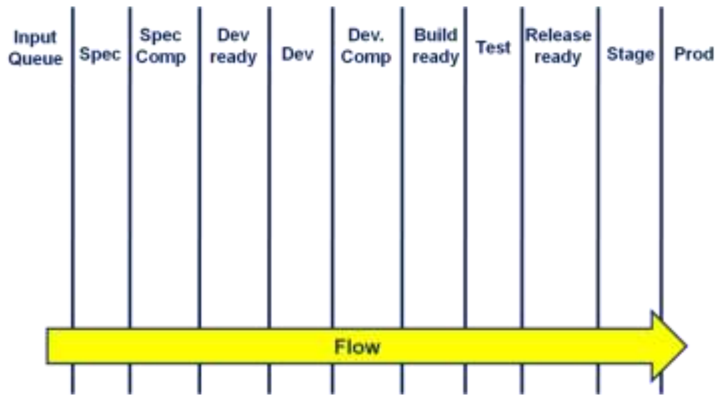


Figure 14. Workflow with added buffers and queues

The first school of thought says do not try to second guess the location of bottleneck or source of variability that will require a buffer. Rather implement the system and wait for the bottleneck to reveal itself and then make changes to introduce a buffer.

A variant on this suggests that WIP limits should be set fairly loosely initially so that variability, waste and bottlenecks do not have a significant impact on the pull system when it is first implemented. This is discussed more fully in chapter 10 and later in chapters 17 & 19.

The second school of thought takes a different approach to the ideas from above. It suggests that rather than implement loose WIP limits to avoid a challenging introduction of the system, each stage should be buffered while the activity steps should have tight limits. Bottlenecks and variability will reveal themselves by how full the buffers become. Small simple changes can then be made to reduce buffer sizes and eventually eliminate unnecessary buffers.

At the time of writing, there isn't enough evidence to suggest which approach is better.

Some teams have adopted a convention of showing buffers and queue columns by using a 45 degree rotated card. This provides a strong visual indicator of how much work is flowing rather than queuing at any given instant in time. This allows the team and other stakeholders to literally "see" the amount of economic cost (or waste) in the system.

[Figure 15. Insert a picture from Jeff Patton's examples]

Electronic Tracking

Electronic tracking has been a feature of kanban systems in software development since their first introduction in 2004. It is, however, optional. For teams that are distributed geographically or who have work from home policies that allow team members to work from their homes one or more days per week, electronic tracking is essential. Electronic tracking can be performed with basic ticketing and work item tracking systems such as Jira, Microsoft Team Foundation Server, Fog Bugz, and HP Quality Center. However, a more powerful system will allow you to visualize the workitem tracking as if it were on a card wall.

At the time of writing a number of web based and application based tools are emerging on the market to provide electronic tracking using visual boards that simulate card walls with columns, WIP limits and other essential aspects of Kanban. These include but are not limited to: Lean Kit Kanban; Agile Zen; Target Process; Silver Catalyst; RadTrack; Kanbanery; VersionOne; Greenhopper for Jira; Flow.io; and some additional open source projects to add Kanban interfaces to tools such as Team Foundation Server and FogBugz.

Electronic tracking is necessary for teams that aspire to higher levels of organizational maturity. If you anticipate the need for quantitative management, organizational process performance (comparing the performance across kanban systems, teams or projects), and/or causal analysis and resolution (root cause analysis based on statistically sound data) then you will want to use an electronic tool from the beginning.

Setting Input & Output Boundaries

Align the design of the kanban system and card wall with the decision made early to limit the boundary of WIP control. It is likely that upstream and downstream partners will ask later to visualize their work on your card wall. However, it is better to provide transparency onto your own work first and wait for others to ask to be part of your Kanban initiative.

In the example shown in figure 16, the input is defined as the backlog generated by the business analysts. In this organization, the business analysts reported to the Program Management Office (PMO) which in turn reported to the Chief Finance Officer. Neither the business analysis function nor the PMO reported within the IT department. There was little trust or collaboration between the managers in these functions and the IT department. It made sense to set the input point at this step in the lifecycle.

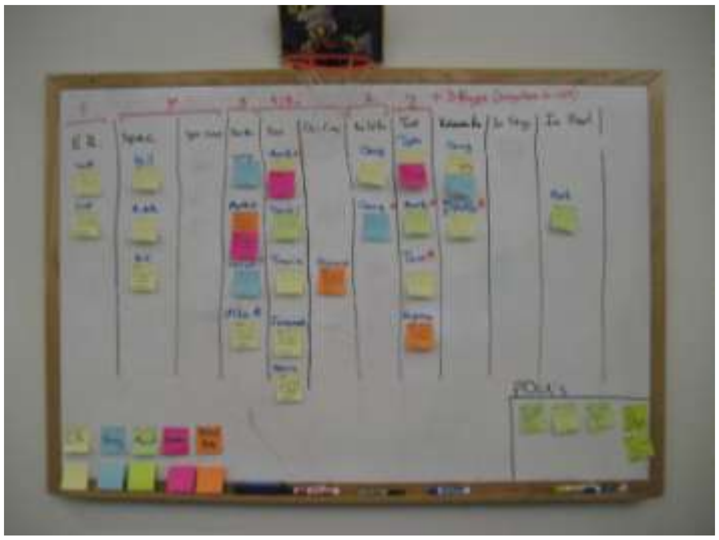


Figure 16. Showing Engineering Ready (E.R.) Input Queue

In this example, the downstream handoff is the deployment to production. Once the software is deployed and handed off the to the network and systems operations department for daily maintenance and support it is considered out-of-scope.

Coping with Concurrency

One common occurrence when designing a card wall for a kanban system is a process where two or more activities can happen concurrently, for example, software development and test development. There are two basic patterns for coping with this situation. One is not to model it at all. Just leave a single column where both activities can occur together. This is simple but many teams have not preferred it. Some teams have adapted this model by using different colors or shapes of ticket to show the different activities.

[Figure 17. Show example of an open column for concurrency]

The other option is to split the board vertically into two (or more) sections.

[Figure 18. Show example of a split board for dev and test dev]

In this example, some tagging mechanism to tie items in the top and bottom of the board is necessary. This may simply involve use of, for example, the top right hand corner of the ticket to cross reference the associated item. In a good electronic tracking system it will be possible to link related items such as development and test activities.

Coping with Unordered Activities

Particularly in highly innovative and experimental work, there may be several activities that need to happen with a piece of customer-valued work, but those activities do not need to happen in any particular order. In these circumstances, it is important to realize that Kanban should not force you to complete the activities in a given order. What is most important when modeling your kanban system is that it must reflect the way the real work is done.

There are a couple of strategies to the multiple unordered activities problem. The first is similar to coping with concurrency, simply have a single column as a bucket for the activities and do not explicitly track which of them is complete on the board.

[Figure 19. Insert picture]

The second and potentially more powerful choice is to model the activities in a similar fashion to the concurrent activities. In this design, the tickets have to move vertically up and down the column as they are pulled into each of the specific activities. Visualizing which activities have been completed to which item can be done by modifying the ticket design to have a small box for each activity. When the activity is complete, the box can be filled to visually signal that the item is ready to be pulled to another activity in the same column. If all the boxes are filled then the item is ready to be pulled to the next column on the board or can be moved to a “done” column.

[Figure 20. Insert picture showing four unordered activities and a done column. Show partially complete tickets with some of the four boxes filled, and some complete tickets in the done column with all four boxes filled.]

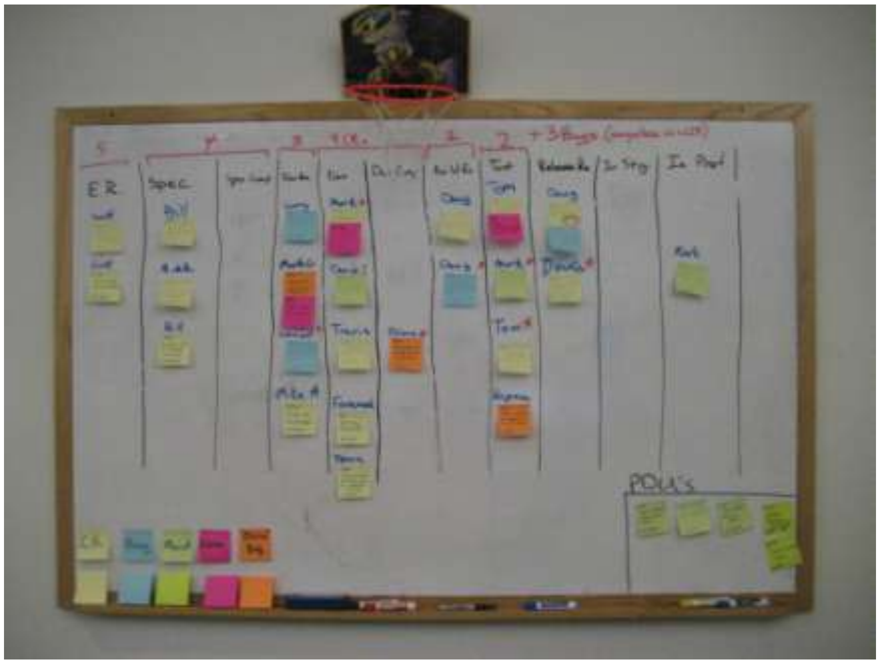
Takeaways

- Decide the outer boundaries of the kanban system. It is often best to limit this to immediate span of political control. Do not force visualization, transparency and WIP limits on any department that does not volunteer to collaborate.
- Model the card wall in alignment with the decision to bound the limiting of WIP and visualization of work.
- Define work item types and model how their work flows. Some types may not require every step in the flow
- Design the work item card to have enough information to facilitate self-organization for pull and to enable team members to make good quality decisions with respect to risk based on work item type, service level agreements and classes of service
- Use an electronic tracking system if the team is distributed, has some work from home policy, or aspires to higher maturity behaviors that require quantitative information that an electronic system can provide
- Where appropriate discuss methods for and choose how to model and visualize concurrency in activities
- Where appropriate discuss methods for and choose how to model and visualize activities which do not need to follow a specifically ordered flow

Chapter 7 Coordination in Kanban Systems

Visual Control & Pull

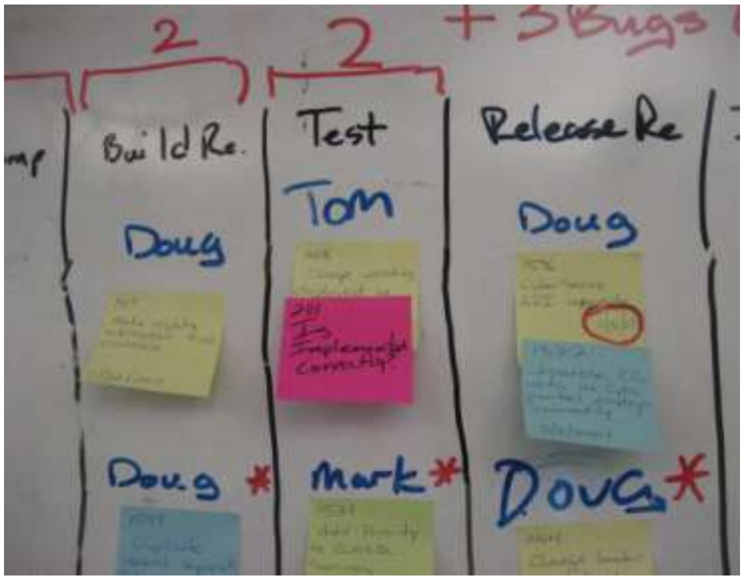
When people talk about kanban the most popular form of coordination that comes to mind is a card wall. Typically, the work-in-progress limits are drawn on the board at the top of each column or across a span of columns. Pull is signaled if the number of cards under the limit is less than the limit. In the photograph we can see that the limit above Specification is 4 items. However, there are only 3 cards under the limit. 4 – 3 = 1. This is signaling that we can pull one item into Specification (the systems analysis function) from the input queue, Engineering Ready. In turn the input queue has a limit of 5. There are currently only 2 items remaining in the queue. When we pull one into Analysis there will only be one item remaining. 5 – 1 = 4. This is signaling that we can prioritize 4 new items into the input queue at the next prioritization meeting.



Picture 21. Showing kanban limits above columns on a card wall

When the team decides to pull an item, they can choose which item to pull based on available visual information such as the work item type, the class of service, the due date (if applicable) and the age of the work item. Policies for pull related to class of service are discussed in chapter 11.

Picture 22 shows a close up of sticky notes representing work items in our card wall. Color is being used to communicate a combination of work item type and class of service. The name of the owner or assigned member of staff is written above the card. Some teams like to use additional smaller sticky notes with names, or small avatars stuck to the work item to signify who is working on it. This allows everyone on the team to see who is working on what.



Picture 22. Close up of card wall showing anatomy of card details

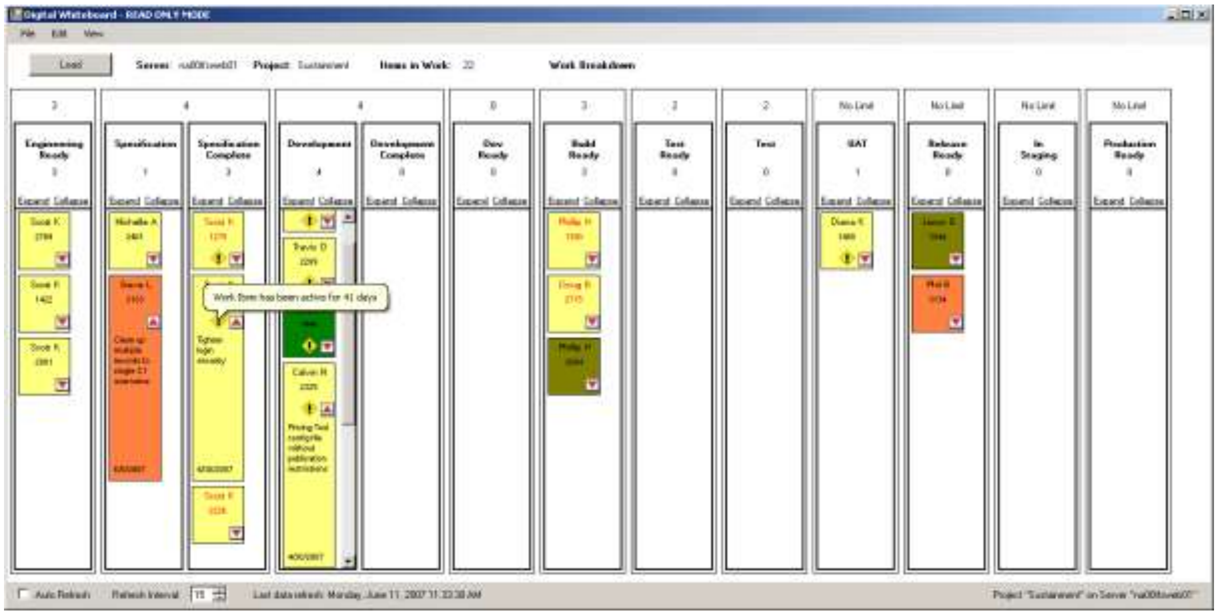
The electronic tracking number is shown in the top right hand corner of the sticky note. The date an item entered the input queue is shown in the bottom left. The age of the item can be deduced from this date. If an item is of a class of service where there is a guaranteed delivery date then that is shown in the bottom right. If an item is late then that is signified with a red star above top right of the card. If something is blocked the issue ticket in pink is attached to the blocked item. The issue is a first class work item in its own right and hence has its own electronic tracking number, a date when it entered the system and potentially an assigned member of staff.

This scheme is idiosyncratic to the first kanban implementation at Corbis. Your implementation will almost certainly differ. However, you are likely to want to visually capture the assigned member of staff, the start date, the electronic tracking number, the work item type, the class of service, and some status information such as whether the item is late. The goal is to visually communicate enough information to make the system self-organizing and self-expediting at the team level. As a visual control mechanism the kanban board should enable team members to pull work without direction from their manager.

Electronic Tracking

As an alternative, or a supplement to, a card wall an electronic system is often used to track work in a kanban system. At the time of writing there are many card wall tools available both commercially and as open source, however, few of them offer the kanban signaling mechanism out-of-the-box. There have been various plug-in or overlay products offered on generic work-item tracking systems. For example, a product called Green Hopper offers kanban tracking on top of the Jira issue tracking system. There have been several implementations of kanban tracking on top of Visual Studio Team Foundation Server. Some of these are available in open source.

With my own team, we implemented our own *Digital Whiteboard* application (pictured) on top of Team Foundation Server. In the case study in chapter 3, the electronic tracking was done using an internal tool at Microsoft called Product Studio. This was a forerunner to Team Foundation Server and since 2005 Microsoft have replaced the use of Product Studio with Team Foundation Server for their internal development project tracking.



Picture 23. Digital Whiteboard application used at Corbis

The application shown in picture 23 shows the kanban limits grouped above columns. It is capable of visually showing when a kanban limit has been exceeded. It also displays a number of status items for each work item, including icons to show whether an item is late and another showing whether it is blocked with an issue.

Electronic tracking is important for kanban systems because it permits several things not viable with a simple card wall. Electronic tracking allows the gathering of data that can be used to generate metrics and reports for both day-to-day management and retrospective use at monthly operations reviews.

Daily Standup Meetings

Standup meetings are a common element of agile development meetings. They typically take place in the morning before work starts and have a generally agreed format. A typical standup meeting is a for a single team of up to 12 people – typically about 6. The format usually involves working around the assembled group and asking 3 questions: what did you achieve yesterday?; what will you do today?; are you blocked or do you need assistance?. Each team member answers these questions and the team is coordinated to do its work for the day.

Standup meetings have evolved differently with kanban. The need to go around the room and ask the 3 questions is obviated by the card wall. The wall contains all the information about who is working on what. Attendees who come regularly can see what has changed since yesterday and whether something is blocked or not is visual evident. So standups take a different format in a kanban system. The focus is on flow of work. The facilitator who is typically a project manager or line manager will work through the issues (pink tickets) on the board. The team will discuss briefly who is working an issue and when it will be resolved. There will also be a call for any other blocking issues that are not on the board and for anyone who needs help to speak up.

As a result of this different focus, standup meetings have been able to scale to larger teams. The picture shows a standup for a large project where there were often 50 people attending. Despite the large number of attendees these meetings were completed in around 10 minutes.

The After Meeting

The After Meeting consists of huddles of small groups of 2 or 3 people. This emerged as spontaneous behavior as folks want to discuss something on their mind: perhaps a blocking issue; perhaps a technical design or architecture issue; but more often a process related issue. The After Meeting is a vital element of the cultural transformation that emerges with kanban. After Meetings generate improvement ideas and result in process tailoring and innovation.

On larger projects some After Meetings took the form of established agile Scrum style standup meetings. Teams of up to 6 people working together on a feature, story or requirement would meet briefly to coordinate their efforts for the day. There is an interesting difference between this emergent kanban process behavior and Scrum. With Scrum the teams meet first and then send a delegate to a Scrum-of-Scrums to coordinate a program or large project. In kanban it works in reverse. The program level meeting happens first.

Prioritization Meetings

Prioritization meetings are held with a group of business representatives or product owners (to use popular agile development vernacular). It's recommended that these meetings happen at a regular interval. Providing a steady cadence for prioritization meetings reduces the coordination cost of holding the meeting and provides certainty and reliability over the relationship between the business and the software development team.

The purpose of a prioritization meeting is to fill the input queue for a kanban system for a single value stream, system or project. The meeting should involved representatives of the business who have an interest in the deliveries from the team and who have items waiting in the backlog. It is recommended that the business attendees be as senior in their organizations as possible. More senior people can make more decisions and often have access to a wider set of business contextual information which improves the quality of their decision making.

Ideally, a prioritization meeting will involve several product owners or business people from potentially competing groups within the company. The tension created by this actually becomes a positive influence on good decision making and stimulates a healthy collaborative environment with the software development team. If there is only one product owner there is the potential for the relationship to adversarial.

Other interested stakeholders should be present at the meeting. This should include: anyone responsible for releases e.g. a project manager; at least one technical functional manager such as a development or test manager or a more senior technical function manager; some people who can assess technical risk for example a technical or data architect, a usability professional, an operations and systems specialist and a business analyst. At Corbis the meeting was typically attended by one development manager, the manager of the analysis team and occasionally by the enterprise architect or the data architect. The development managers took it in turns to attend the meeting on a rotating schedule.

The cadence of prioritization meetings will affect the queue sizing in the kanban system and hence the overall lead time through the system. To maximize the agility of the team, it is recommended that the meetings be as frequent as reasonably possible. Weekly is a commonly recommended interval.

Some teams have evolved into demand driven prioritization rather than use a regular meeting. This is recommended only for more mature organizations where all stakeholders at the meeting can be available on-demand. In the Microsoft case study from chapter 3, the project manager created database triggers that alerted him to a free slot in the input queue. He would then instigate a prioritization discussion with the 4 product owners via email, alerting them that a slot was available for prioritization. An electronic discussion would ensue and a new item from the backlog would be selected. This process typically took 2 hours. By having this on-demand system rather than a weekly meeting the input queue size could be reduced with a subsequent improvement in cycle time through the system.

Release Planning Meetings

Release planning meetings happen specifically to plan releases. If releases are happening regularly with a cadence of say bi-weekly, then it makes sense to schedule the release planning activity to happen regularly. This reduces the coordination cost of holding the meeting and insures that everyone who needs to attend will have time available.

Release planning meetings are typically led by the person responsible for coordinating the release, usually a project manager. Any other interested party should be invited. This usually includes configuration management specialists, systems operation and network specialists, developers, testers, business analysts and for all of these folks their immediate supervisors or managers. Specialists are present for the technical knowledge and risk assessment capabilities. Managers are present so that decisions can be made.

A mature organization will have a checklist or framework for a release that facilitates planning. Things to be considered are: which items in the system are (or will be) ready for release; what is required to actually release them to production; what testing will be required post-release to validate the integrity of production systems; what risks are involved; how are these risks being mitigated; what contingency plans are required; who needs to be involved in the release and present during the push to production; how long will the release take; what other logistics will be involved.

The outcome should be completed template representing a release plan. With teams that are particularly sophisticated I have seen the release scripted as a sequence of procedures to be executed in a given order.

Completing the release plan may not be possible in a large meeting and some independent follow up work may be required on the part of the project manager.

Triage

Triage is a term borrowed from the medical profession where it refers to the practice of assessing and classifying emergency patients into categories for priority of attention. The system was first used in battlefield medical units where patients were separated into three categories: beyond help and likely to die soon; likely to live if given immediate treatment; likely to survive without immediate treatment. Emergency rooms use a similar system to prioritize patients arriving for treatment.

Triage was adopted into software development for classifying defects (bugs) during the stabilization phase of a traditional software project. Triage is used to classify bugs that will be fixed and with what urgency versus bugs that will not be fixed and allowed to escape into production when the product is released. A typical defect triage involves a test lead, supervisor or manager, a development lead, supervisor or manager and a product owner.

In kanban it still makes sense to triage defects. However, the most useful application of triage is to the backlog of items waiting to enter the system.

Backlog triage should be held at relatively infrequent intervals. Monthly, quarterly and di-annually are intervals that are popular with teams. The attendees at a backlog triage will typically be the same product owners or business representatives who attend the prioritization meeting, along with the project manager. The technical people typically do not attend in such large numbers. Perhaps one technical function middle manager will be present.

The purpose of the backlog triage is to go through each item on the backlog and decide whether it should remain in the backlog or be deleted. It is not to stack rank or provide any prioritization beyond the simple "keep or delete" choice.

Some teams have avoided the need for triage through automation and policy. The Microsoft XIT team from chapter 3 would delete any item older than six months, at a regular monthly interval. The reasoning given was that an item that had not been selected for the input queue in 6 months was unlikely to be of significant value and unlikely ever to be selected. If this changed, it was also likely that it would be requested again and hence nothing was lost by deleting it from the backlog.

The purpose of triaging the backlog is to reduce its size. The benefit of a smaller backlog is that it facilitates easier prioritization discussions. If there are 200 items in a backlog it will take significantly less time to pick winners at a prioritization meeting than if there are 2000 items in the backlog.

A good rule of thumb might be that if the backlog exceeds 3 months worth of work, that is, 3 months of delivery throughput, and all the items in the backlog could not enter the system within that time, then it would be a good to prune the backlog down in size. The appropriate size of the backlog will vary according to different markets and domains. Domains with high volatility will want a backlog sized to perhaps one month's worth of items. Domains with very low volatility might be able to hold a backlog with up to one year of items.

Hence, there is a relationship between the size of the backlog, the volatility of the domain in which the individual kanban systems is operating and the delivery velocity or throughput of the team. If a team delivers 20 user stories per month and the domain has some but not excessive volatility so that a 3 month backlog is desirable then the backlog should have approximately 60 items.

Issue Log Review and Escalation

When work items in the kanban system are impeded they will be marked as such and an issue work item created. The issue will remain open until the impediment is removed and the original work item can progress through the system. Reviewing open issues, therefore, becomes vital to improving flow through the system.

Issue log review should happen frequently and regularly. Again a regular cadence reduces coordination costs and insures that relevant stakeholders make time to attend. Very mature organizations may be able to dispense with regular meetings and hold on-demand meetings. This would be appropriate if there is a relatively small number of issues occurring and the increased coordination cost of on-demand meetings is actually less than the cost of holding a regularly scheduled meeting.

Issue log reviews should involve team members who have items blocked and the project manager. The main questions to be answered are: who is assigned to and working the issue; and, when is the expected resolution? Issues that are not progressing and are in themselves blocked or stale should be escalated to more senior management.

It may not be necessary to have senior management present at an issue log review but it is important to have clearly defined escalation paths and policies. When an issue is blocked the project manager should take responsibility and escalate the matter appropriately.

Issue management and escalation is typically done very badly even in agile development organizations. Resolving issues quickly, particularly issues that are external to the development team such as environment availability or ambiguous requirements or lack of test equipment improves flow and greatly enhances the productivity of the team and the value delivered. Issue management and escalation are core disciplines that provide a big return. Improving them should be a priority even for the most immature of teams.

Sticky Buddies

The concept of a Sticky Buddy was introduced at Corbis to resolve a coordination problem. There was a policy that allowed telecommuting at least one day per week, particularly for employees who lived further out of town. The policy dated back to an office move from Bellevue to Seattle, Washington several years earlier. Folks telecommuting were able to access the electronic tracking system, version control, build environment and so forth via VPN. So they were able to see work assigned to them, work on it, complete it and test it. They were able to update the electronic status of work, marking it complete and available to be pulled downstream.

However, they were not physically present in the office and able to move the sticky note on the card wall.

The solution to this was for each person to make a peer to peer agreement with someone who would be present in the office to act as their delegate. When the telecommuter completed an item and changed its electronic status, they would contact their sticky buddy by instant message, email or phone and ask them to update the physical board.

Sticky buddies also facilitated distributed development across different geographic locations. This was particularly important as the test team was in Chennai, India. There were also some specialist financial systems developers in Southern California.

Synchronizing across Geographic Locations

Synchronizing kanban teams across multiple geographic locations comes up again and again as an elementary question from folks considering implementing a kanban system. Often the questioner assumes that the early kanban implementations were done in a single geographic location and that I (and other early kanban advocates) had not considered the challenges of coordinating across geographically distributed teams.

Actually the opposite is true. The first team at Microsoft, from chapter 3, was actually located in Hyderabad, India with management and product owners located in Redmond, Washington. As just mentioned above, Corbis also had people in India and other locations outside Seattle including telecommuters.

The key to coordination across multiple sites is to use an electronic system. It isn't enough to only have a card wall. There are now a number of electronic card wall tracking tools available on the market. At the time of writing there are several including products from VersionOne, Thoughtworks and Borland, as well as available extensions or plug-ins for other products from Microsoft and Atlassian. Doubtless by the time you are reading this, the list will have grown and both the availability and acceptability of electronic card wall tracking tools will be well established.

In addition to electronic tracking, it will be necessary to keep physical card walls synchronized at the very least on a daily basis. It is important to assign someone to take responsibility for this at each location.

Some teams also coordinate stand-up meetings over the telephone or using a video meeting system. Prior to any standup meeting, video conference or telephone call, the local responsible person should take time to insure that the physical board is synchronized with the electronic system.

Takeaways

- ❖ Best practice is to use both a physical card wall and an electronic tracking system
- ❖ Kanban is possible across multiple geographic locations provided that an electronic tracking system is used
- ❖ Electronic systems that simulate the functionality of a physical card wall are available from a variety of vendors
- ❖ Holding regular meetings reduces the coordination cost for those meetings and improves attendance
- ❖ Prioritization and release planning should be done independently and should have independent cadence
- ❖ Daily standup meetings should be used to discuss issues, impediments and flow and do not typically follow the established pattern in other agile development methods
- ❖ Daily standups are an essential part of encouraging a culture of continuous improvement. Because the standup brings together the whole team briefly each day they provide an opportunity for all stakeholders to suggest and discuss improvement opportunities. The period immediately after the standup often develops into an informal process improvement discussion
- ❖ Grooming the backlog with a regular triage to reduce the size improves the effectiveness and efficiency of prioritization meetings
- ❖ Issue management, escalation and resolution is a core discipline in improving the performance of a team and should be addressed early in the development of the team
- ❖ Escalation paths and policies should be clearly defined

Chapter 8 Establishing a Delivery Cadence

In Chapter 5 we learned how to bootstrap a Kanban initiative and how Kanban involves striking a different type of bargain with the rest of the external stakeholders than is typical between a software development organization and its external partners. Part of this new type of bargain involves agreement and commitment to regular deliveries of working software.

The term "delivery cadence" in the title of this chapter, implies establishing a pattern of delivery of working software at a regular interval. For example, if we agreed to make a delivery every 2 weeks then we'd have a delivery cadence of bi-weekly or 26 times per year. Perhaps, we would even agree the day of delivery. For example, every second Wednesday as was the case for maintenance releases of IT systems at stock photography company Corbis in Seattle.

It is generally established in Agile software development circles that a regular cadence is important. Agile development methods achieve this with a time-boxed iteration typically of 1 week to 4 weeks in length. The argument for time-boxing is based on the notion that a steady "heart beat" to a project is important. There is an underlying assumption that in order to achieve this it is necessary to use strictly time-boxed iterations. At the start of the iteration a scope, or backlog is agreed upon and a commitment made. Work starts! Some amount of analysis, test planning, design, development, testing and refactoring is performed. If all goes well all of the committed scope is completed. The iteration ends with delivery of working software and a retrospective meeting to discuss future improvements and process adjustments. The cycle then begins again. All of this is happening at a regular cadence that has been agreed in advance – weekly, bi-weekly, monthly.

Kanban dispenses with the time-boxed iteration and instead decouples the activities of prioritization, development and delivery. The cadence of each is allowed to adjust to its own natural level. Kanban does not dispense with the notion of a regular cadence. Kanban teams still deliver software regularly with a preference for a short timescale. Kanban still delivers on the Principles Behind the Agile Manifesto[xix]. However, Kanban avoids any dysfunction introduced by artificially forcing things into time-boxes.

Over the last 10 years, teams using Agile methods have learned that less WIP is better than more. They've learned that small batch transfers are better than large ones. As a response to this learning they adopted shorter iteration lengths. Typical Scrum teams went from 4 weeks to 2 weeks and Extreme Programming teams from 2 weeks to 1 week. One of the problems this introduced is that it can be difficult to analyze work into small enough units to get it done in the available time window. The market responded to this by developing more sophisticated ways for analyzing and writing user stories. The object of which was to reduce the size of stories to make them more granular and reduce variability in the size in order to fit them into smaller iterations. While this approach is sound in theory it is hard to achieve. It falls into the category of the 5th element of the Recipe for Success: Reduce Variability and Improve the Process. As I explained in Chapter 4 any activity that reduces variability requires people to change their behavior and learn new skills. This means it is hard to do.

So teams have struggled to write user stories consistently small enough to fit into small time-boxed iterations. This has led to several dysfunctions. The first is to reverse the trend to smaller iterations and go back to larger ones. The alternative is to write stories that are focused on elements of the architecture or some technical decomposition of the requirements and have a story for the user interface, a story for the persistence layer and so on. A second alternative is to break the story across three iterations in phases where the first iterations performs analysis and perhaps test planning, the second involves developing the code and the third involves system testing and bug fixing. Any or all of these dysfunctions are possible. The latter two make a mockery of the notion of time-boxed iterations and disguise the fact that work is actually still in progress when it is being reported as completed.

So Kanban decouples the time it takes to create a user story from the delivery rate. While some work is complete and ready for delivery some other work will be in progress. Having decoupled cycle time for development from delivery cadence, it makes sense to question how often prioritization (and perhaps planning and estimation can happen.) It would seem unlikely that planning, estimation and prioritization discussions should need to happen at the same pace as delivery and software release. They are two completely different functions often requiring different groups of people to be involved. The coordination effort around delivery is surely different from the coordination effort around prioritization of new work. Kanban allows the decoupling of these activities.

So Kanban decouples the prioritization cadence from the cycle time to develop working code from the delivery cadence. In this chapter we will discuss the elements involved in agreeing a suitable delivery cadence and when or if it would make sense to have on-demand or ad hoc delivery rather a regularly scheduled delivery. Chapter 9 will discuss how to set a prioritization cadence and when or if it would make sense to have on-demand or ad hoc prioritization rather than a regular meeting. Chapter 11 will discuss how to set expectations around cycle time and how to communicate the contents of a release.

Coordination Costs of Delivery

Coordinating every software delivery has costs. It’s necessary to get people together to discuss the deployment, or release, manufacture, packaging, marketing, marketing communications, documentation, end user training, reseller training, help desk and technical support training, installation documentation, installation procedures, staff on-call, and on-site schedules during deployment and so on and on. Planning a release of a piece of working software can be incredibly complex depending on the nature of the business domain and the type of software. Upgrading a web site can be quite trivial in comparison to upgrading firmware deployed in military equipment spread across the globe, or satellites in orbit, fighter aircraft, or the nodes in a telephone network.

In 2002, when we were planning the release of the PCS Vision upgrade to the Sprint PCS cellular phone network in the United States 10s of thousands of people had to be trained. 17,000 retail staff working in stores across the country had to be trained in the features of the new network and the workings of the 15 or so new handsets being offered. A similar number of people had to be trained to answer the inevitable support calls that would ensue when the unsuspecting public took ownership of their new devices. Just planning the training for around 30,000 people is a major cost in both money and time.

So it is important to understand the coordination costs in making a delivery: How many meetings?; how many people?; how much time will it consume?; and what opportunity cost is incurred while you distract people away from their regular activities? For example, if software developers have to attend release coordination meetings, is this distracting them from actually building the software for the release?

Transaction Costs of Delivery

With physical goods it is quite easy to understand the transaction costs of making a delivery. First there is payment. The customer will arrange to pay the supplier with some monetary instrument, for example a credit card. For the pleasure of taking payment via credit card, the leading vendors such as Mastercard and Visa charge the vendor a transaction cost of typically 2%-4% of the value of the transaction.

In addition to any transaction costs on the financial transaction between the consumer and vendor, there may also be delivery charges. Delivery costs money but it also takes time and involves manpower. There may also be installation charges. For example, you buy a washing machine from Sears. You arrange for delivery on a given day. The arranging delivery and coordinating with the driver that the correct model washing machine is being delivered to the correct house at the correct time is a coordination cost of delivery. The driver actually picking up the machine at the warehouse, driving it to your home and unpacking it for you is a transaction cost. Perhaps the same person, or another person, a plumber will have to install it for you. The plumber will take time to drive to your home, and yet more time to make the installation. For all of this time and effort there will be a charge.

The delivery and installation costs are also transaction costs of buying a washing machine.

Economically, the retailer absorbs the cost of the credit card transaction. The other transaction costs for delivery and installation are often passed onto the consumer. Not all of the transaction costs are "seen" or "felt" by all the players in the value chain but they affect the economic performance of the system as a whole. The net effect of all these costs is to inflate the final price paid by the consumer without actually increasing the value delivered.

It is true that the washing machine without delivery or installation is of little value but its value added capability is washing clothes. The delivery and installation are non-value-added activities that should be counted as transaction costs.

In software development the transaction costs of delivery can also be physical in nature. Some firms such as Microsoft still "release to manufacture" (RTM) and print physical media such as DVDs and box them and ship those boxes to distributors, retailers and other partners. With embedded software it may be necessary to manufacture a set of chips or at the least to "blow" the software code into firmware using technology like EE-PROM. The chips then need to be physically mounted into the hardware that they control.

In other cases, it may be possible to do an electronic deployment. For example, cell phones now permit what is called over-the-air device management to upgrade the firmware and device settings. Many satellites and space probes allow firmware to be upgraded over-the-air. This soft deployment capability makes space missions much more agile than they were in the past. The mission can be changed by uploading new software. Defects can also be fixed in-situ. Some famous defects such as the focusing on the Hubble Telescope were (partially) corrected with software changes. This has changed the economics of deployment.

For many people reading this you may be involved in web development or internal application development. Deployment may involve simply copying files across to a set of disks on other machines. While this sounds trivial it often isn't. Often it is necessary to plan an elaborate procedure to gracefully switch off databases, application servers, and other systems and then upgrade them and bring them all back again. One of the biggest issues is data migration from one generation of a database schema to another. Databases can get very large. The process of serializing the data to a file, parsing it, unpacking it, perhaps embellishing or augmenting it with other data then re-parsing it and unpacking it into a new schema can take hours – perhaps even days.

In some environments software deployment can take hours or days. This is often not because the software is of poor quality or poorly architected, it simply reflects the nature of the domain in which the software is used. All the activities involved in successfully delivering software whether it is packaged applications, embedded firmware or IT applications running on internal servers, need to be accounted for, planned, scheduled, resourced and then actually performed. All of these activities are transaction costs of making a delivery.

Efficiency of Delivery

The equation to calculate the efficiency of delivery can be assessed in two ways. The simpler way is to look at the labor and costs involved. The more complex method is to consider the value delivered.

First the cost only model. We must consider the total costs incurred between releases. Often this is a known amount – the burn rate of the organization. If we release once per month and our burn rate is $1.3 Million per month then our costs are at least $1.3 Million per release. We may also incur physical manufacturing costs, printing costs, advertising costs, and out of pocket expenditure to coordinate the release. All of this is relatively easily accounted for. Let's imagine it is $200,000 in this case. So our total cost of release is $1.5 Million.

We know that our additional out of pocket delivery costs are $200,000 but how much of the $1.3 Million was also spent planning, coordinating and actually performing the delivery. If we have suitable time-tracking data available we might be able to calculate this. Even if we don't we could make a good guess. How many meetings? How many people? How many hours spent in meetings? Include the number of man hours for actual deployment or delivery activities. Multiply by the hourly rate. If this added to $300,000 then we'd have a transaction and coordination cost of $500,000 for a delivery.

Delivery Efficiency% = 100% x (Total Cost – (Coordination Cost + Transaction Cost)) / Total Cost of Software Release

In this example we have an efficiency% = 100% x ($1,500,000 - $500,000) / $1,500,000 = 66.7%

To be more efficient we have to (a) increase the time between deliveries, or (b) reduce the coordination and transaction costs. Choice (a) is the typical choice of 20th Century Western business. It is the choice of "economy of scale." Do things in larger batches in order to amortize the costs over the larger scale. Choice (b) is the typical choice of late 20th Century Japanese businesses and businesses pursuing Lean Thinking. Choice (b) focuses on reduction of waste through the reduction of the coordination and transaction costs in order to make the batch size efficient – in this case the time between releases efficient.

How much efficiency is enough?

This is really an open question. Each business will have separate views on suitable numbers for efficiency and a lot will depend on the value to be delivered.

Agreeing a Delivery Cadence

If we understood how much value was to be derived from a given release then we could make a better choice about how frequently to deliver. If our monthly delivery of software was to realize $2 Million in revenue against our costs of $1.5 Million then we know that we are making $500,000 in margin from the activity. We could re-write our efficiency equation as

Delivery Efficiency% = 100% x (Transaction Costs + Coordination Costs) / (Margin + Transaction Costs + Coordination Costs)

For our working example, this would produce a delivery efficiency % = 100% x $500,000 / ($500,000 + $500,000) = 50%

Now this gets very complicated because calculating the true value of a delivery can be almost impossible. We may not have firm orders at firm prices. We may be speculating on the uptake in the market and the price and margin we can achieve. We may be releasing items of intangible value such as a revision to our brand identity and marketing materials or a set of usability and bug fixes to our product or web site.

Calculating whether or not we should delay and release less often to be more efficient is equally difficult. Increasing our time to market may have an adverse effect on our market share, price and margin. So this concept of delivery efficiency is not an exact science. What is most important is that you, the team and the organization are aware of the costs of making a delivery in both time and money and are capable of making some form of rational assessment about the acceptable frequency of delivery.

If it takes 10 people 3 days to make a successful code delivery from a team of 50 people, is it acceptable to make a release every 10 working days (or 2 calendar weeks)? The answer is probably not. Perhaps once per month or 20 working days is a better choice. On the other hand, the market might be one where agility and time to market is vital, where a lot of risk can be mitigated through more frequent releases and the cost is worth paying. You need to decide for yourself.

Remember the golden rule of Kanban. Your aim is to change as little as possible and to strain the system just enough to provoke the next incremental change.

Improve Efficiency to Increase Delivery Cadence

To follow the example, we have determined it takes 10 people 3 days to release the code. From this we have derived that a monthly release is acceptable. However, several folks believe that with improved code quality, improved configuration management, better tooling to handle data migration and regular rehearsals of the deployment procedures it will be possible to cut the 3 day time down to 8 hours. Suddenly a release every 2 weeks looks viable. Perhaps every week is possible? What should you do?

My advice is to choose conservatively initially. Agree to a release every month. Let the organization prove that it can achieve this level of consistency. After a month or two reflect up on the code quality and instigate a program to improve configuration management. If slack resources are available, engage them to create tooling to improve data migration across schemas during a release. And finally, encourage the team to rehearse releases using a staging environment. Perhaps you'll need to buy, install and commission such an environment. All of this will take time.

Challenge the team and the immediate function managers who control and perform releases to reduce the transaction and coordination costs. As these costs come down, review the progress at the operations review meeting and liaise with other stakeholders. When you feel confident that you can make a commitment to a more frequent delivery cadence such as bi-weekly, then do so.

Reducing coordination and transaction costs is at the heart of Lean. It is waste elimination is its most potent form. It allows smaller batches to become efficient. It enables business agility. Reducing coordination and transaction costs is game changing. Do not simply focus on reducing them. Reduce them with a goal in mind. Reduce them with the goal to make more frequent deliveries of working software and deliver more value to your customers, more often.

Making On-Demand or Ad Hoc Deliveries

Regular delivery has advantages. Making a promise to delivery on specific dates, for example, every second Wednesday, allows those involved to schedule around it. It provides certainty. It can also reduce coordination costs because there is no overhead involved in deciding when to make a delivery and who needs to be involved – all of that is established once and is fairly consistent from then on.

Regular delivery also helps to build trust. This works at a very deep neurological level within the brains of those involved. Predictor neurons in the brain are programmed to expect a delivery of good quality working code. If the prediction comes true then dopamine is released, the person feels satisfied and happy and the brain learns to expect a little more reliability from future predictions. In a related mechanism, oxytocin is released from the realization of expectations. Oxytocin is the brain's trust chemical while dopamine is the brain's pleasure chemical as well as a key part of its learning and predictive feedback mechanism. The net result is that frequent deliveries made on time will be both trust building and addictive. Though I imagine this is a very mild form of addiction in comparison to the affect of drugs, smoking, alcohol, gambling or love, it is nevertheless a form of addiction. The flipside to all of this is that a failure to meet a delivery will result in a loss of trust and a bad behavioral reaction. I think we all understand what it's like to be addicted to something: candy; television; alcohol; gambling; drugs; love; golf; driving fast; the list could be endless. And we know how it feels when we are denied our hit of our favorite addiction. How do you feel when the TV Network reschedules your favorite show and you turn up in front of the screen at 8pm Tuesday expecting to get your regular fix of your favorite drama and pow! - an announcement that the show is on hiatus for 3 weeks and is replaced with either a repeat of an older episode or perhaps a sports event or live telecast? So imagine how your value stream partners and customers will feel if you let them down with a late or cancelled delivery!

Having made a strong case for a regular delivery cadence, it does make sense to have on-demand or ad hoc deliveries in some circumstances. What might those circumstances be?

Firstly, on-demand or ad hoc delivery makes sense when the coordination costs of the release activity are small. When coordination costs are low then there is no benefit in scheduling coordination activities regularly. Secondly, it makes sense when the transaction costs are low, perhaps because the deployment of the code is largely automated and the quality is assured in advance prior to deployment. And finally it makes sense in environments when the deployments are so frequent that there is no real need for a pattern to be developed. New software is being delivered so often as to appear continuous to most observers and external stakeholders. Their brains haven't been programmed to anticipate a delivery at any given point in time so their predictor neurons don't fire and they don't become addicted in the way described earlier. As a result they don't suffer the same negative feelings if you miss a delivery. When there is no expectation, there can be no disappointment.

This type of near continuous deployment of code seems to be useful and necessary in some industries. The examples that have emerged with early Kanban adopters have mostly been in the media industry, for example, IPC Media in London where they use multiple Kanban systems to plan development for online media properties such as mousebreaker.com, a highly addictive online game.

The first two circumstances of low coordination and low transaction costs would tend to indicate higher maturity organizations. This has also been observed with early adopters of Kanban. The Microsoft XIT department involved a CMMI ML5 vendor in India and Microsoft IT group in Redmond Washington which is approximately CMMI ML3 in maturity. High maturity organizations tend to have established levels of trust with value chain partners and external stakeholders including senior management. So they don't need a regular delivery cadence in order to build that trust.

So in general my advice is to choose a regular delivery cadence except in circumstances where trust already exists, high levels of capability and maturity already exist and in domains where near continuous deployment is desirable. Where deployment is so frequent as to seem continuous and avoids the neurological trap of addicting customers to regular anticipated deliveries, there is no need for a regular delivery cadence.

There is one final circumstance where it is acceptable to make an on-demand delivery. That is in the case of an urgent request that is being treated as a special case and expedited. This concept of an Expedite class of service is explained in Chapter 11. We might choose to expedite for several reasons. The first and most obvious is in the event of a critical production defect. In the circumstances where nothing else matters but to fix the problem then an off-cycle release should be planned.

There are other circumstances where an off-cycle release might make sense. Perhaps the sales team have taken an order from a big customer who wants a customized version of the software and due to budget restraints and the fiscal cycle they need to take delivery before the end of the month (and the quarter.) The order comes through from the operations group that software engineering has to drop everything to fulfill this one customer order because it is worth a great deal in revenue. The Lean Thinking behind this reasoning and the framework to determine whether expediting a request is the correct choice is discussed in Chapter 20. For now we can assume that a request to expedite a delivery for a specific customer is the correct decision.

Under these circumstances, it makes sense to plan a special off-cycle release. This release should be treated as exceptional and the regular release cadence should be re-established as soon as possible following the exceptional release. It does pay to use some common sense. For example, if a regular release is scheduled for Wednesday and the exceptional release is required on Friday of the same week, it may make sense to delay the Wednesday release until Friday. If you do choose to do this, it is important to communicate it properly and sufficiently early so that expectations are reset. You don't want to lose trust with your value chain partners as a side-effect of trying to be accommodating and helpful.

Takeaways

- "Delivery cadence" means an agreed regular interval between delivery of working software
- Kanban decouples delivery cadence, from development lead time, and prioritization cadence
- Short time-boxed iterations have led to dysfunction with some teams attempting to adopt Agile development methods
- Delivery or release of software involves coordination of many people from various functions. All of this coordination has a cost. This cost can be measured
- Delivery or release of software carries with it a set of transaction costs in both time and money. These costs can be determined and tracked
- Efficiency of delivery can be calculated by comparing the sum of the transaction and coordination costs of making a delivery against the total cost (or burn rate) of creating the software for that delivery
- Delivery cadence can be established by comparing the cost of making a release against the value that release will enable
- Efficiency and cadence can be increased by focusing on reducing the transaction and coordination costs
- Regular delivery builds trust
- Setting an expectation of regular delivery and then delivering consistently against that expectation can be addictive
- Scheduling regular deliveries reduces coordination costs
- Ad hoc or on-demand delivery can make sense for high maturity organizations with established high levels of trust and low transaction and coordination costs of delivery
- Legitimate requests for expedited delivery may also be cause for an off-cycle release. Regular release should be re-established as soon as possible after such a special exceptional release

Chapter 9 Establishing an Input Cadence

In this chapter we will discuss the elements involved in agreeing a suitable prioritization cadence and when or if it would make sense to have on-demand or ad hoc prioritization rather a regularly scheduled prioritization meeting.

Coordination Costs of Prioritization

When we were introducing Kanban at Corbis in 2006, we chose to start with the sustaining engineering effort that handled minor upgrade requests and production bug fixes for the full suite of IT systems including functions such as finance and human resources as well as the more business specific systems of the digital asset management system and the e-commerce web site. These systems served at least 6 business units including sales, marketing, sales operations, finance, and the function that supported the supply chain for ingest of digital photography, meta-data tagging, cataloging and fulfillment essentially the business' supply chain.

6 departments competed for the shared resource available to make these small changes and upgrades. When the system was first introduced a business case had been made for a sustaining engineering function that could provide frequent, tactical releases enabling some limited business agility while other new IT system functionality was built using a traditional program management office style governance function with a portfolio of projects each of which was justified and authorized based on its own independent business case. The sustaining function had been approved by the executive committee and an additional 10% funding for software engineering authorized. This funding turned into an additional 5 headcount for the department. This new process was named the Rapid Response Team (or RRT). The name was a complete misnomer.

It wasn't feasible to create a specialist maintenance department with those 5 new people. Corbis had a considerable diversity of IT systems and many required specialist skills. Particularly the analyst function that developed and elaborated system requirements relied heavily on specialists. So the additional 5 heads were spread somewhat evenly across the software engineering function that included project management, system analysis, development, test, configuration management and build engineering functions. So there was no team as such. The T is RRT was meaningless. The challenge for management was to show that this additional 10% of resources was being spent on maintenance and sustaining work and wasn't simply being absorbed into major project portfolio work.

It was decided to dedicate a project manager to the sustaining engineering effort. While this lady was not full time on sustaining engineering, she did provide a single focal point for communication and coordination and counted as 0.5 of the 5 headcount allocated to the initiative. It was also decided to dedicate a build engineer from the configuration management team to the initiative. His duties were to maintain the pre-production systems required for test and staging and to build code and push it into the test environment when required.

Corbis had a policy that only build engineers were permitted to promote code from the development environment to the test environment in order to maintain the integrity of the shared test environment which was being used by multiple projects all at once. This policy was to change later but in September 2006 the reality was that a build engineer was required to promote code for testing.

Prior to introducing Kanban, the coordination effort required to agree a release scope for a maintenance release was prohibitive. The project manager and often her boss the group project manager would convene a meeting of all the relevant parties including business analysts, business representatives, system analysts, development managers, test leads and the build engineer and sometimes the configuration management manager, as well as system operations and help desk personnel. These meetings could take several hours and were often inconclusive. Team members would be sent off to do estimates and another meeting convened. The later meeting would often become bogged down in debate over priorities and again was often inconclusive. In September 2006 it was taking 2 weeks of calendar time with several meetings lasting many hours to agree the scope for a release that was supposed to take only 2 weeks to build and deploy. Because of the 2 week iteration length only very small requests could be accommodated and many potentially valuable requests were ignored. These requests would have to be rerouted to a major project and would likely take months or years before being implemented. So the system was neither rapid nor responsive initially. So the RR in RRT was also meaningless.

Kanban freed this team from all these dysfunctions and rightly earned the initiative the name Rapid Response Team.

When introducing the kanban system, the business owners were educated on the workflow, the input queue and the pull mechanism. It was explained to them that they would be asked to simply replenish empty slots in the queue and that it would not be necessary to prioritize the backlog of requests. If there were two slots free in the queue then the question would be "Which 2 new items would you like next?" Assuming we had data on the average cycle time and the due date performance against the cycle time target in the service level agreement, then the question might more elaborately become, "Which 2 items would you most like delivered 30 days from now?" The challenge then would be for 6 competing business owners to somehow agree to choose 2 items from the many possible choices.

Nevertheless, the question is a simple one and it was suggested that answering such a simple question should easily be achievable in one hour. There was a consensus that one hour was reasonable and hence the business owners were asked if they would give up one hour of their week to attend a weekly prioritization meeting to refill the input queue for the sustaining engineering function.

Agreeing a Prioritization Cadence

The meeting was scheduled to happen every Monday morning at 10am. It was generally attended by higher ranking business managers usually with Vice President titles from each function. In addition the meeting contained the project manager, the Senior Director for Software Engineering and the Senior Director for IT Services (included the project management function), at least one development manager, the test manager, the analysis team manager and occasionally some additional individual contributors.

Agreeing a regular cadence provided everyone with predictability. They were able to set aside the 1 hour at 10am on Monday mornings and attendance at the meeting was generally good.

Weekly is a good choice of prioritization cadence. It provides frequent interaction with business owners. It builds trust through the interaction involved. In the collaborative cooperative game of software development it enables the players to move the pieces once per week. Weekly meetings are facilitated by the simplicity of the question to be answered and the guarantee that the meeting can be completed in one hour. When asking business people to give up their time away from running their business, they need to see that time is well spent.

Many things with kanban contribute to making the weekly prioritization meeting a gratifying experience: it's a collaborative experience; there is transparency on to the work and the work flow; progress can be reported every week; everyone gets to feel that they are contributing to something valuable. To several of the vice presidents at Corbis it felt like the RRT process was making a difference. They gained a new level of respect for the IT department and they learned to collaborate with the peers in other departments in a way that hadn't been common previously at Corbis.

Efficiency of Prioritization

Weekly coordination meetings may not be the right answer for your organization. You may find that your coordination challenges are harder or simpler than those at Corbis. Some teams all sit together, so there is no need for a meeting. Prioritization coordination amounts to a quick discussion across the desk. On the other hand, some teams may have people in multiple time zones and several different continents, so weekly meetings may not be so easily scheduled. Perhaps, the question to be answered won't be so simple and the meeting will take longer. It's hard to imagine a situation with more than 6 groups competing for the same shared resource but it is possible. The more groups involved the longer the meeting is likely to take. The longer the meeting, the less frequently you are likely to hold it.

As general advice, more frequent prioritization is desirable. It allows the input queue to be smaller and as a result the waste in the system is less. WIP is lower and the cycle time will be shorter. More frequent prioritization allows all parties to work together more often. This builds trust through the collaborative working experience and improves the culture. So strive to find the smallest most efficient coordination scheme possible and hold prioritization meetings as often as is reasonable.

Transaction Costs of Prioritization

In order to facilitate an efficient meeting every Monday, Diana, the project manager would generally send an email on Thursday or Friday prior and inform attendees of the estimated number of free slots in the queue anticipated by Monday morning. They were asked to browse through the backlog of requests and pick out likely candidates for selection on Monday. This "homework" often prompted them to prepare some argument to support the successful selection of their favorite item. We began to see supporting documents at the Monday meeting. Some people would prepare a business case, or some presentation to support their choice. Others began to lobby each other. It was likely that someone on the prioritization board would take another to lunch on Friday specifically with the intent to garner support for their choice at the Monday meeting. Horse trading was introduced where board members might agree to support another's item this week in exchange for support for one of their items in a future week. The nature of the rules of the game, where multiple organizations competed for the shared RRT resource introduced a whole new level of collaboration.

Occasionally, the business might be concerned that a request was too costly to implement in comparison to its worth. So they might solicit the analysis team to make an estimate. Later rules regarding class of service were introduced to guide whether it was worthwhile estimating an item or not. This is fully explained in chapter 11.

All of these activities, including estimation, business plan preparation and candidate selection from the backlog are preparatory work for prioritization. Generically, in economic terms these are the transaction costs of prioritization. It is desirable to keep these costs low. If the transaction costs become onerous then regardless of how low the coordination costs are, the team will not want to meet regularly. By avoiding detailed estimates as much as possible the transaction costs are lowered. This facilitates more frequent prioritization meetings.

Improve Efficiency to Increase Prioritization Cadence

In general, the management team must be aware of all of the transaction and coordination costs incurred by everyone, not just the development team, involved in prioritization and selection of new items to queue for development and delivery.

Many Agile methods use a form of prioritization called Planning Poker that uses a 'wisdom of the crowds' technique where every team member gets to vote using a card containing a sizing number. The votes are averaged or sometimes a consensus is sought by discussing outliers in the voting and then revoting until everyone on the team agrees the estimate. The poker cards often use a non-linear numbering scale such as the Fibonacci Series.

It is argued that this planning technique which is also a form of a collaborative cooperative game is highly efficient as it allows a fairly accurate estimate to be established quickly. There is anecdotal evidence to support this but equally there is evidence to suggest that group think is also possible. I've heard reports of teams such as a startup in San Francisco where they consistently underestimate despite using a transparent collaborative game such as Planning Poker and I've heard reports from senior managers at a well known travel booking web site where they report that teams consistently over-estimate. Whether or not you believe that these planning games are effective or not, the argument that they are efficient is worth considering more deeply.

It is true that planning games involving the whole team can create an estimate for an individual item such as a user story very quickly. However, the exercise involves the whole team. There is a significant coordination cost to this. It will work effectively on small teams focused on single products. However, if we extrapolate the technique to an organization like Corbis where we are doing maintenance on 27 IT systems using 55 people, many of whom are specialists in one field, domain, system or technology, then it would be necessary to have almost all 55 people in the meeting to make an estimate. The transaction costs of planning and estimation may be small but the coordination costs are high.

In general, these Agile planning methods are efficient only for small teams focused on single systems and product lines due to this coordination cost effect.

By choosing to eliminate estimation for several classes of service of work item both the transaction costs and the coordination costs of prioritization are reduced. This facilitates much more frequent prioritization meetings because the meetings remain efficient. This has enabled kanban teams to make ad hoc or on-demand prioritization.

Making On-Demand or Ad Hoc Prioritization

As described in Chapter 3, in 2004, Dragos Dumitriu introduced a kanban system with his XIT Sustaining Engineering team at Microsoft. The upstream business partners were 4 product managers who represented several business units. They focused and prioritized change requests for the 80 or so IT systems supported by XIT.

When Dragos and I designed the kanban system for introduction with XIT, we designed an input queue big enough to cope with at least one week of throughput. Despite the fact that all 4 business representatives were based in Redmond, Washington on the Microsoft campus along with Dragos, the prioritization meeting would take place by phone. The Microsoft campus is huge. Buildings number into the hundreds though there are actually only about 40 buildings in total. The area covered is several square miles and transport between sections of the campus is by minibus or Toyota Prius. Many "softies" prefer conference calls for coordination meetings rather than face-to-face. This has a negative impact on the level of trust and social capital in their workforce but it facilitates efficient working.

So Dragos established a weekly phone call to prioritize new change requests into their backlog. The four product managers represented business units that provided funding via inter-company budget transfer to fund Dragos' team. Based on this funding, it was possible to determine roughly how many times someone should get to pick an item from the backlog. The product manager who provided six tenths of the funding would get to pick 3 from every 5 opportunities. Others would get to select items in a similar way based on their level of funding. The product manager who provided lowest funding would get to pick roughly once in every 10 times. We might describe this as a weighted round-robin method of selection.

So the rules of the XIT prioritization collaborative cooperative game were simple. Each week the product managers would refill the open slots in the input queue – typically 3 slots. Each of them would get to choose based on their position in the round-robin queue. The target cycle time in the service level agreement was 25 days. So if they got a chance to choose a change request for development then they were asking themselves, "Which of the items in my backlog do I most want delivered 25 days from now?" The order in which they got to choose was very clear and simple based on their level of funding for the department.

Due to the simple nature of these rules, the meeting was over very quickly. It became clear that a coordinating phone call really wasn't necessary. Dragos had the Microsoft Product Studio (a forerunner to Visual Studio Team System, Team Foundation Server) database provide an email from a trigger indicating when a slot became free. He would then forward that email to the 4 product managers. They would quickly agree whose turn it was to choose an item and that person would select something. Typically, an empty slot in the queue was replenished within 2 hours.

The exceptionally low coordination costs, coupled to the low transaction costs related to the decision not to estimate change requests, along with the relatively high maturity of the team involved allowed Microsoft XIT to dispense with regularly scheduled prioritization meetings.

It is worth noting that Microsoft in Redmond is roughly the equivalent of a CMMI ML3 organization and the vendor used for XIT development and testing was a CMMI ML5 team based in Hyderabad, India. So this team had the advantage of low coordination costs based on simple explicit prioritization policies, low transaction costs due to a policy not to estimate any requests, and particularly high levels of organizational maturity. The net effect of all three of these meant that on-demand prioritization meetings made the team more effective.

As a general rule, you should choose ad hoc or on-demand prioritization when you have a relatively high level of organizational maturity, low transaction costs and low coordination costs of prioritization. Otherwise, it is better to use a regularly scheduled prioritization meeting and coordinate selection of input queue items with a regular cadence.

Takeaways

- "Prioritaztion cadence" means an agreed regular interval between meeting to prioritize new work into the input queue for development
- Kanban removes potential dysfunction around the coordination of iteration planning in Agile methods by decoupling the prioritization cadence from the development lead time, and delivery
- Prioritization of new work request such as user stories involves coordination of many people from various functions. All of this coordination has a cost. This cost can be measured.
- Estimation to facilitate prioritization decisions represents the transaction costs in both time and money involved in prioritization. These costs can be determined and tracked
- Policies concerning the method of prioritization and the inputs for decision making represent the rules of the collaborative cooperative game of prioritizing kanban software development
- Planning games used in Agile methods do not scale easily and can represent a significant coordination cost on larger teams with broader focus than a single product line
- Prioritization cadence can be established by encouraging those involved in prioritization decision making to meet as regularly as reasonable based on the transaction and coordination costs involved.
- Efficiency and cadence of prioritization can be increased by focusing on reducing the transaction and coordination costs
- Frequent prioritization meetings build trust
- Scheduling regular prioritization meeting reduces coordination costs and is particularly useful in lower maturity organizations
- Ad hoc or on-demand prioritization can make sense for high maturity organizations with established high levels of trust and low transaction and coordination costs associated with the policies for prioritization decision making

Chapter 10 Setting Work-in-Progress Limits

As we learned in Chapter 1, one of the two fundamental principles behind Kanban is that it limits work-in-progress, the other being that work is pulled when capacity is available via a card passing signaling mechanism. So it's true to say that one of the most important decisions you'll make when introducing Kanban is establishing the correct limits for work-in-progress throughout the workflow.

In Chapter 5 we learned that the work-in-progress limits should be agreed by consensus with upstream and downstream stakeholders and more senior management. It is true that limits could be unilaterally declared. However, there is power in gaining a consensus and obtaining a commitment from external stakeholders regarding the WIP limit policy. When your team and process is put under stress you can fall back on that agreement to maintain the WIP limit discipline and avoid having to override or abandon your WIP limit policy.

Limits for Work Tasks

At Microsoft with the XIT team, Dragos Dumitriu decided that developers and testers should work on a single item at a time. There would be no multi-tasking. This was unilaterally declared but fortunately this choice did not prove problematic with other stakeholders. The organization was mature enough to maintain disciple and follow the process that was agreed. You may recall that at the beginning in fall of 2004 there were 3 developer and 3 testers in the team. So the WIP limit for development was 3, and also 3 for test.

At Corbis with the sustaining engineering activity processing small change requests and production bug fixes, we made a similar decision, that analysts, developers and testers should general only work on one customer valued work item at a time. With major new projects we tended to make different decisions. There was more collaborative working. Perhaps teams of two or three people would work on a single item, and those items may become blocked or delayed, so we speculated it might make sense to allow some task switching and some additional parallelism with WIP.

There has been some research and empirical observation to suggest that two items in progress per knowledge worker tends to be optimal. This result is often quoted to justify multi-tasking. However, I believe that this research tends to reflect working reality in the organizations observed. There are a lot of impediments and reasons for work to become delayed. The research does not report the organizational maturity of the organizations studied, or correlate the data against the number of external issues (assignable cause variations) occurring. Hence, the result may be a result of the environments studied and not indeed an ideal number. Nevertheless, you may find resistance to the notion that one item per person, pair or small team is the correct choice. The argument may be made that such a policy is too restricting. In this case, choosing a WIP limit of two items per person, pair or team is reasonable. There may even be cases where a limit of three per person, pair or team is acceptable. In the case of our example from Chapter 5 with a development manager in Copenhagen where his team had 7.5 items in progress per person, reducing this to 3 per person would represent a massive improvement.

It is also acceptable to come up with a fractional outcome. For example, a team of 5 developers might agree that 8 items in progress is a good number – slightly less than two per person. They are attempting to achieve a single-tasking, focused workflow but recognize that their work suffers from variability, so rather than choose 5 as their limit they opt for a slightly larger number.

There is no magic formula for your choice. What is important to remember is that the number can be empirically adjusted. You can select a number and then observe whether it is working well. If not then adjust it up or down.

Limits for Queues

When work is completed and waiting to be pulled by the next stage in your workflow it is queuing. How big should these queues be? The answer is as small as possible. The WIP limit for a queue is often bracketed with its preceding work step. For example development and the development complete queue will be bracketed together. If a really tight policy on work task WIP was established such as strictly one item per person, pair or small team then it will be necessary to have some queue to absorb variation and maintain flow. If your Kanban system in operation suffers from stop/go behavior causing workers to be idle simply due to variability in the length tasks take to complete then you may need to increase queue sizes. However, if you already made a choice to have, for example, 2 items in progress per person, pair or team then you already have buffered for variability so your queue size can be effectively zero. Simply bracket the work task column and the complete queue together.

Buffer Bottlenecks

The bottleneck in your workflow may require a buffer in front of it. This is a typical bottleneck exploitation mechanism explained in Chapter 16. The sizing of the buffer is important. Again you want it to be as small as possible. Buffers and queues add WIP to our system and have the effect of lengthening the cycle time. However, buffers and queues smooth flow and improve the predictability of cycle time. They also insure that people are kept working and provide for greater utilization. A balance needs to be struck. In many instances you are looking for business agility through shorter cycle times, and higher quality partly through lower work-in-progress. However, you cannot sacrifice predictability in order to achieve agility or quality. If your queue and buffer sizes are too small and your system suffers a lot of stop/go behavior due to variability then your cycle times will be unpredictable with a wide spread of variability. The key to choosing a WIP limit for a buffer is that it must be large enough to insure smooth flow in the system and avoid idle time in the bottleneck. More detail on buffer sizing and how to design buffers for capacity constrained and non-instant availability bottlenecks is discussed in Chapter 16.

Input Queue Size

The size of the input queue can be directly determined from the prioritization cadence and the throughput or production rate in the system. For example, if a team is producing 5 completed work items per week on average with a range of 4 to 7 items typically, and the prioritization cadence is weekly then the queue size should probably be set to 7. Again this can be empirically adjusted. If you run your system for several months and the queue is never totally depleted before your prioritization meeting occurs then it's probably too large, so reduce it by one and observe the results. Repeat until you have a prioritization meeting where you are asking the business representatives to refill the entire queue.

If on the other hand, you have a weekly prioritization meeting on a Monday but the queue was depleted by Thursday afternoon and some of the team was idle as a result then your queue is too small. Increase the queue size by one and observe for a few more weeks.

Queue and buffer sizes should be empirically adjusted as required. Hence, do not fret over a decision to establish a WIP limit. Do not delay rollout of your Kanban system because you can't agree the perfect WIP limit numbers. Choose something! Choose to make progress with imperfect information and then observe and adjust. Kanban is an empirical process.

What size should the input queue be, if you are using on-demand prioritization? You may recall from Chapter 3 that the XIT team had an input queue of 5 items. This was designed to be large enough to absorb one week of throughput. It was based on the assumption that the prioritization meeting was happening weekly. However, very quickly the product managers decided that the meeting was unnecessary and it was acceptable to make event-driven decisions when a slot in the queue became free. Once this happened, I should have advised Dragos to reduce the input from 5 to only 1. It's a reflection of my inexperience at the time that I didn't do this. The system had changed. The assumptions on which it was designed had changed. The input queue size policy was based on those assumptions and should have been revisited. Had we done so, the cycle time improvements would have been even more impressive.

When XIT switched to on-demand prioritization it took them typically 2 hours to refill an empty slot in the queue. It would have been acceptable to assume that the longest time to reset the queue would have been 4 hours. However, the developers were not collocated with the product managers. The prioritization decision makers were in Redmond, Washington and the developers in Hyderabad. Each in turn were working (officially) 8 hour days at opposite ends of the day. So it is likely that there might be occasions when the Indians turn up for work in the morning finish off a task and need their queue replenished but the product managers are safely asleep in bed. Given this non-instant availability problem we should probably allow 16 hours to replenish a single item in the queue under extreme circumstances. Remember that the developers were the bottleneck in this workflow. In order to maximize throughput we never want those developers to be idle. So we need to be conservative. 16 hours is conservative when the average queue refill decision only takes 2 hours. So what would the throughput be in an average 16 hour period? At peak performance the team achieved 56 items in a single quarter. That's less than 5 per week. So in a 16 hour period it is unlikely they will complete even a single item. So a queue size of 1 is perfectly acceptable. No queue at all would be unacceptable. There is still some chance that the team would suffer idle time when they finish an item during the 16 hour window when the product managers may be unavailable to refill the queue.

Unlimited Sections of Workflow

In the Theory of Constraints pull system solution for flow problems known as, Drum-Buffer-Rope, all work stations downstream from the bottleneck have unlimited WIP. This design is based on the assumption that they have a capability of greater throughput than the bottleneck and have slack capacity resulting in idle time. As a result there is no need for a WIP limit.

[Insert stick man diagram]

With Kanban most or all stations in the workflow have WIP limits. This has a potential advantage because impediments, that unexpected, unanticipated variability, may cause an upstream step to become a temporary bottleneck. The local WIP limit with Kanban will stop the line quickly avoiding the system clogging and becoming overloaded. When the impediment is removed, the system will then restart gracefully.

[Insert stick man diagram]

However, it may be acceptable for a Kanban system to have unlimited downstream process steps. In the XIT example at Microsoft, it was assumed that the user base available to perform acceptance testing was unlimited and hence there was no need to limit WIP in user acceptance testing. At Corbis the release ready queue was unlimited. This was based on the assumption that the batch of release ready work would never become excessive given the agreed release cadence of bi-weekly. If on the other hand, it was possible that release ready material would have become excessive, raising the complexity of a release with the result that the coordination effort and the transaction costs of the release become uneconomical, then it would have been necessary to limit WIP in the release ready queue. However, this was never the case at Corbis and as a result the release ready state was unrestricted.

Don’t Stress Your Organization

Choosing overly tight WIP limits initially may cause excessive stress on your organization. Lower maturity organizations with poorer capabilities that result in many impediments blocking work may find that introducing Kanban causes excessive pain if WIP limits are too low. If there are lots of impediments, visualized by lots of pink tickets across the card wall then overly tight WIP limits will mean everything grinds to a halt and lots of people are idle. While idle time tends to focus attention and accelerate efforts to resolve issues and remove impediments, it may just be too painful. Senior managers can become irritable observing many idle people who are still collecting a pay check. When introducing change, you need to be aware of the J-curve effect. What you desire with each change is a little j where any impact on performance is shallow and you quickly recover and show improvement. If you make the WIP limits too tight, you will suffer a J-curve effect that is too deep and too long. This may result in very undesirable effects. Kanban is exposing all the problems in the organization but it may end up being blamed for making everything worse. It was be seen as part of the problem rather than the solution. So tread carefully. With more capable, more mature organizations that suffer few unexpected issues (assignable cause variations) then you can be more aggressive with your WIP limit policies. For more chaotic organizations, you will want to introduce initially looser limits with much more WIP.

It’s a Mistake Not to Set a WIP Limit

While I will caution you not to be aggressive when setting initial WIP limits, I have become convinced that not setting WIP limits is a mistake.

Some early adopters of Kanban, such as Yahoo!, chose not to set WIP limits because they assumed their teams were too chaotic to cope with the pain that it will introduce. The hope was that these organizations would mature through the visual control elements of Kanban and that WIP limits could be introduced later. However, this proved very problematic and several of these teams abandoned Kanban without seeing much improvement, while other teams were disbanded in corporate reorganizations or project cancellations. This was true at Corbis where several teams on major projects pursued Kanban with only very loose WIP limits on courser-grained, higher level functionality. The results were somewhat mixed.

I’ve become convinced that the tension created by imposing a WIP limit across the value-stream is a positive tension. This positive tension forces discussion about the organizations issues and dysfunctions that are causing impediments to flow resulting in sub-optimal productivity, cycle time and quality. This discussion and collaboration is healthy. It is the mechanism that enables the emergence of a continuous improvement culture. Without WIP limits progress on process improvement is slow. Teams that have imposed WIP limits from the beginning have reported accelerated growth in capability and organizational maturity and have delivered superior business results with frequent, predictable deliveries of high quality software in comparison to teams who have deferred introduction of WIP limits and have generally struggled showing only limited improvements with their partial Kanban implementation.

Takeaways

- WIP limits should be agreed through consensus with upstream and downstream stakeholders and more senior function management
- Unilateral declaration of WIP limits is possible but may prove difficult to defend later when the system is placed under stress
- WIP limits for work tasks should be agreed as an average number of items per person, developer pair, or small collaborative team
- Typically, the limit should be in the range of 1 to 3 items per person, pair or team
- Queue limits should be kept small and typically only enough to absorb the natural (chance cause) variation in size of items and task duration
- Bottlenecks should be buffered
- Buffer sizes should be as small as possible but large enough to insure optimal performance in the bottleneck and maintenance of flow in the system
- All WIP limits can be empirically adjusted
- Kanban is an empirical process
- Excessive time should not be wasted trying to determine the perfect WIP limit, simply pick a number that is close enough and make progress. Empirically adjust if necessary
- Unlimited downstream sections of workflow are possible
- Care should be taken that unlimited workflow steps do not introduce bottlenecks or cause excessive transaction or coordination costs when deliveries (batch transfers downstream) are made

Chapter 11 Establishing Service Level Agreements

When processing work through our Kanban system we want to minimize waste by eliminating as many non-value-added activities as possible, while maximizing both the flexibility we offer the customer and the value being delivered.

The key to doing this is to recognize that not all work is born equal. Some requirements are needed more quickly than others and some are more valuable than others. By offering to treat different types of work with different classes of service we offer the customer more flexibility while optimizing value delivery.

Classes of service offer us a short hand to optimize customer satisfaction. By quickly identifying the class of service for an item we are spared the need to make a detailed estimate or analysis. Equally the use of the policies associated with the class of service affect how items are pulled through the system. This allows for a self-organizing, value optimized approach to prioritization and "re-planning." Classes of service, therefore, enable us to eliminate lots of transaction and coordination costs associated with planning and scheduling work. The trade off is that we don't treat each item as unique and we don't make explicit estimates or promises for every item. We use discretion to focus those wasteful activities only on items that carry greater risk and greater reward.

Typical Class of Service Definitions

Classes of service are typical defined based on business impact. Either, different colored sticky notes, index cards or tickets are assigned to clearly signify the class of service of any given request, or separate swim lanes are drawn on the card wall to signify membership of a class of service.

[Insert diagram showing whiteboard with swim lanes for class of service]

Each class of service comes with its own set of policies that affect prioritization decisions and imply a promise to the customer. Here is a short example set of class of service definitions. While this set isn't a precise facsimile of those used at any specific Kanban implementation, the set does represent a strong generalization of classes of service observed in the field.

In this example set, 4 classes of service are offered. As a general guideline, you may want to offer up to 6 classes of service. Too many will become too complicated to administer and operate. You want the number of classes of service to be few enough that everyone involved, team members and external stakeholders, can remember them all, while sufficient enough to offer flexibility in scheduling and prioritization to optimize the value delivered.

Expedite

The Expedite (or "Silver Bullet") class of service is well understood in manufacturing industry. A typical scenario might involve a sales team attempting to hit a quarterly sales target and a customer with budget remaining to spend in a fiscal year. The customer has been delaying a purchase decision then finally makes a choice as time runs out on the current fiscal year. They agree a price and quantity and give the salesman the order with the specific instructions that the order must be fulfilled, delivered and invoiced before the final day in the quarter. The order typically hits the factory via the regional sales vice president's office with a request to expedite delivery given the tight time and value of the order.

However, expediting orders badly randomizes manufacturing supply chains and distribution systems. Expediting is known in industrial engineering and operations research to both increase inventory levels and increase cycle times for other non-expedite orders. The business is making a choice to realize value on a specific sale at the cost of delay to other orders and the cost of additional carrying costs of higher inventory levels. If the company is well governed then the value generated through expediting will exceed the costs incurred through longer cycle times and potential lost business as a result and the cost of carrying the higher inventory level.

Manufacturing companies often create policy to limit the number of expedite requests hitting a factory. One common policy is to grant a fixed number of so called "silver bullets" to a regional sales vice president in a given time period. Hence, the term "silver bullet" has become synonymous with expediting in manufacturing or distribution.

Unfortunately the term "silver bullet" is already in use in software engineering. Fred Brooks defined it as a single change (in technology or process) that would create an order of magnitude (10x) improvement in programmer productivity. Hence, I recommend that you stick with the term Expedite class of service. However, in companies that do manufacturing or where the senior management is familiar with manufacturing, I've observed that they prefer to adopt the use of "silver bullet." This is fine so long as the technology people realize the difference in usage.

Fixed Delivery Date

[Insert Image of purple card] When a requirement has a unit step cost of delay function (or near unit step) then it must be treated as though it has a fixed delivery date. The date cannot be exceeded and everything should be done to insure that the item is prioritized into and through the Kanban system in sufficient time so as to arrive ready for release in an optimal fashion – not too early but never late.

Examples of this type of request include regulatory requirements in the financial sector that require deployment on a given date and may result in a fine or a loss of trading ability until resolved. Or requirements that have some physical or calendar constraint such as delivery for a holiday or the firing of a rocket and space probe that requires a particular alignment of planets in the Solar System. Businesses with seasonal calendars such as the schools and colleges tend to have hard calendar constraints. If you are working in a sector like education then you may find that customers want deliver of software at fixed times of year and failure to deliver in their window results in a lost sale. Anything that has a physical or cultural "launch window" should be considered as having a (near) unit step cost of delay function and should be treated as having a fixed delivery date. One real example involved the upgrading of an e-commerce solution to work with the newer version of the Cybersource credit card clearing functionality. Cybersource had replaced their system and its API and given the old system a 15 month sunset period before they turned it off. The older system was due to be decommissioned on March 31st 2007. If the e-commerce software was not fully upgraded to the new Cybersource API prior to that date, the firm would cease to trade via the Internet on April 1st. This was no joke. The request to upgrade the e-commerce sales functionality to the latest Cybersource release entered the Kanban system as a Fixed Delivery Date class of service request.

Other examples might include a contractual obligation with a major strategic business partner. Failure to meet a date may result in invocation of a penalty clause or in a loss of trust between the partners that may be material to the successful execution of the contract and the on-going business relationship.

Standard Class

The standard class involves a typical linear cost of delay curve where the value of delay can be defined as a tangible amount (under uncertainty). Both real out-of-pocket costs will be incurred, along with a tangible opportunity cost from late delivery, can be calculated. That tangible opportunity cost will typical be an s-curve that can be approximated with a linear graph.

Some business may choose to delineate several classes of service based on the steepness of the cost of delay graph. Up to 3 classifications could be quite reasonable depending on the funds put at risk through delay. For example, if the cost of delay would put the business into severe financial difficultly within a month or two then clearly any delay is serious. If a short delay would result in a thread of extinction then the request is more likely to be treated as a Fixed Delivery Date item. At the other end of the scale, the loss may amount to a small portion of discretionary funds and may not be material to the accounting of the business. If this is the case, the item may be treated as an Intangible Class item (see below). In between these two extremes are a myriad of scenarios that result in the loss of significant funds that the business and its owners might reasonably have expected to deploy elsewhere. It is these items that will be treated with the standard class of service. It is also likely that most requests in the system will be of this class.

Intangible Class

[Insert Image or green card] The intangible class covers any changes that cannot be calculated to have a tangible business value but the business either knows intuitively to be important or has some subjective empirical evidence such as customer complaints on which to base a valuation. Intangible items may include production bug fix requests, usability improvements, branding and design changes and their like. Again some business may find value in delineating several classes of service within the intangible set. For example strategic brand initiatives may require faster service than minor usability changes or bug fixes. An initiative to change the brand of a business probably has a project plan and a budget of its own. Perhaps a whole chain of retail stores is being rebranded, remodeled and reopened. Perhaps a major advertising campaign is being scheduled. Any technology changes required to support the new brand initiative must arrive on-time or will cause a significant cost. In this case it is likely that the request will be treated as either a Fixed Delivery Date or a Standard class of service item. In an alternative example, production bug fixes may be given different classes of service based on severity. Critical severity items – severity 1 – may be Expedite class of service, with High severity – severity 2 – as either Fixed Delivery Date or Standard class of service, depending on how the fix was promised to the market place.

Policies for Class of Service

Classes of service work through the use of a visualization technique that readily identifies the class of service. As explained earlier, either different colors or tickets or different swim lanes on the card wall are the most common. This chapter will use different colors to signify class of service but this is not an indication that color is the only way or the best way to achieve the desired result. The goal is to insure that the simple prioritization policies associated with a class of service can be used by any staff member to make a good quality prioritization decisions, in the field, on any given day, without management intervention or supervision.

Below I provide an example set of prioritization policies for the 4 classes of service defined above. Naturally, with every implementation of Kanban, the class of service definitions will be different and so should the policies in use. What appears below is based on empirical evidence and fairly accurately reflects policies I've observed used in the field.

Expedite Policies

Expedite requests will be shown with white colored cards.

Only one expedite request will be permitted at any given time. In other words, a kanban limit of 1 is assigned to the Expedite class of service.

Expedite requests will be pulled immediately by a qualified resource. Other work will be put on-hold to process the expedite request.

The kanban limit at any point in the workflow may be exceeded in order to accommodate the expedite request.

If necessary a special (off-cycle) release will be planned to put the expedite request in production as early as possible.

Fixed Delivery Date Policies

Fixed delivery date items will use purple colored cards.

The required delivery date will be displayed on the bottom right hand corner of the card.

Fixed delivery dates will receive some analysis and an estimate of size and effort may be made to assess the flow time. If the item is large it may be broken up into smaller items. Each smaller item will be assessed independently to see whether it qualifies as a fixed delivery date item.

Fixed delivery date items will be held in the backlog until close to the point where they must enter the system in order to be delivered on-time given the flow time estimate.

Fixed delivery date items will be given priority of selection for the input queue at the appropriate time.

Fixed delivery date items will be pulled in preference to other less risky items. In this example, the will be pulled in preference to everything except expedite requests.

Unlike expedite requests, fixed delivery date items will not involve putting other work aside or exceeding the kanban limit.

If a fixed delivery date items gets behind and release on the desired date becomes less likely it may be promoted in class of service to an expedite request.

Standard Class Policies

Standard class items will use yellow colored cards.

Standard class items will be prioritized into the input queue based on an agreed mechanism such as democratic voting and will typically be selected based on their cost of delay or business value.

Standard class items will use First in, First out (FIFO) queuing approach to prioritize their pull through the system. Typically when given an option a team member will pull the oldest standard class item from an available set of standard class items ready for the next step in the process.

Standard class items will queue for release when they are complete and ready for release. They will be released in the next scheduled release.

No estimation will be performed to determine a level of effort or flow time.

Standard class items may be analyzed for size. Large items may be broken down into smaller items. Each item may be queued and flowed separately.

Standard class items will generally be delivered with X days of selection. Typically, we've seen service level agreement delivery times where X equals a number in the range 28 to 44 days. Determining a good estimate of the time required is explained later in this chapter.

Intangible Class

Intangible class items will use green colored cards.

Intangible class items will be prioritized into the input queue based on an agreed mechanism such as democratic voting and will typically be selected based on their intangible business value.

Intangible class items will be pulled through the system in an ad hoc fashion. Team members may choose to pull an intangible class item regardless of its entry date so long as a higher class item is not available as a preference.

Intangible class items will queue for release when they are complete and ready for release. They will be released in the next scheduled release.

No estimation will be performed to determine a level of effort or flow time.

Intangible class items may be analyzed for size. Large items may be broken down into smaller items. Each item may be queued and flowed separately.

Typically, the preference would be to put aside an intangible class item in order to process an expedite request.

It is therefore sensible and shows a good spread of risk when intangible class items are flowing through the system.

Determining a Service Delivery Target

In the example set of classes of service above, the Standard class of service involved the use of target cycle time for example 28 days (4 weeks). The idea with the target cycle time is to avoid wasteful activities such as estimation and to avoid making explicit promises that due to uncertainty we may fail to meet. By avoiding making promises we are unlikely to be able to keep, we avoid the danger of losing the trust of our customers. So it's important to communicate that the target cycle time in the Standard class of service is just that, a target!

To determine the target cycle time, it helps to have some historical data. If you don't have any make a reasonable guess. If you do have some then the most scientific means of determining the target cycle time, would be to process the cycle times (from first selection until delivery) through a statistical process control package and use the upper control limit (or upper 3 sigma limit) for your cycle time. This will insure a time that you can hit under most normal circumstances and miss only when there is a genuine assignable cause.

If on the other hand, that last paragraph meant nothing to you, then a more lay person explanation is that you want the cycle time to be achievable most of the time, but aggressive enough that it keeps the team focused. It is likely that your work items vary in size, complexity, risk and expertise required. So the cycle times will suffer significant variation. That's OK. If you perform a spectral analysis of some historical data and can see that perhaps 70% is delivered with 28 days, with the remaining 30% spread out over another 100 days, then perhaps it's OK to suggest a target delivery date of 28 days.

We've learned that classes of service is a very powerful technique. At Corbis in the first year of running Kanban approximately 30% of all requests were late against the target cycle time. We reported this as the Due Date Performance metric. It was never above 70%. However, despite this dismal performance against target date, we found that there were very few complaints. The reasons for this became evident. All the important items, those with high risk, high value were always on time, and there was a trust that those that were late would be delivered with 2 or 4 weeks as releases were happening with regularity.

The Expedite and Fixed Delivery Date classes of service were insuring that important items were always on time. Meanwhile, the other Standard class items that were late were generally delayed only by 1 release (14 days) or 2 releases (28 days). The customers trusted the release cadence. That trust was earned by doing. We consistently shipped a release every 2nd Wednesday. With the insignificant cost of delay associated with many Standard (and Intangible as they were not delineated separately) class items, the business focused on what had been delivered and planning for future items rather than worrying too much about precise delivery dates.

This result was significant because Kanban with classes of service had clearly changed the customer psychology and significantly changed the nature of the relationship and the expectations. The customer was now oriented around the long term relationship and the performance of the system and not on the delivery of any one item or items. This gave the development team the freedom to focus on the right things and not waste time addressing issues related to a low level of trust between them and the customer.

Putting Classes of Service to Use

Classes of service should be defined for each kanban system. The policies associated with each class of service should be explained to every team member involved. Everyone attending a standup meeting in the morning should appreciate and understand the classes of service in use. To make this effective, the number of classes of service should be kept fairly small - 4 to 6 would be a good guideline. And again, because we want each team member to remember the classes of service, their meanings and how to use them, the number of policies for each class of service should be kept small and simple. The definitions should be precise and unambiguous. Again a good guideline would be no more than 6 policies per classes of service.

Armed with an understanding of classes of service and knowledge of the policies associated with them, the team should be empowered to self-organize the flow of work. Work items should flow through the system in a fashion that optimizes business value and customer service and the result should be releases of software which maximize customer satisfaction.

Takeaways

- Classes of service offer a short hand for optimizing customer satisfaction
- Work items should be assigned a class of service according to their business impact
- Classes of service should be clearly visually displayed by using for example different colored cards to delineate the class of service or different swim lanes on the card wall
- A set of management policies should be defined for each class of service. Only classes of service related to riskier items should involve wasteful activities such as estimation
- Team members should be trained to understand the classes of service in use and the policies associated with them
- Some classes of service should include a target cycle time
- Due Date Performance (%) should be monitored for target cycle times
- Classes of service enable self-organization, empower team members and free up management time to focus on the process and not on the work
- Classes of service change the customer psychology
- If classes of service are used properly combined with a regular delivery cadence, then very few complaints will be received even if a significant portion of items miss their target cycle time

Chapter 12 Metrics & Management Reporting

While Kanban tries to be minimally invasive and change as little of the value stream, job roles and responsibilities as possible, it does change the way the team interacts with its partners, the external stakeholders. Because of this Kanban needs to report slightly different metrics than you may have been used to with a traditional project management approach or an Agile project management approach.

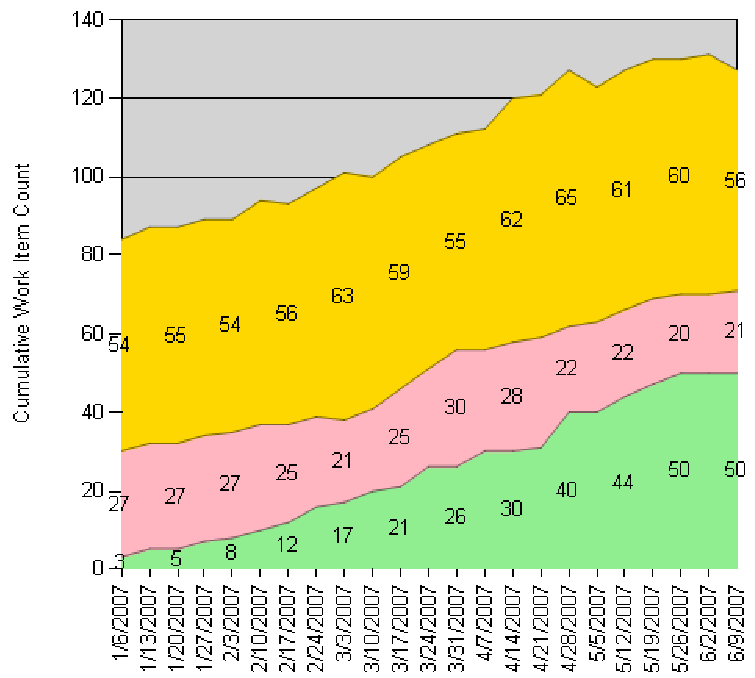
Kanban's continuous flow approach means that we are less interested in reporting on whether a project is "on-time" or whether a specific plan is being followed. What's important is to show: that the Kanban system is predictable; that the organization exhibits business agility; that there is a focus on flow; and that there is clear development of continuous improvement.

For predictability we want to show how well we perform against the class of service promises. Are work items being treated appropriately, and if the class of service has a target cycle time, how well are we performing against that?

For each of our indicators we are going to want the trend over time, so that we can see the spread of variation. If we are to demonstrate continuous improvement we want the mean trend over time to improve. If we are to demonstrate improved predictability we want the spread of variation to reduce.

Tracking WIP

However, before we get to performance indicators, I believe that the most fundamental metric should show that the Kanban system is operating properly. To do this we need a cumulative flow diagram that shows the quantities of work-in-progress at each stage in the system. If the Kanban system is flowing correctly then the bands on the chart should be smooth and stable in height.



Picture xx Example of Cumulative Flow Diagram from a Kanban System

This example shows how well the team is doing at maintaining the Kanban limits. We can see that WIP (the pink band in the middle) is growing. This was due to a manager in the chain consistently exceeding (or overriding) the WIP limit. We can also read the average cycle time by scanning this chart horizontally.

Cycle Time

The next metric we are interested in relates to how predictable our organization delivers against the promises in the class of service definitions. The underlying metric for this is cycle time. If an item was expedited, how quickly did we get it into production from order? If it was of standard class, did we deliver it within the target cycle time? I've found the best way to show this data is with a spectral analysis of the cycle time, indicating the target cycle time (or service level agreement time, SLA) on the chart.

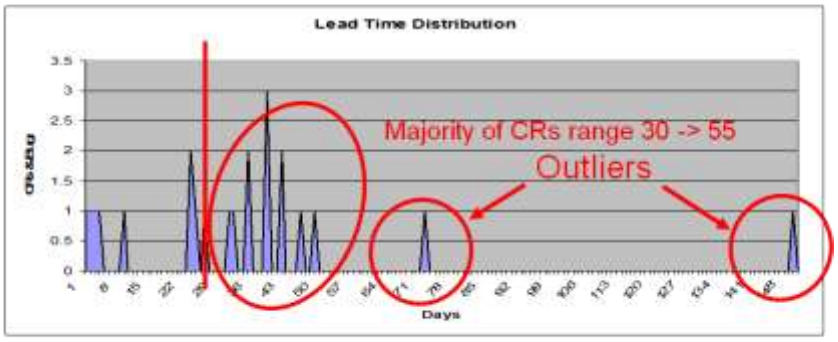


Figure xx. Example of Cycle Time spectral analysis

Reporting the average cycle time has some use as a report card on overall performance but it is not very useful as an indicator to inform improvement initiatives.

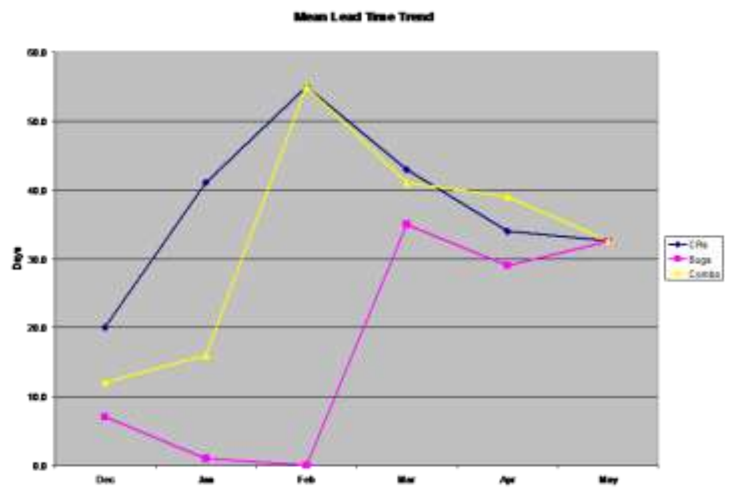
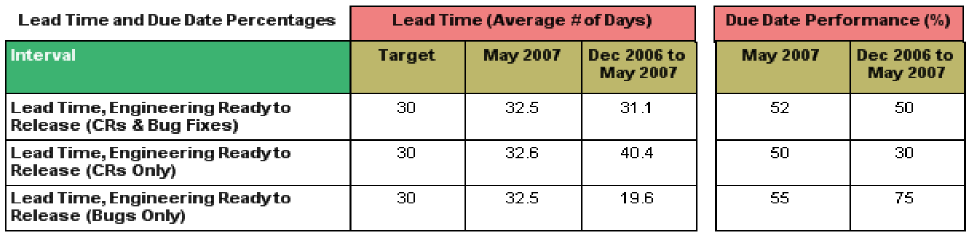


Figure xy. Example of Mean Cycle Time Trend

The spectral analysis is much more useful because it informs us of items which just failed to meet the target time, and of other statistical outliers. In the example, it would make sense to investigate the root causes of the cluster of items which just failed to meet the target cycle time. If these root causes could be addressed then the Due Date Performance (or percentage of items delivered with expectation) should improve. This brings us to the next reporting metric, Due Date Performance.



Lead Time and Due Date Percentages	Lead Time (Average # of Days)			Due Date Performance (%)	
Interval	Target	May 2007	Dec 2006 to May 2007	May 2007	Dec 2006 to May 2007
Lead Time, Engineering Ready to Release (CRs & Bug Fixes)	30	32.5	31.1	52	50
Lead Time, Engineering Ready to Release (CRs Only)	30	32.6	40.4	50	30
Lead Time, Engineering Ready to Release (Bugs Only)	30	32.5	19.6	55	75

Figure yy. Example of Mean Cycle Time and Due Date Performance

Due Date Performance

I've found it useful to report Due Date Performance for the most recent month and for the year to date. You may also want to report performance year or year (or 12 months ago) for comparison.

With Fixed Delivery Date class of service items, you can include these in the Due Date Performance metric. In this case, you are answering the question, was the item delivered on-time? However, while you will have a cycle time recorded, that in itself is not so interesting as to compare the estimated cycle time against the actual. Estimate versus actual demonstrates how predictable the team is and how well they are performing with Fixed Delivery Date service items. Naturally, the most important metric is whether the item was delivered on-time prior to the hard date. The accuracy of the estimate is an indicator of how efficiently the system is running. If estimates are known to be inaccurate, then the team will tend to start Fixed Delivery Date items early, in order to be sure of delivery. This is not optimal and the overall performance in terms of throughput can be improved by improving estimation.

Throughput

Throughput should be reported as the number of items, or some indication of their value, that was delivered in a time period such as 1 month. Throughput should be reported as a trend over-time. The goal should be to constantly increase it. Throughput is very similar to the Agile "velocity" metric. It indicates how many user stories, or story points were completed in a given period. If you are not using Agile requirements techniques but processing other things such as functional specification items, change requests, use cases, etc, then report the number of those.

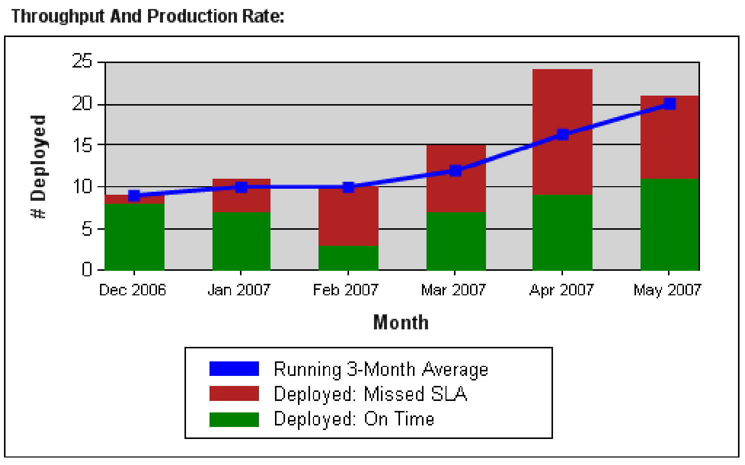


Figure yz. Example of Throughput bar chart

In the first instance it is important to be able to report the raw number. As your team matures and becomes more sophisticated you may be able to report the relative size such as total number of story points, function points or some other measure of quantity. If your organization is very sophisticated then you may be able to report the value of the work delivered as a dollar amount. At the time of writing I only know of one team at the BBC in London that is capable of reporting the dollar value of work delivered.

Throughput is used in Kanban for an entirely different purpose than velocity is used in typical Agile development. Throughput is not used to predict the quantity of delivery or any commitment. Throughput is used as an indicator of how well the system (the team and organization) is performing and to demonstrate continuous improvement. Commitments in Kanban are made against cycle time and target delivery dates.

Issues & Blocked Work Items

The issues and blocked work items chart shows a cumulative flow diagram of reported impediments overlaid with a graph of the number of work items in-progress that have become blocked. This chart gives us an indication of how well the organization is at identifying, reporting and managing blocking issues and the impact they are having. If Due Date Performance is poor there should be corresponding evidence in this chart to demonstrate that a lot of impediments were discovered and were not resolved quickly enough. This chart can be used on a day-to-day basis to alert more senior management of impediments and their impact. It can also be used as a long term report card to indicate how well the organization is capable of resolving impediments and keeping things flowing.

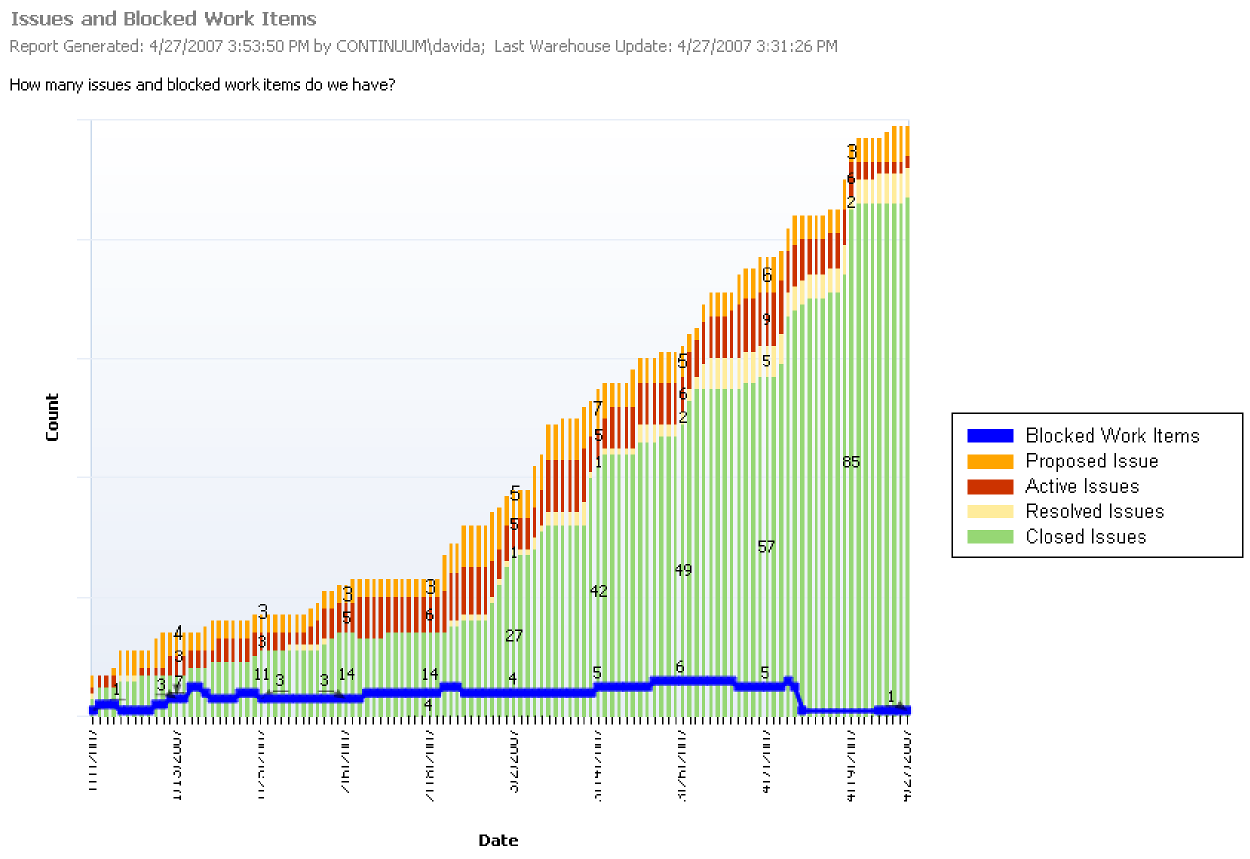


Figure ZZ. Example of Issues & Blocked Work Items chart

Flow Efficiency

A good Lean indicator of the waste in the system is to measure the cycle time against the touch time. In manufacturing touch time implies time a worker actually spends touching a job. In software development this is very difficult to measure. However, most tracking systems will let us track assigned time (to an individual) against time spent blocked and queuing. Hence, while reporting the cycle time to assigned time ratio doesn't give us an accurate indication of the waste in the system, it does give us a conservative ratio that shows how much potential there is for improvement.

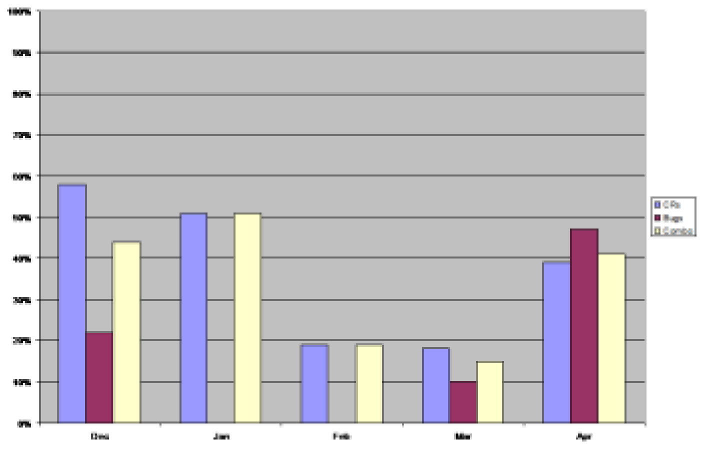


Figure zy. Example of Cycle Time : Assigned Time ratio

Do not be alarmed if this ratio is initially about 10:1. I've attended many conferences and seen many presenters from different industries as diverse as new aircraft design and medical equipment design report similar ratios. It seems with knowledge work we are terribly inefficient and incapability of the agility required to turn an idea or request into a working product in an efficient manner.

The flow efficiency metric is not very useful day-to-day but again is another indicator of continuous improvement.

Initial Quality

Defects represent opportunity cost and affect the cycle time and throughput of the Kanban system. It makes sense to report the number of escaped defects as a percentage against the total WIP and throughput. Over time, we want to see the defect rate fall to close to zero.

[Insert example]

Failure Load

Failure load is the concept of how many work items are we processing because of earlier poor quality. How many work items are production defects, or new features requested through our customer service organization because of poor usability or a failure to properly anticipate the needs of the user. We want to see failure load fall over time. This is a good indicator that we are improving as a whole organization and thinking at a system level.

[Insert Example]

Takeaways

- Track WIP with a cumulative flow diagram to monitor Kanban limits day-to-day
- Track cycle time for each item processed and report the mean and the spectral analysis for each class of service
- Cycle time is an indicator of business agility
- Track estimate versus actual cycle time for Fixed Delivery Date class of service items
- Report Due Date Performance as an indicator of predictability
- Impediments block flow and impact cycle time and due date performance, report blocking issues and number of blocked work items in a cumulative flow diagram with a overlying graph of blocked items. Use it to indicate capability to report problems and resolve them quickly
- Flow efficiency is the ratio of cycle time to assigned engineering time. It indicates how efficient the organization is at processing new work and is a secondary indicator of business agility. It also indicates how much room for improvement is available without changing engineering methods
- Initial Quality reports the number of bugs being discovered by testers within the system and indicates how much capacity is being wasted through poor initial quality.
- Failure Load reports the percentage of work that is generated through some failure of the system and shows capacity left on the table that could be used for new value-added features

Chapter 13 Scaling Kanban with Two-tiered Systems

So far the examples and stories of Kanban implementations I've presented have focused on software maintenance and small changes with rapid release to production. There a lot of systems that are in maintenance and a significant portion of readers involved in software development will find the advice and guidance useful. Equally, there are many more IT personnel involved in support and operations where ticketing systems for short order jobs are also common and a Kanban approach would be equally useful. However, there will be others for whom development of significant sized projects is the norm. If you are reading this asking why and how would I use Kanban on larger projects and across a project portfolio then I hope that chapters 13 & 14 have persuaded you that Kanban enables significant and positive cultural changes. The benefits observed with Kanban are sufficiently desirable that they challenge us to ask the question, "How would we do Kanban on large projects?"

Large projects present some significant challenges. There are a lot of requirements that have to be released together. It will be some considerable number of months before the release is made. The team size is larger. There might be lots of work happening in parallel. Significant pieces of work may need to be integrated. Not all of this work will be software development. For example, documentation and packaging design may need to be integrated with the final software build before a release can be made.

So how do we deal with these challenges?

The answer is to look to first principles. The first principles of Kanban are that we want to limit work-in-progress and pull work using a visual signaling system. Beyond that we look to Lean principles, Agile principles and the workflow and process that is already in place as our starting position. So we want to limit WIP, use visual control and signaling and pull, but we also want small batch transfers, prioritize value, manage risk, make progress with imperfect information, build a high trust culture and respond quickly and gracefully to changes that arrive during the project.

With a large project just as a maintenance initiative you will need to agree a prioritization cadence. And the general rule is more often is better. Look at the principles again. What are, the transaction and coordination costs involved in sitting down with the marketing team or business owners and agreeing the next items to queue up for pull? At the other end of the value stream, you will have several integration or synchronization points building towards a release rather than a single release point. So again from first principles look at the transaction and coordination costs of any integration or synchronization and agree a cadence. Again more often is better. Ask the question, what is involved in meeting with the business to demo recent work and to integrate that work so that it is "release ready?"

Next, you will want to agree WIP limits and the principles for thinking about this will not change. And classes of service will still make sense and help you to cope gracefully with changes.

Hierarchical Requirements

You will also need to define work item types for your project. Many major projects feature hierarchical requirements. It is not uncommon for these to be up to three levels deep. There may also be different types of requirements such as customer requirements that came from business owners and product requirements that came from a technical, quality or architectural team. The requirements might be broken out further into functional and non-functional or quality of service requirements. Even with Agile software development, the customer might specify requirements in terms of epic sized stories which are broken out into user stories and are perhaps broken down to a lower level of tasks. I have also seen epics broken into architectural stories that in turn are broken into user stories. Feature driven development has three tiers of requirements also – features, feature sets (or activities) and subject areas.

It has made sense for teams adapting Kanban for major projects to set different work item types for different levels of the hierarchy. For example epic stories are one type of work item, and smaller user stories are another type of work item. In a more traditional project, customer requirements are one type, while product requirements are another and functional specification items are a smaller third type.

Typically teams have chosen to track the two top levels on a Kanban board. I personally have not seen a team or project where they tried to track three levels with Kanban. If there is a third lower level such as tasks in an Agile project, the tasks are not tracked on the project card wall or within the team level kanban system. Individual developers may choose to track tasks or perhaps small cross-functional teams might choose to track their tasks locally but off the larger project board and out of sight of managers and value-stream partners. This isn't to motivated by a need to hide information. It simply that the lowest level of activity isn't interesting from a value-stream and performance level. The lowest level is often focused on effort and activity rather than on customer value and functionality.

Decouple Value Delivery from Work Item Variability

What emerged with most Kanban teams tracking the two highest levels of requirements was the idea that the highest level of requirements, the most coarse-grained requirements, were generally describing some atomic unit of value to the market or customer. These epic user stories or customer requirements were often written at a level where they made sense to release to the market. Were the product already in maintenance these requests may have been processed and released individually. Sometimes the Kanban community refers to this level of requirements as a "minimal marketable feature" (or MMF). There is confusion as MMF was defined by Denne and Cleland-Liang in their book Software by Numbers and this definition isn't strictly adhered to. I would prefer a definition of a minimum marketable release (MMR) that described a set of features that were coherent as a set to the customer and useful enough to justify the costs of bringing them to market.

It does not make sense to treat a MMR as a single item to flow through our kanban system. A MMR is made up of many work items. MMR makes sense from a release transaction cost perspective not from a flow perspective. In some cases a small but highly valuable, differentiated new feature may make economic sense to release. On the other hand, as many have found out, "the first MMF is always large" because the first release of a new system must include all of the table stakes capabilities to enter a market and all the infrastructure to support them. There can be two or three orders of magnitude difference in the size of MMFs (or MMRs). A work item type that has instances that vary in size up to 1000 fold will be problematic.

Kanban systems do not appreciate such wide variation in size. They require large buffers and excess WIP to smooth flow or without them they will suffer wide fluctuations in cycle time. Large buffers and more WIP will mean long cycle times and a loss of business agility. The alternative worse! If we do not buffer for variability in size, there will be wide fluctuations in cycle time. As a result it is impossible to offer a target cycle time under a service level agreement that can be met with any consistency. The result will be poor predictability, and a loss of trust in the system. So designing a kanban system around the concept of MMF is likely to lead to a loss of business agility and/or a loss of predictability, a loss of trust between IT and the business and general dissatisfaction with Kanban as an approach.

However, use of a Minimal Marketable Release (MMR) to trigger a delivery, in conjunction with smaller fine-grained work item types, is likely to minimize costs and maximize satisfaction with what is released.

Teams can adapt to this challenge by focusing on analysis techniques that produce a lower level of requirements, such as user stories or a functional specification. These will generally be fine-grained, small, and with a relatively small variation in size. An ideal size would be something in the range of half a day to 4 days or so of engineering effort.

On one major project, we found that each larger work item, called a "Requirement" and tracked with green tickets, broke out into an average of 21 smaller "Features" tracked with yellow tickets. While the features were still written in a user and value-centric fashion, they were analyzed to be small and similarly sized. Had this been an Agile project these two levels might have been Epic, tracked in green, and User story, tracked in yellow.

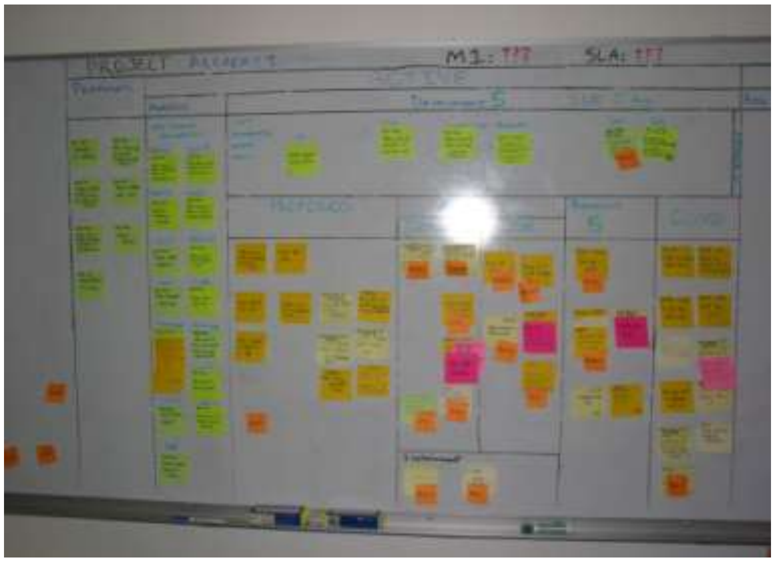
The smaller, fine-grained items enable flow, and predictability of throughput and cycle time, while coarser-grained items at the higher tier in the board allow us to control the number of releasable, marketable requirements in progress at any given time.

By adopting this two-tiered approach we have decoupled the delivery of value from the variability of the size and effort required to deliver that value.

It makes sense to set WIP limits at both levels. With several projects we found it made sense to assign small cross-functional teams to each of the higher level requirements. That team would then flow and pull all the smaller, finer-grained items for that higher level requirement until the requirement was complete and ready for integration or delivery. The team would then pull another coarse-grained requirement. There would also be the opportunity to re-assign team members either adding or releasing individuals from the team depending on the size of the next item to be pulled.

Two-tiered Card Walls

The first teams using Kanban on large projects, adopted a two-tiered style of card wall, as shown in figure xx.



Picture xx Photograph of a two-tiered board

In this picture the coarse-grained requirements are shown with green tickets. They flow left to right through a set of states namely, backlog, proposed (analysis), active (design & development), resolved (testing) and closed.

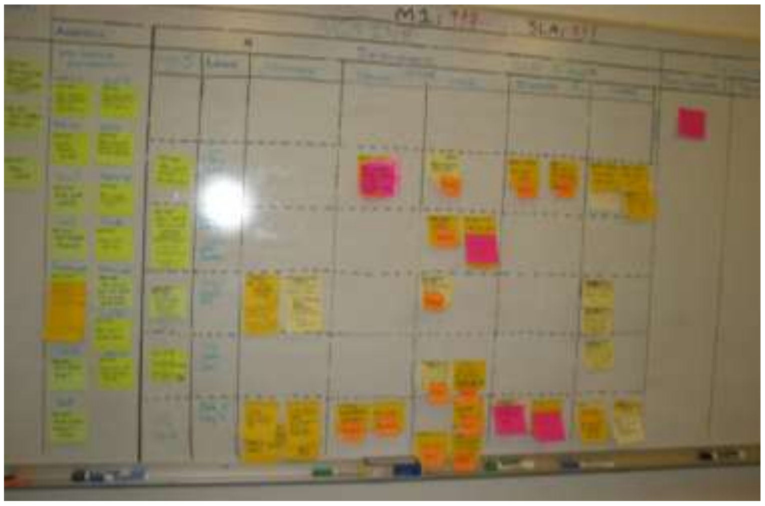
The requirements that are active are shown across the top of the middle of the picture. They in turn are broken out into lots of smaller features shown with the yellow tickets. The features are being flowed through their own set of states, namely, proposed (analysis), active (design & code), resolved (testing) and done. The states the features flow through is similar to the higher level requirements but it doesn't need to be. You can choose to model this however you see fit. My advice is to model what you do now. Do not change the process if you can avoid it.

The yellow tickets are linked back to their parents by tagging them with ID numbers of their parents.

In an example like this it is possible to limit WIP at both levels in the hierarchy but the yellow tickets are all grouped in the one pool. I don't have enough evidence from the field to know whether or not this is a good strategy. What I do know is that it didn't stick with this team.

Introducing Swim Lanes

It turns out that relating the finer-grained yellow tickets to their coarser-grained parent is important. It also seems to make sense to limit WIP at the lower level within the individual cross-functional team. To facilitate this approach, teams innovated with the card wall and introduced horizontal swim lanes.



Picture xy Photograph of a two-tiered board with swim lanes

In figure xy, the higher level requirements shown with green tickets are flowing through the same set of states, namely, backlog, proposed, active, resolved and closed. However, the middle section has been redrawn compared to figure xx. The active coarse-grained requirements in green are vertically stacked to the center left. From each of those green tickets extends a swim lane that is divided into the set of states for the finer-grained yellow features. The number of swim lanes is now the WIP limit for the coarse-grained customer marketable requirements, while the WIP limit for the finer-grained features can now be set on each swim lane if the individual teams choose to do so. The column immediately to the right of the vertical stack of green requirements contains the names of the permanently assigned team members. The small orange tickets attached to yellow tickets actually contain the names of specialist floating resources such as user experience designers and database architects.

This swim lane variant of the card wall means that we are now managing customer marketable WIP vertically, while managing WIP on the low variability features horizontally. This format proved to be very popular and stuck.

Incorporating Classes of Service

The two most obvious methods for visually differentiating cards on the wall are color and swim lane. However, on large projects we typically have three attributes of each ticket that we need to communicate: the work item type; the level in the hierarchy; and the class of service. It's worth noting that in the example shown the choice to have different work item types for different levels in the hierarchy and then both color the work item types and visualize hierarchy with swim lanes means that hierarchy is being overloaded with two methods of visualization.

If you do need to communicate class of service in addition to type and level in the requirements hierarchy, it may make sense to use color for class of service. If types are used for other purposes than hierarchy level, for example, if you want to show bugs or defects, or value-add versus failure load, then you may want to choose another approach, perhaps introducing an icon or a sticker attached to the card to signify type. Or you may prefer color for type and use an icon or sticker for class of service e.g. a silver star for an expedite request.

Takeaways

- Major projects should follow the core principles of Kanban
- WIP limits, prioritization cadence, delivery cadence and classes of service are valid techniques for major projects
- Major projects tend to have hierarchical requirements and these levels of hierarchy should be modeled with work item types
- Typically teams track the top two levels of requirements in the hierarchy on the card wall and limit WIP at one or both levels
- The highest level of requirements typically models customer marketable requirements that make atomic units that could potentially be released individually
- The second level of requirements is typically written in customer and user-centric language and analyzed in such as way as to make the requirements both fine-grained and similarly sized
- This second level of fine-grained requirements facilitates flow by reducing variability in the kanban pull system
- A two-tiered card wall will be required to visualize both levels of requirements that are being tracked
- Swim lanes have become a popular technique to show hierarchy and facilitate limiting WIP
- The coarse-grained WIP is limited by the number of swim lanes
- The fine-grained WIP can be limited on each swim lane if desired
- Small cross-functional teams are typically assigned to each swim lane

Chapter 14 Operations Review

It's 7.30am on the 2nd Friday in March 2007. I'm in work early because this morning is our department's fourth monthly operations review. I'm joined by Rick Garber, the manager of our software process engineering group. Rick has the job of coordinating the ops review meeting and agenda. He's busy printing out the handout that contains the 70 or so Powerpoint slides for today's meeting. Once the printing is done, we head over with a box of 100 hand outs to the Harbor Club on 2nd Avenue in Downtown Seattle. Ops review is scheduled to start at 8.30am but a hot buffet breakfast is served from 8am. The meeting is planned for about 80 people. The invite includes all of my organization and all of my colleague Erik Arnold's organization. However, with some folks in India, some in other parts of the USA and always a few who can't make it for personal reasons, we generally get around 80 attendees.

The invite also includes my boss, the CIO of Corbis, and a number of other senior managers, our value stream partners. The external group that attends in highest numbers is the Network and Systems Operations team run by my colleague Peter Tutak. They after all feel the pain of our failure most when they have to recover failed systems in production. They also have the greatest impact when we make new releases to production. So arguably they have the most to gain by taking an active interest in participating.

The team begins to arrive in good time to enjoy their breakfast. The room is on the top floor of a Seattle tower block and affords us all beautiful views of the city and Elliott Bay. The floor is laid out in rounds with 6 to 8 seated at each table. We have a projector screen at one end and a lectern. The schedule is managed precisely by Rick. Each presenter has around 7 minutes for their 4 or 5 slides. There are a few time buffers to allow variability caused by questions and discussion. I kick things off promptly with a few opening remarks. I ask everyone to think back to the end of January and what we were doing back then. I remind everyone that we are here to review the organizational performance for the month of February. Rick has picked out a nice picture from the company archives to represent a theme for the month and help jolt memories and remind everyone of a key activity from the month.

I hand off the proceedings Rick who summarizes the management action items from last month and gives everyone an update on the status. Next we introduce our finance analyst who presents a summary of the company performance for the month – the reason for delaying until the 2nd Friday of the subsequent month is so we can have the numbers from the closed books for the prior month. She then summarizes the detail of the budgets for my cost center and Erik's cost center. We look at planned versus actual for all major budget spend and headcount targets. We discuss open requisitions and encourage team members to submit candidates for open positions. Coming out of this first section, everyone attending knows how well the company is doing and how well the software engineering group is managing to budget and how much slack we have to buy items like flat screen monitors and new computers. The purpose of leading with the financial numbers is to remind everyone on the team that we are running a business. We are just showing up to have fun with 1s and 0s with a group of friends each day.

The next speaker is a guest – a vice president from another part of the company. I had the bright idea that if we wanted our value stream partners to take an interest, then we should invite them to present and show an interest in them. We offered our guest 15 minutes and he took it. So we got a presentation on sales operations, the part of the business that fulfilled customer orders and insured delivery of product. While some of Corbis' business is done on the web and fulfilled electronically, not everything the firm offers is delivered as a download and a whole department fulfills more complex orders for professional advertising agencies and media firms. My colleague Erik Arnold had the bright idea of asking the guest to sponsor our breakfast to keep our costs under control. That worked too. Over the next few months our team got to learn about many aspects of business and senior leaders throughout the company got to learn what we did, how we did it and how hard we were trying to deal with our issues. Nine months later executives were openly talking about how well governed the IT team was and how their business unit ought to be following our lead.

Once our guest speaker had finished, we moved into the main section of the meeting. Each manager had 8 minutes to present on their department's performance. We followed this with some project specific updates from our program management office. Each of the immediate team managers would get up and spend 5 minutes quickly presenting their metrics. Generally, they followed the format laid out in chapter 12. They would present information on defect rates, cycle time, throughput, value-added efficiency and occasionally they'd have a specific report that would be drilling into some aspect of our process where they needed more information. They would then take questions, comments and suggestions from the floor for a few minutes.

This fourth month of the Ops Review, March 2007, was particularly interesting. The first Ops Review was in December. Everyone came, almost 100% turnout. Lots of curiosity and afterwards lots of comments like "have never seen transparency like this in my career," and " that was very interesting." The most useful piece of feedback was, "next time can we have a hot buffet rather than cold?" So we added hot breakfast. The second month people said, "Yes, another good month. Somewhat interesting! Thanks for the hot breakfast!" On the third month some of the developers were asking, "Why do I need to get up so early?" and "Is this a good use of my time?"

What happened in the fourth month is that we reviewed a significant problem. The company had acquired a business in Australia. IT had been asked to switch off all the Australian subsidiary's IT systems and migrate all 50 users to Corbis systems. The request had an arbitrary but urgent date. This date was based on "economy of scale" style cost savings that had partly justified the acquisition price. So there was a "cost of delay" involved. The request had arrived as a single item in our maintenance queue. It was big enough to have justified 10 tickets but we treated it as only one. The effect of an outsized item like this entering a Kanban system is well understood in industrial engineering. It will clog the system and greatly extend the cycle time for everything that comes in behind it. And so it was with us. Cycle time jumped from 30 days to 55 days. Queuing theory will also tell you that reducing a backlog when fully loaded takes a long time. Again, we were to discover that it would take 5 months to recover the cycle time target.

In addition, we had a release that had required an emergency fix.

All of a sudden the room was alight with questions, comments and debate. After 3 months of boring good data we had a story to tell. The staff were amazed that we (the management) were willing to talk openly about the problems and what to do with them and that Ops Review wasn't only about showing off how good we were and presenting the good data. No one ever asked again why we held the meeting every month.

The meeting would end with Rick summarizing the management action items from this meeting and thanking everyone for coming. It was 10.30 and time to head back across the street to the office.

There are a lot of important things to understand about Operations Review. Firstly, I believe that Operations Review is the lynchpin or keystone of a Lean and Kanban system implementation. Operations Review is an objective data-driven retrospective on the organizations performance. It is above and beyond any one project and it looks sets an expectation of objective data-driven quantitative management rather than the more subjective, anecdotal, qualitative management that is more established practice with Agile project and iteration retrospectives. Operations review provides the feedback loop that enables growth of organizational maturity and organization level continuous improvement. I truly believe that is essential to delivering successful enterprise scale Lean (or Agile) transition.

I also believe that Operations Reviews have to be monthly. More often can be burdensome for data collection and the time involved for the meeting means that there is a desire to not do it too often. Fitting such a meeting into 2 hours is challenging. If it were not a data-driven meeting full of charts and reports it would not be possible. A subjective anecdotal style meeting at that scale would not be completed within two hours. A typical project retrospective takes longer than 2 hours. So imagine trying to perform an organization-wide retrospective and complete it in two hours using a pluses and deltas style analysis? So part of the secret to keeping the length of the meeting short is to hold it based on objective data. Keep the agenda tight and manage it throughout the meeting.

There can be a tendency to want to hold operations reviews less often. Quarterly is a common option. My experience with quarterly operations reviews dates from my time with Motorola's PCS division. My observation of these meetings is that they were upper management reporting and review sessions and not organizational sessions designed to drive continuous improvement and organizational maturity. Quarterly is too seldom to really drive an improvement program. The data is often 4 months old by the time it hits a quarterly review. A quarter is a long time to review in a single meeting so the review tends to be superficial. Reports and metrics tend to be of lagging indicators and focused around reporting performance against target to senior leaders.

Quarterly meetings seem attractive because they feel more efficient - just one two hour meeting every quarter rather than every month. They also cost less on an annual basis – just 4 meetings rather than 12. After I left Corbis at the beginning of 2008 my former boss reduced the cadence of the operations reviews to quarterly to save money. After 3 quarters, and with that boss also gone, the new leadership was questioning the value of the meetings and decided to cancel them altogether. Within another few months the performance of the organization had allegedly depreciated considerably and the level of organizational maturity had reportedly fallen back from approximately equivalent of CMMI level 4 to CMMI level 2 from Quantitatively Managed to merely Managed.

Several things can be drawn from this. The loss of a feedback loop reduced the opportunities for reflection and adaptation that would lead to improvements. The elimination of a meeting focused on objective performance review of the organization sent a message that leadership no longer cared about performance. The result was a significant step backwards in organizational maturity and performance in terms of predictability, quality, cycle times, and throughput.

The operations review also shows the staff what managers do and how management can add value in their lives. It also helps to train the workforce to think like managers and to understand when to make interventions and when to stand back and leave the team to self-organize and resolve its own issues. Operations review helps to develop the respect between the individual knowledge workers and their managers and between different layers or management. Growing respect builds trust, encourages collaboration and develops the social capital of the organization.

While individual project retrospectives are always useful, an organization-wide ops review fosters institutionalization of changes, improvements and processes. It encourages improvements to spread virally across an organization and creates a little "inter-mural" rivalry between projects and teams that encourages everyone to improve their performance. Team want to demonstrate how they can help the organization with better predictability, more throughput, shorter cycle times, lower costs, and higher quality.

I didn't invent operations reviews. They are quite common at many large companies. However, I learned how to do them in this objective, business unit wide fashion, while I was working at Sprint PCS. My boss, the Vice President and General Manager of sprintpcs.com, instituted them for very similar reasons. He wanted to develop the maturity of his organization – a 350 person business unit responsible for the web site and all e-commerce and online customer care for Sprint's cellular telephone business. At sprintpcs.com we held the ops review every 3rd Friday of the month at 2pm. It lasted for 2 hours and involved around 70 senior staff and managers from the business unit plus director or senior manager level invitees from our upstream and downstream partners. Senior leaders including the Chief Marketing Officer and the VP of Strategic Planning were also regular attendees. The format was very similar. It was entirely objective data-driven. Each manager presented their own data. The meeting led off with financial data first. The schedule was planned and managed tightly. After the meeting everyone got to go home early on a Friday. The meeting was held offsite, in this case at a local college campus. While sprintpcs.com had struggled with Agile software development techniques, the operations review was a key element in developing the organization maturity and improving the governance of the organization. It showed staff that managers were making a difference and knew how to manage and it gave staff a line managers a chance to show senior leaders how they could help and where they needed interventions to truly make a difference.

Given two experiments over a period of 4 years throughout the last decade I've become convinced that the operations review is a critical piece of a successful Lean or Agile transition and a vital component in developing organizational maturity.

Takeaways

- Operations reviews should be organization wide
- Operations reviews should focus on objective data
- Each department should report their own data
- Presentations should be kept short and should typically report metrics and indicators similar to those discussed in chapter 12
- Leading with financial information first underscores that the software engineering function is part of a wider business and that good governance is important
- A monthly cadence for operations reviews appears to be about right. More often is burdensome in time commitment and data gathering and preparation. Less often tends to reduce the value and undermine the nature of the meeting
- Meetings should be kept short, typically 2 hours
- Operations reviews should be used to provide a feedback loop and drive continuous improvement at the enterprise or business unit level
- Operations reviews show individual contributors how management can add value in their lives and what effective managers do
- Effective operations reviews build mutual trust between the managers and the workers
- External stakeholders attending operations review get an opportunity to see how the software engineering and IT group functions and to understand their issues and challenges. This fosters trust and collaboration
- Operations reviews should examine bad data and problems just as much as basking in success and extolling the virtue of teams with good results
- Hold meetings offsite seems to help focus the minds of attendees
- Providing food appears to encourage attendance
- The involvement of senior leaders communicates that the organization takes performance and continuous improvement seriously
- Signaling a serious interest in performance, continuous improvement and quantitative management is vital in developing a kaizen culture in the general workforce
- Operations review has been shown to lead directly to improved levels of organizational maturity
- Improvement suggestions should be captured as management action items and reviewed for progress at the beginning of the next and subsequent meetings
- Managers should be held accountable and should demonstrate follow through on suggestions

Chapter 15 Getting Started with Kanban

Getting started with Kanban isn't typical of process initiatives you may have undertaken in the past. It's important to lay the foundations for long term success. To do that, it's necessary to understand the goals behind using the Kanban approach to change. I subtitled this book, "Successful change management for technology organizations." I did this to underscore the point that the main reason for adopting Kanban is change management. Everything else is secondary.

Kanban accelerates organizational maturity and capability

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)[xx] within their Capability Maturity Model Integration (CMMI). It's been shown[xxi] 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[xxii] that Kanban merely optimizes something that needed to be changed. However, there is now considerable empirical evidence[xxiii] 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 Organizational Innovation and Deployment. When choosing to use Kanban as a method to drive change in your organization, you are subscribing to the view that it is 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.

Because of this, it is unlikely that you will drive adoption of Kanban through a planned transition initiative and a prescribed training program. It is true that some training will be necessary. It will be necessary to have team members and other stakeholders understand basics such as the relationship between WIP and cycle time and that strictly limiting the quantity of WIP will improve the predictability of cycle time. It may also be necessary to provide a brief overview of likely improvement opportunities such as bottlenecks, waste and variability. As these opportunities for improvement are uncovered more training in new skills and techniques may be needed. For example, if defects are a major source of waste, the development team may require training in techniques that will greatly reduce defects and improve code quality, such as, continuous integration, unit testing, and pair programming.

However, rather than waste too much time on education, in the first instance, it is more important that you gain a consensus around the introduction of Kanban and start using it. This chapter seeks to layout the foundations for a successful Kanban transition and provides you with a simple 12 step guide to getting started. The following chapters will then expand on these ideas with detail and explanation.

While our main goal with Kanban is to introduce change with minimal resistance there must be other goals. Change for the sake of change is pointless. These other goals should reflect genuine business needs such as predictable delivery with high quality. The goals given here are intended as examples. The specific goals for your organization may differ. Step 1 in your process should be to agree the goals for introducing Kanban into your organization.

The Primary Goal for our Kanban System

We are doing Kanban because we believe it provides a better way of introducing change. Kanban seeks initially to change as little as possible. So change with minimal resistance must be our first goal.

Goal 1. Optimize existing processes through changes introduced with minimal resistance

Secondary Goals for our Kanban System

We've learned that Kanban allows us to deliver on all 5 elements of the Recipe for Success (from Chapter 4.) However, we might want to word the goals slightly differently from the wording in the recipe and some of the points in the recipe need to be expanded to reflect that one point can help us deliver on more than one goal.

Goal 2. Deliver with High Quality

Kanban helps us focus on quality by limiting work-in-progress and allowing us to define policies around what is acceptable before a work item can be pulled to the next step in the process. These policies can include quality criteria. If, for example we set a strict policy that user stories cannot be pulled into acceptance test until all other tests are passing and bugs resolved, then we are effectively "stopping the line" until the story is in good enough condition to continue. With a new team doing Kanban we may not have such a strict rule but there should be some policies relating to quality that focus the team on developing working code with low numbers of defects.

Goal 3. Deliver a predictable cycle time by controlling the quantity of work-in-progress

We know that WIP is directly related to cycle time and that there is also a correlation between cycle time and a non-linear growth in defect rates[1] So it makes sense that we want to keep WIP small. It makes our life easier if we simply agree to limit it to a fixed quantity. This should make cycle times somewhat dependable and help us to keep defect rates low.

Goal 4. Give team members better quality of life through improved work/life balance

While employee satisfaction often gets lip service in most companies, it is seldom a priority. Investors and senior managers all too often take the view that resources are fungible and easily replaced. This reflects a cost-centric bias in their management or investment approach. It doesn't take into account the huge impact on performance that comes with a well motivated and experienced workforce. Staff retention is important. As the population of software developers ages, they care more about the rest of their lives. Many lament wasting their 20s locked up in an office slaving over a piece of code that failed to reach market expectations and became obsolete a short period later.

Work/life balance isn't only about balancing the number of hours someone spends at work with the number of hours they have available for their family, friends, hobbies, passions and pursuits. It is also about providing reliability. For example, a team member with a passion for art wants to take a painting class at the local middle school. It starts at 6.30pm and runs every Wednesday for 10 weeks. Can your team provide certainty to that individual that they'll be free to leave the office on-time each week in order to attend the class?

Providing a good work life balance will make your company a more attractive employer in your local market. It will help to motivate employees and it will give your team members the energy to maintain high levels of performance for months or years. It's a fallacy that you get top performance from knowledge workers when you overload them with work. It might be true tactically for a few days but it isn't sustainable beyond a week or two. It's good business to provide a good work/life balance by never overloading your teams with too much work.

Goal 5. Provide slack by balancing demand against throughput

[1] At the time of writing academics are beginning to investigate this relationship between cycle time and defect insertion rates. It is hoped that some academic papers will be published in 2010 to validate my belief that cycle time affects defect rates in a non-linear fashion.

While the 3rd element of the Recipe for Success – Balance Demand against Throughput – can be used to avoid overloading team members and allow them a reliable work/life balance, it has a second effect. It creates slack in the value chain. There must be a bottleneck in your organization. Every value chain has one. The throughput delivered downstream is limited to the throughput of the bottleneck, regardless of how far upstream it might be. Hence, when you balance the input demand against the throughput, you create idle time everywhere in your value chain with the exception of the bottleneck resource.

Most managers baulk at the idea of idle time. They've generally been trained to manage for utilization (or "efficiency" as it is often called) and inherently it feels like changes can be made to reduce costs if there is idle time. This may be true but it is important to appreciate the value of slack.

Slack can be used to improve responsiveness to urgent requests and to provide bandwidth to enable process improvement. Without slack team members cannot take time to reflect upon how they do their work and how it might be done better. Without slack they cannot take time to learn new techniques, to improve their tooling or their skills and capabilities. Without slack there is no liquidity in the system to respond to urgent requests or late changes. Without slack there is no tactical agility in the business.

Goal 6. Provide a simple prioritization mechanism that delays commitment and keeps options open

Once a team is capable of focusing on quality, limiting WIP and delivering often, and balancing demand against throughput, they will have a reliable, trustworthy, software development capability: an engine for making software! A "software factory" if you will! Once this capability is in place it would behoove the business to make optimal use out of it. To do this requires a prioritization method that maximizes business value, and minimizes risk and cost. Ideally a prioritizations scheme that optimizes the performance of the business (or technology department) is most desirable.

The software engineering and project management fields have been developing prioritization schemes since software projects began perhaps 50 years ago. Most of the schemes are simple. For example, "High, Medium, Low" provides three simple classifications. None of these have any direct meaning for the business. Some more elaborate schemes came into use with the arrival of Agile software development methods such as MoSCoW ("Must have", "Should have", "Could have, "Won't have.") Other methods such as Feature Driven Development featured a modified and simplified version of the Kano Analysis technique popular amongst Japanese companies. Yet others advocated strict enumerated order (1, 2, 3, 4, …) by business value or technical risk. The challenge with this latter scheme is that it often creates a conflict between the high risk items that should be prioritized first and the high value items that should also be prioritized first.

All of these schemes suffer from one fundamental problem. In order to respond to change in the market and evolving events, it is necessary to reprioritize. Imagine for example you have a backlog of 400 requirements prioritized in a strict enumerated order 1 to 400 and you are doing incremental delivery with an Agile development method in one month iterations. Every month you have to reprioritize the remaining backlog of up to 400 items.

In my experience, asking business owners to prioritize things is challenging. The reason for this is simple: there is so much uncertainty in the marketplace and the business environment. It's hard to predict the future value of one thing against another, when something might be needed and whether something else might be more valuable to have earlier. Asking a business owner to prioritize a backlog of technology system requirements is to ask them a very hard set of questions of which the answers are uncertain. When people are uncertain they tend to react badly. They may move slowly. They may refuse to cooperate. They may become uncomfortable and dysfunctional. They may simply react by thrashing and constantly changing their minds, randomizing project plans and wasting a lot of team time reacting to the change.

What is needed is a prioritization scheme that delays commitments as late as possible and provides a simple question that is easy to answer. Kanban provides this by asking the business owners to refill empty slots in the queue while providing them a reliable cycle time and due date performance metric.

We already have 6 lofty and valuable goals for our Kanban system and for many businesses that might be enough. However, I and other early adopters of Kanban have discovered that two other even loftier goals are both possible and desirable.

Goal 7. Provide a transparent scheme for seeing improvement opportunities enabling change to a more collaborative culture that encourages continuous improvement

When I first started to use Kanban systems, I believed in transparency on the work-in-progress, the delivery rate (throughput) and the quality because I understood that it built trust with customers and more senior management. I was providing transparency onto where a request was within the system, when it might be finished and what quality was associated with it. I was also providing transparency into the performance of the team. I did this to provide customers with confidence that we were working on their request and when it might be completed. In addition, I wanted to educate senior management on our techniques and performance and build their confidence in me as a manager and my team as a well formed professional group of software engineers.

There is a second order effect from all of this transparency that I hadn't predicted. While transparency onto work requests and performance is all very well, transparency into the process and how it works has a magical effect. It lets everyone involved see the effects of their actions. As a result people are more reasonable. They will change their behavior to improve the performance of the whole system. They will collaborate on required changes in policy, personnel, staff resourcing levels and so forth. This is discussed more fully in Chapter 13.

Goal 8. A process that will enable predictable results, business agility, good governance and the development of what the Software Engineering Institute calls a "high maturity" organization

For most senior business leaders that I speak to, this final goal really represents their wishes and expectations for their business and their technology development activities. Business leaders want to be able to make promises to their colleagues around the executive committee table, to their board of directors, to their shareholders, to their customers and to the market in general, and they want to be able to keep those promises. Success at the senior executive level depends a lot on trust and trust requires reliability.

In addition, they recognize that the world today is fast paced and change happens rapidly: new technologies arrive; globalization changes both labor markets and consumer markets causing huge fluctuations in demand (for product) and supply (of labor); economic conditions change; competitors change their strategies and market offers; and market tastes change as the population ages and becomes wealthier and more middle class. So business leaders want their business to be agile. They want to respond to change quickly and take advantages of opportunities.

Underlying all of this they want good governance. They want to show that investors' funds were spent wisely. They want costs under control and they want their investment portfolio risk spread optimally.

To do all of this they'd like to have more transparency into their technology development organizations. They'd like to know the true status of projects and they'd like to be able to help when it is appropriate. They want a more objectively managed organization that reports facts with data, metrics and indicators not anecdotes and subjective assessment.

All of these desires equate to an organization operating at what the Software Engineering Institute defines as Maturity Level 4 on its 5 point scale of capability and maturity in the Capability and Maturity Model Integration (CMMI). Level 4 and 5 on this scale are known as the "high maturity" levels. Very few organizations have achieved this level of maturity regardless of whether they have sought a formal SCAMPI appraisal or not. It is no wonder then that most senior leaders of large technology companies are frustrated by the performance of their software engineering teams.

Know the Goals and Articulate the Benefits

So now we have a set of goals for our Kanban system. We need to know these goals and be able to articulate them because before we start with Kanban we need to gain agreement with the stakeholders in our value chain. Kanban will change the way we interact with other groups in the business. If these stakeholders are to accept the changes then we must be able to articulate the benefits.

What follows is a prescriptive step by step guide to bootstrapping a Kanban system for a single value chain in your organization. This guide has been developed based on real experience and validated by several early adopters of Kanban both those who followed (roughly) these steps and were successful and those who recognized that their partial failure could have been prevented had they had this guide available at the time.

This guide is provided in part to draw attention to the difference between Kanban and earlier Agile development methods. Kanban requires a collaborative engagement with the wider value chain and middle (and perhaps senior) management right from the start. A unilateral grass roots adoption of Kanban without first building a consensus of managers external to the immediate team will have a limited success and deliver limited benefits to the business.

It has been pointed out to me that this set of steps can seem daunting and some people have remarked that had they read this they might have been put off trying Kanban altogether. I hope that the subsequent chapters will explain how to engage on each of these steps and provide you with useful advice learned from experience in the field.

Steps to Get Started

1. Agree a set of goals for introducing Kanban
2. Map the value stream, the sequence of all the actions in the development organization carried out to fulfill a customer's/stakeholder's request. (explained in Chapter 6)
3. Define some point where you want to control input - define what is upstream of that point and who the upstream stakeholders are (explained in Chapter 6.) For example, do you wish to control requirements arriving to the design team pre-production? The upstream stakeholders might be product managers.
4. Define some exit point beyond which you don't intend to control - define what is downstream of that and who the downstream stakeholders are (explained in Chapter 6.) For example, maybe you don't need to control the delivery fulfillment of the product.
5. Define a set of work item types based on the types of work requests that come from the upstream stakeholders (explained in Chapter 6.) Do you have some item types that are time-sensitive and others that are not? If so, then you may require some classes of service (explained in Chapter 21)
6. Meet with the upstream and downstream stakeholders – this might be one big meeting, or it might be lots of little meetings. (explained in more depth later in Chapter 5)
    a. Discuss policies around capacity of the bit of the value stream you want to control and get agreement on a WIP limit (explained in Chapter 10)
    b. Discuss and agree an input coordination mechanism such as a regular prioritization meeting with the upstream partners (explained in Chapter 9)
    c. Discuss and agree a release/delivery coordination mechanism such as a regular software release with the downstream partners (explained in Chapter 8)
    d. You may need to introduce the concept of different classes of service for work requests (explained in Chapter 21)
    e. Agree a cycle time target for each class of service of work items. This is known as a service level agreement (SLA) and is explained in Chapter 11.
7. Create a board/card wall to track the value stream you are controlling (explained in Chapter 6 and Chapter 7)
8. Optionally create an electronic system to track and report the same (explained in Chapter 6 and Chapter 7)
9. Agree to have a standup meeting every day in front of the board with the team (invite upstream and downstream stakeholders but don't mandate their involvement) (explained in Chapter 7)

10. Agree to have a regular operations review meeting for retrospective analysis of the process (invite upstream and downstream stakeholders but don't mandate their involvement) (explained in Chapter 14)
11. Educate the team on the new board, WIP limits, and pull system. Nothing else in their world should have changed. Job descriptions are the same. Activities are the same. Handoffs are the same. Artifacts are the same. Their process hasn't changed other than you are asking them to accept an WIP limit and to pull work rather than receive it in a push fashion
12. Start using your new Kanban process

Kanban Strikes a Different Type of Bargain

Kanban requires the software development team to strike a different bargain with its business partners. To understand this we must first understand the typical alternatives that are in common use.

Traditional project management makes a promise based on the triple constraint of scope, schedule and budget. After some element of estimation and planning, a budget is set aside to provide resources, and a scope of requirements and a schedule are agreed.

Agile project management meanwhile doesn't make such a bold rigid commitment. There may be an agreed delivery date some months in the future but the precise scope is never agreed. Some high level definition of scope may be agreed but fine details are never locked down. A budget (or burn rate) may be agreed in order to provide a fixed set of resources. The Agile development team proceeds in an iterative fashion delivering increments of functionality in short time-boxed iterations (or sprints.) Typically these are one to four weeks in length. At the beginning of each of these iterations some planning and estimation are performed and a commitment is made. The scope is often prioritized and it is understood that if the team cannot make the commitment, it is scope that will be dropped and the delivery date will be held constant. At the iteration (or time-box) level, Agile development looks very similar to traditional project management. The only key difference is the explicit understanding that scope will be dropped if something has to give, where a traditional project manager may choose to slip the schedule, add resources, drop scope or some combination of all three.

Kanban strikes a different type of bargain. Kanban does not seek to make a promise and commitment against something which is uncertain. A typical Kanban implementation involves agreement that there will be a regular delivery of high quality working software – perhaps every two weeks. The external stakeholders are offered complete transparency into the workings of the process and, if they want, daily visibility of progress. Equally, they are offered frequent opportunities to select the most important new items for development. The frequency of this selection process is likely to be more frequent than the delivery rate – typically, once per week, though some teams have achieved on-demand selection or very frequent rates such as daily or twice per week.

The team offers to do its best work and deliver the largest quantity of working software possible and to make ongoing efforts to increase the quantity, frequency and lead time to delivery. In addition to offering the business incredible flexibility to select items for processing in very small quantities, the team may also offer the business additional flexibility on priority and importance by offering several classes of service for work. This concept is explained in Chapter 21.

Kanban does not offer a commitment on a certain amount of work delivered on a certain day. It offers a commitment on reliable regular delivery, transparency, flexibility on prioritization and processing, and a commitment to continuous improvement on quality, throughput, frequency of delivery and lead time. Kanban offers to commit to things that customers truly value. In exchange the team is asking for a long term commitment from the customers and value chain partners. A commitment to have an ongoing business relationship where the software development team strives to constantly improve the level of service through improved quality, throughput, frequency of, and lead time to, delivery.

The traditional approach to forming a commitment around scope, schedule and budget is indicative of a one-off transaction. It implies that there will be no ongoing relationship. It implies a low level of trust.

The Kanban approach is based on the notion that the team will stay together and engage in supplier relationship to a customer for high quality software over a long period of time. The Kanban approach implies lots of repeat business. It implies a commitment to a relationship not merely to a piece of work. Kanban implies a high level of trust is desired between the software team and its value stream partners. It implies that everyone believes they are forming a long term partnership and they want that partnership to be highly effective.

A Kanban commitment is asking everyone in the value chain to care about the performance of the system – to care about the quality and quantity of software being delivered, the frequency of delivery and the cycle time to deliver it. Kanban asks the value chain partners to commit to the concept of true business agility and to agree to work collaboratively to make that happen. This significantly differentiates Kanban from earlier Agile approaches to software development.

By taking the time to establish the Kanban bargain with upstream and downstream stakeholders, you are establishing an underlying commitment to system level performance. You are establishing the foundation for a culture of continuous improvement.

Striking the Kanban Bargain

A critical piece of a successful Kanban implementation is the initial negotiation of this different type of bargain. What’s going on during these initial negotiations is the establishment of the rules of the collaborative game of software development that will be played going forward. It’s vital that value stream partners are involved in agreeing these rules because it will be necessary for them to stick to them if the game is to be played fairly and the outcome reflect the goals and intent.

Step 6 in our 12 step process for introducing Kanban suggests that we meet with upstream stakeholders such as marketing or business people who provide requirements, and downstream partners such as systems operations and deployment teams or sales and delivery organizations. We need to agree with them policies around WIP, prioritization, delivery, classes of service and cycle time. The set of policies we agree with these partners will define the rules of our collaborative game of software development. It's hard to treat each of the 5 elements in isolation as they are essentially inter-related. So while we understand that we must set policy around each of the five elements, the negotiations are likely to be quite circular in nature as the participants iterate on options. For example, if a proposed cycle time target is unacceptable, it may be possible to introduce a different class of service to offer a shorter cycle time for certain types of work request. The five elements of WIP, prioritization, delivery, classes of service and cycle time, provide levers that can be pulled to affect the performance of the system. The skill is in knowing how to pull those levers and tradeoff options to devise an agreement that will work effectively.

I met a development manager in Denmark who told me that his developers work on 7.5 tasks on average simultaneously. This is clearly undesirable. I wonder if anyone would truly believe this level of multi-tasking to be appropriate. If I were him, I would use this fact as a starting point for my negotiations. I'd open up the conversation by stating that on average team members were working on 7.5 things in parallel. I would point out what this does to cycle time and predictability and I'd invite my colleagues – the other stakeholders – to suggest what a better number might be. Some of them may suggest that only 1 item per person is the best idea. And sure enough it may be but it's a very aggressive choice. What if something got blocked? Would it not be good to have an alternative to switch to? So perhaps another person will suggest that 2 things in parallel is the right answer. Some may argue for 3 things. However, the range of suggestions is likely to lie between 1 and 3. If the team has 10 developers and you can gain a consensus around a maximum of 2 things in process per person then you have an agreement on a WIP limit of 20 for the development team.

There are other alternatives. Perhaps you want teams to work in pairs of programmers, so 2 things per pair with 10 developers would mean a WIP limit of 10. Alternatively, you may be using a highly collaborative method such as Feature Driven Development or Feature Crews where small teams of up to 5 or 6 people work on single Minimum Marketable Features or User stories or batches of small Features (as in FDD) known as a Chief Programmer Work Package (CPWP). An FDD team may agree to limit CPWPs to 3 over a team of 10 developers. [A CPWP is typically optimized for development efficiency based on architectural analysis of the domain and contains between 5 to 15 very fine-grained features.]

So we've had a conversation about the WIP limit with our stakeholders. We did this by discussing what a reasonable expectation might be for multi-tasking and relating it to reliability of delivery and cycle time expectations. Getting our partners to agree on the WIP limits is a vital element. While we could unilaterally declare WIP limits, by involving other stakeholders and forming a consensus we establish a commitment to the rules of our collaborative game. At some point in future this commitment will prove invaluable. There will be a day in the future when our partners ask us to take on some additional work. They will do this because something is important and valuable. Their reasons and motives will be genuine. When they do so, we will be able to respond by asking them to acknowledge that we have an agreed WIP limit. It is likely that our system will be full and accepting another item, however, important will break that limit. So our answer should be,

> "Yes, we'd love to accept this new work as we realize it is very important to you. Equally, you know and understand that we have an agreed WIP limit. You were part of that decision and you understand why we made it. We want to be able to process requests reliably and in a timely manner. In order to take on your request we will need to put something else aside. Which one of the current items in progress would you prefer that we drop in order to start your new item?"

If we hadn't included our partners in the WIP limit decision then we'd be unable to have this discussion. They would simply continue to push us. Our WIP limited pull system would be broken and our organization would be slipping down the precipitous back to a push system.

If we are to have a truly successful collaborative game of Kanban software development then the rules for that game must be agreed by consensus amongst all the stakeholders.

We also want to agree a mechanism for prioritization. Typically, we are looking for an agreement to have a regular prioritization meeting and a mechanism for how new work will be selected. We can have this conversation by asking, "If we were to ask you a very simple question such as 'Pick two things you need delivered 42 days from now' how often would you be able to meet with us to have such as discussion? We'd hope that the meeting would take no longer than 30 minutes." Because you offering to make the meeting extremely focused, and you are asking a very direct question and suggesting that the time commitment is minimal, you will typically find that upstream partners are quite willing to be very collaborative. It's not unusual to get agreement on a weekly meeting. More often is common in fast moving domains such as media where the release cycles may be very frequent.

Now we must agree a similar thing with downstream partners. A delivery cadence that makes sense is very specific to a domain or situation. If it's web-based software, we have to deploy to a server farm. Deployment involves copying files and perhaps upgrading a database schema and migrating data from one version of the schema to another. This data migration will probably have its own code and it will take its own time to execute. The total deployment time will be a factor of how many servers, how many files to copy, how long it takes to gracefully pull systems down and reboot them, how long data takes to migrate and so forth. Some deployments may take minutes, others hours or even days. In other domains, we may need to manufacture physical media such as DVDs and package them in boxes. We need to distribute the media through physical channels to distributors, dealers, retailers or existing corporate customers. There may be other elements involved such as printing of physical manuals, or training of sales and support staff. We need to devise a training program for these people.

For example, in 2002, I was involved in the release of the first the staged upgrades to the Sprint PCS mobile phone network. This first upgrade on the road to 3G technology was called 1xRTT. It was launched on the market as PCS Vision. The launch involved release of around 15 new handsets with a target of 16 new features that utilized the high speed data capabilities of the new network. Sprint had a retail network across the United States that employed 17,000 people. They had a similar number of folks in call centers who took customer care calls from users. Both the retail sales channel and the customer care associates had to be trained to support the launch of the new service. I jokingly suggested that the best way to do this would be shut everything down for two days, fly everyone into Kansas City for a night and rent out the Kansas City Chiefs stadium, where we could deliver the 2 hour Powerpoint presentation on the big screens at either end of the stadium. This might have been the most efficient way but it was totally unacceptable for several reasons. Our customers would hardly accept a 48 hour outage in support while we trained our operators on the next generation of technology. And losing two days of sales revenue from retail wouldn't have helped our annual revenue targets.

So a training program was devised and train-the-trainer education was delivered. A program of training for retail staff in regions was devised and a similar one for call centers. Trainers were sent out into the field for 6 weeks to train small groups of people as they came off shift. The cost of delivering the training was huge. The time commitment at 6 weeks was very significant and the half life of the training in the memories of the workforce was also about 6 weeks. If we missed the launch window for the new service then the training would have to be repeated and a minimum 6 weeks of further delay in deployment would be incurred.

So, if your domain is like the telephone network, you will know that the release cadence will be infrequent. When the transaction costs of making a release involve 6 weeks of training then releasing any more frequently than annually is unacceptable.

The outcome you desire is the most frequent release cadence that makes sense. So start by asking, "If we give you high quality code, with minimal defects to release, and it comes with adequate warning, transparency into its complexity, and reliability of delivery, how often could you reasonably deploy it to production?" This will provoke some discussion around the definitions you've used and some reassurance will be required. However, you should push for a result that maximizes business agility without over-stressing any part of the system.

When it comes to our conversation about cycle time, it helps to have some historical data on past performance. Ideally, we want to have cycle time and engineering task time data. Like the Microsoft example from Chapter 3, we knew that cycle times were around 125 days for severity 1 defects and 155 days for other severities. The first thing that should strike you about this is that there are two classes of service. Severity 1 defects have historically received some form of preferential treatment. There may never have been any formality around this but the net outcome is that severity 1 defects were being processed faster.

Knowing this may enable us to offer up two different classes of service from the get go. We may suggest to the external stakeholders that we will adopt two classes of service and have separate cycle time targets for each.

Equally, we also knew from historical data that average engineering effort was 11 days and that the high end was 15 days. So we chose to suggest a 25 day cycle time. There was no more science involved. Now, imagine the psychological effect of this. The business was used to a performance in the 4 to 5 months range and we had just offered 25 days. The difference is we'd offered 25 days of cycle time not including any initial queuing and the 155 days was a lead time that did involve queuing. Nevertheless, it sounds like a fantastic improvement. It is not surprising that the business agreed.

Other alternatives exist. You might take the historical engineering effort data and place it in a statistical process control chart. This will give you an upper control (or 3 sigma) limit. You may then want to buffer that upper limit number with a small amount for safety that will absorb external variations. If you are doing this, you should be transparent with partners and show them how you are calculating the numbers.

Another alternative would be to ask what level of responsiveness the business actually needs. This would be best done in the context of a set of classes of service. For example, if the business answers, "We need delivery in 3 days." You might reply, "Does everything need to be delivered in 3 days?" The answer is almost certainly "No." That would give you the opportunity to ask for a definition of types of request that need delivered within 3 days. You can then create a class of service for this type of work. Then repeat the process for the remaining work. The outcome should be the stratification of work requests into several bands for which a class of service can be devised. It is likely that each of these bands will contain work that exhibits the same shape of function for cost of delay. The detail around creating classes of service and the concept of cost of delay functions is explained in full in Chapter 11.

The cycle time you are agreeing for each class of service should be presented as a target rather than a commitment. You will commit to doing your best to achieve the target time and to report due date performance against the cycle time target in the service level agreement (SLA) for each class of service. In some situations, there may not be sufficient trust to allow agreement that cycle time in the SLA is a target rather than a commitment. If you do need to agree that cycle time in the SLA represents a commitment then you should buffer the target with a margin for safety. This will highlight directly that a lower level of trust results in a direct economic cost.

The exit criteria for your partner discussions is that: you have a consensus on WIP limits along the value-stream; you have an agreement on prioritization coordination and the method to be used; you have a similar agreement on delivery coordination and method; and you have a definition of a set of service level agreements that include a target cycle time, for each class of service.

Takeaways

- There are at least 8 possible goals for introducing Kanban to your organization
- Improved performance through process improvements introduced with minimal resistance
- Deliver with High Quality
- Deliver a predictable cycle time by controlling the quantity of work-in-progress
- Give team members a better life through improved work/life
- Provide slack in the system by balancing demand against throughput
- Provide a simple prioritization mechanism that delays commitment and keeps options open
- Provide a transparent scheme for seeing improvement opportunities enabling change to a more collaborative culture that encourages continuous improvement
- A process that will enable predictable results, business agility, good governance and the development of what the Software Engineering Institute calls a “high maturity” organization
- It’s important to define your goals and be able to articulate the benefits of introducing Kanban in order to gain a consensus agreement with other stakeholders
- Follow the 12 step guide to bootstrapping a Kanban process
- Kanban strikes a different bargain with external stakeholders and business owners. It is a bargain based on an assumption of a long term relationship and a commitment to system level performance
- Including external stakeholders to form an agreement on the basic elements of the Kanban system makes them collaborators
- Basic policies on WIP limits, cycle time targets, classes of service, prioritization and delivery represent the rules of the collaborative game of software development
- Involving external stakeholders as collaborators to agree the rules of the game will enable collaborative behavior later when the system is put under stress

Section Four
Making Improvements

Chapter 16 Three Types of Improvement Opportunity

Chapters 5 through 15 have described how to build and operate a kanban system and adopt the Kanban approach to change management and improvement. The remainder of the book will describe how to recognize opportunities for improvement, what to do about them and how to choose between them.

Bottlenecks, Waste Elimination, and Variability Reduction

Improvement opportunities come in three main types. Each of these types has been fully explored and developed in its own body of knowledge. Each has its own school of continuous improvement. With Kanban, I have chosen to synthesize all three and to provide an overview of how to recognize these improvement opportunities and details on how to implement improvements. Each of the three schools of continuous improvement described below has its own set of thought leaders, its own conferences, its own canon of knowledge and experience, and its own set of followers. You company may subscribe to one or more of these schools. Being able to show how the techniques of Kanban can provide opportunities for improvement in your organization's favorite flavor may be an advantage. Knowing that you have a wide set of improvement paradigms and tools to choose from should provide greater flexibility to make change.

Those widely familiar with continuous improvement methodologies may choose to skip the rest of this chapter and move straight to Chapter 17. Those who wish an overview of available methods and some background on the literature and history may find the remainder of Chapter 16 valuable.

Theory of Constraints

The Theory of Constraints was developed by Eli Goldratt and first published in his business novel, The Goal, in 1984. Over the last 25 years, The Goal has gone through several revisions and the theoretical framework known as The Five Focusing Steps has become more obvious in the text of the recent editions.

The Five Focusing Steps is the basis for continuous improvement in the Theory of Constraints. It is known as a POOGI (a Process Of OnGoing Improvement). The Theory of Constraints (or TOC) is full of acronyms. Strangely, The Five Focusing Steps is the exception.

In the 1990s the Theory of Constraints evolved a method for root cause analysis and change management known as the Thinking Processes (or TP). The reason for this development was the discovery amongst the TOC consulting community that their constraint to achieving improvement with clients was change management and resistance to change.

It seemed that The Five Focusing Steps only appeared to work well for flow problems and many workplace challenges did not neatly fit into the flow paradigm, or so it seemed. So TP was invented. The professional qualification and training program for TOC consultants was changed from a class in the use of The Five Focusing Steps and its applications such as Drum-Buffer-Rope, to a class in TP. Hence, many in the TOC community when referring to TOC are in fact referring to TP and not The Five Focusing Steps. In my observation attending TOC conferences, use of The Five Focusing Steps amongst the TOC community has become somewhat of a lost art.

The TOC community tended to accept paradigms as they were established rather than challenge them. Hence, the TOC solution for project management, Critical Chain, evolved around the incumbent project management paradigm of the triple constraint (scope, budget, and schedule) and the dependency graph model for scheduling the tasks in a project. No one challenge the incumbent model. No one until I published my first book, Agile Management for Software Engineering, challenged the project management paradigm and suggested it was better to model projects as a value-stream and flow problem and apply The Five Focusing Steps. By doing it was then possible to use the whole Lean body of knowledge and this led to the development of Kanban.

I have argued that any process or workflow that involves division of labor can be defined as a value stream. And any value stream can be observed to have flow. Lean and the Toyota Production System are really built of this assumption. And if any value stream has flow then The Five Focusing Steps can be applied to it. And hence, The Five Focusing Steps is a perfectly satisfactory POOGI and TP is not required unless you are using it as a change management tool. I personally have not developed an affinity with TP. My preferred change management tool is Kanban as described in this text.

Five Focusing Steps

The Five Focusing Steps is a simple formula for a process of ongoing improvement. It states:

1. Identify the constraint
2. Decide how to exploit the constraint
3. Subordinate everything else in the system to the decision made in 2
4. Elevate the constraint
5. Avoid inertia, identify the next constraint and return to step 2

Step one is asking us to find a bottleneck in our value stream.

Step two asks us to identify the potential throughput of that bottleneck and compare that to what is actually happening. As you will see, the bottleneck is rarely or never working at its full capacity. So ask what would it take to get the full capacity out of the bottleneck? What would we need to change to make that happen? This is the "decide" part of step 2.

Step 3 asks us to make whatever changes are necessary to implement the ideas from step 2. This may involve making additional changes elsewhere in the value-stream in order to get the maximum capacity from the bottleneck. This action of maximizing the bottleneck's capability is known as "exploiting" the bottleneck.

Step 4 suggests that if the bottleneck is operating at full capability and is still not producing enough throughput that its capability needs to be enhanced in order to increase throughput. Step 4 asks us to implement an improvement to enhance capability and increase throughput sufficiently so that the current bottleneck is relieved and the system constraint moves elsewhere in the value stream.

Step 5 requires that we give the changes time to stabilize and then identify the new bottleneck in the value-stream and repeat the process. The result is a system of continuous improvement where throughput is always increasing.

If The Five Focusing Steps is institutionalized properly then a culture of continuous improvement throughout the organization will have been achieved.

Chapter 17 will explain how to identify and management bottlenecks using The Five Focusing Steps.

Lean, TPS & Waste Reduction

Lean emerged in the early 1990's after the seminal text, The Machine That Changed the World, by Womack, Jones and Daniels, described from an outsider's empirical observation how the Toyota Production System (TPS) worked. The early literature on Lean had some flaws. It failed to identify the management of variability that is inherent to TPS and was learned and adapted from Deming's System of Profound Knowledge. Lean also fell foul of misinterpretation and over-simplification. Many Lean consultants jumped on the concept of Waste reduction (or elimination) and taught Lean as purely a waste elimination exercise. In this anti-pattern of Lean, all work activities are classified as value-added on non-value added. The non-value added, wasteful activities, are further sub-classified into necessary and unnecessary waste. The unnecessary are eliminated and the necessary are reduced. While this is a valid use of Lean tools for improvement, it tends to sub-optimize the outcome for cost reduction and leaves value on the table by not embracing the Lean ideas of Value, Value Stream, and Flow.

Kanban enables all aspects of Lean thinking and provides the tools to optimize an outcome for value through a focus on flow management as well as waste reduction.

Chapter 18 will explain how to identify wasteful activities and what to do about them.

Deming and Six Sigma

W. Edwards Deming is generally thought of one of the three main fathers of the Quality Assurance movement of the 20th Century. However, his contribution was considerably greater. He evolved the use of Statistical Process Control (SPC) and developed it into a management technique he called the System of Profound Knowledge. A system intended to prevent managers from making poor quality, even if intuitive, decisions, and replacing them with statistically sound, objective, often counter-intuitive, better decisions. Deming is occasionally mentioned as perhaps the most important management scientist of the 20th Century and in my opinion this is greatly deserved. So his contribution extended from SPC, through Quality Assurance to Management Science.

Deming had a significant influence on Japanese management philosophy around the middle of the 20th Century and his influence and that of SPC and the System of Profound Knowledge is a key pillar of TPS.

While some highly mature Kanban teams, for example at investment bank, BNP Parisbas in London, have adopted the use of SPC, SPC is beyond the scope of this book and will be addressed in a future text on advanced Kanban techniques.

However, the principles of understanding the variation in systems and work tasks that underpin SPC are very useful. Deming's predecessor Walter Shewhart classified variability in task performance into two categories, chance and assignable cause. Deming later renamed these to common and special cause. In the 2nd edition of The New Economics he admitted that this was for "largely pedagogical reasons." There was no specific innovation in the changing of the terms. Understanding variation and how it impacts performance and being capable of classifying it into the two categories and learning the appropriate management actions to take based on the type of variation is a necessary management skill and core to a program of continuous improvement. Both Lean and Theory of Constraints rely heavily on an understanding of variation in order to enable improvement even if those improvements are cast as bottleneck management or waste reduction.

Chapter 19 will explain how to recognize common and special cause variations and suggest ideas for appropriate management action. Chapter 20 will further elaborate on this by describing how to build an issue management capability that responds to special cause variations with the goal of eliminating such issues as quickly as possible in order to maintain flow and maximize value delivery. Note that without a knowledge of and focus on management of variability a focus on flow is lost. Lean without Deming's ideas is Lean without an understanding of variation and by implication is Lean without a focus on maintaining flow. Given that the early Lean literature did not include an understanding of variation and any references to Deming's System of Profound Knowledge, it is easy to understand the root cause of the anti-pattern of teaching Lean as a process of waste reduction only.

While Deming's ideas were embedded into TPS in Japan at the shop floor level where SPC and the System of Profound Knowledge were employed to identify local improvement opportunities, another body of knowledge developed in the United States based around Deming's ideas. Six Sigma started at Motorola but really came of age when it was adopted at GE under Jack Welsh's leadership.

Six Sigma employs SPC to identify common and special cause variation and uses a process similar to that described by Deming to eliminate special cause variations at their root cause and prevent them recurring and to reduce common cause variation and make a process, workflow or system more predictable.

Unlike TPS which is all about shop floor initiatives run by empowered workers implementing small kaizen events by the hundred and thousand, Six Sigma has developed into a much more low trust, command and control method, that tends to involve far fewer improvement opportunities, generally implemented at a more strategic level, and run as specific projects in their own right. The project leader carries the title Black Belt and has generally had years of training in the methodology to earn his status as a Black Belt. Because Kanban embraces the ideas of Deming and provides the instrumentation and transparency to see variability and its affect and to classify that variability into common or special cause, Kanban can be used to enable either a Kaizen style improvement program or a Six Sigma style improvement program.

Fitting Kanban to your Company Culture

Hence, if your company is a Six Sigma company, Kanban can help you run Six Sigma initiatives in the software, system or product development or IT organization. If your company is a Lean company, Kanban is a natural fit. It can enable an entire Lean initiative in your software, system or product development or IT organization. If you company subscribes to and uses the Theory of Constraints, Kanban can enable an entire constraints management (bottleneck removal) program in your software, systems or product development or IT organization. However, you might need to recast the pull system implementation as a Drum-Buffer-Rope implementation rather than refer to is as kanban pull system. As Kanban developed from an earlier Drum-Buffer-Rope implementation, I know this will work. However, discussion of the specifics of how to model the value stream and set WIP limits for the Buffer and Rope are beyond the scope of this text.

Takeaways

- Kanban supports at least 3 types of continuous improvement methods: Constraint Management (bottleneck removal); Waste Reduction; Variability Management (as well as SPC and the System of Profound Knowledge).
- Kanban enables the identification of bottlenecks and a full implementation of The Five Focusing Steps from the Theory of Constraints
- Kanban enables visualization of wasteful activities, failure load and inventory depreciation and can be used to enable a full Lean initiative within the software, system, product development, or IT organization
- Kanban provides the instrumentation for use of W. Edwards Deming's Theory of Profound Knowledge and Statistical Process Control. It can be used to drive a Kaizen initiative or a Six Sigma initiative

Chapter 17 Bottlenecks & Non-instant Availability

Washington SR-520 is the freeway that links Seattle with its northeastern suburbs of Kirkland and Redmond. It provides the main commuter artery for suburban dwellers who work in the city center and for employees of Microsoft and the other high technology firms based in those suburbs, such as AT&T, Honeywell and Nintendo, who live in the city and commute in the opposite direction each weekday. For a total of 8 hours each day the road is a severe traffic bottleneck in both directions. If you stand on the bridge crossing the freeway on NE 76th Street in the small suburb of Medina just up the street from Bill Gates' estate on the shores of Lake Washington, in late afternoon, looking eastward, you will see the westward, city-bound traffic backing up and crawling slowly up the hill from Bellevue before merging down to two lanes to cross the floating bridge into Seattle. The speed of traffic coming up the hill is about 10 miles per hour and the flow is ragged with vehicles constantly slowing and stopping. If you cross the street and look westward towards the skyscrapers of Seattle's downtown, the Spaceneedle and the Olympic Mountains in the far distance, you'll see the traffic moving smoothly away from you at almost 50 miles per hour. What magic is happening right beneath your feet that the traffic speed changes so dramatically and its flow changes from ragged to smooth?

Just before the bridge the road narrows from 3 lanes to 2 before crossing the lake on the pontoon bridge. The right most lane of the freeway is a high occupancy vehicle (HOV) lane that requires vehicles with 2 or more passengers. It is frequented by the many public service buses that shuttle commuters to and from the city and some private cars. The action of these vehicles merging into the other traffic is enough to cause disruption and a slowdown of, and backup in traffic. In the several miles that precede the bridge several other roads merge into the freeway adding additional volume of traffic to what is already a busy road at peak times. The net effect is ragged flow and very slow speeds.

In traffic safety, the planners worry about the distance between cars. Ideally they want enough distance for cars to react to changes and to stop safely if needed. This distance is related to speed and reaction time. The legally advised "distance" between vehicles is recommended as 2 seconds. In Lean language this is the ideal takt time between vehicles. Hence, if we have two lanes and 2 seconds between vehicles the maximum throughput of the road is 30 vehicles per lane per minute or 60 vehicles per minute. This is true regardless of the speed of the vehicles though the rules breakdown a little at extreme limits for very slow speeds and for super excessive speeds well in excess of the 50 miles per hour limit enforced on the SR-520. For practical purposes the throughput (referred to confusingly as capacity in traffic management) is 3600 vehicles per hour.

However, as you stand on the bridge and count the number of cars passing under it on a typical afternoon around 5pm, you'll observe that less than 10 cars per minute are entering the floating bridge towards Seattle. Despite the heavy demand the road is operating at less than one fifth of its throughput potential! Why?

The pontoon bridge over Lake Washington is a bottleneck. We all intuitively understand this concept. The width of the neck of a bottle controls the flow of liquid from the bottle. A wide neck and we can pour quickly but often with greater risk of spillage. With a narrow neck the flow is slower but can be more precise. Bottlenecks restrict our potential for throughput, in this example to 60 cars per minute or 3600 per hour.

In general, a bottleneck in a process flow is anywhere that a backlog of work builds up waiting to be processed. In the example of the SR-520 that is a queue of vehicles occasionally backed up to Overlake 7 miles east. In software development, it can be any backlog of un-started work or work-in-progress: requirements waiting for analysis; analyzed work waiting for design, development and testing; tested work waiting for deployment; and so forth.

As discussed, the SR-520 only delivers about 20% of its potential at peak times when it is needed most. For a full explanation of this we need to understand how to fully exploit the potential of a bottleneck and the effect that variability has upon that potential. These concepts are explained here in chapter 17 and later in chapter 19.

Capacity Constrained Resources

The SR-520 at NE 76th Street Bridge is a capacity constrained bottleneck. Its capacity is 60 cars per minute in two lanes. Prior to this the road is 3 lanes wide and traffic is forced to merge together in order to cross the lake on the aging pontoon bridge which was designed 50 years ago with only 2 lanes. At the time, this was plenty of capacity and the bridge was not a bottleneck as the eastern suburbs were small villages and commuting across the water was rare and in those days only towards the city and not in reverse as is common today.

Elevation Actions

In this respect, as a capacity constrained bottleneck, the SR-520 may be similar to a user experience designer on a software team who is responsible for designing all the screens and dialogs with the user. She works flat out but still her throughput is insufficient to meet the demand placed on her by the project. The natural reaction of most managers in this situation is to hire another person to help. In Eli Goldratt's Theory of Constraints this is known as "elevating the constraint" - adding capacity so that the bottleneck is removed.

In our example of SR-520, this is the equivalent of replacing the floating pontoon bridge across Lake Washington with a new bridge that features 3 lanes of traffic each way. To keep all things equal it should be a bridge featuring 1 HOV lane and a bicycle lane as well as two lanes open to all traffic. This is, in fact, the course of action that the State of Washington Department of Transport is pursuing. The bridge is costing many hundreds of millions and taking a decade to implement. At the time of writing construction has not started.

It turns out that elevating a capacity constrained resource ought to be the line of last resort. Increasing the capacity of a bottleneck costs both time and money. If for example, we have to hire another user experience designer, we need to find the budget to pay this new person and the budget to fund the hiring process including any fees we may pay agents for referrals. We will slow the progress on our current project while we review resumes and interview candidates. Our most precious resource, our capacity constrained user experience designer will be asked to take time out from real project work to read resumes, select candidates and then interview them. As a result, her capacity to complete designs is reduced and the potential throughput for our whole project is reduced. This is partly why Fred Brooks' "Law" states that adding people to a late project only makes it later. While Brooks' observation was anecdotal and we can now make a much more scientific explanation of this phenomenon, the software industry has understood the concept that hiring more people slows you down for at least the last 35 years.

Exploitation/Protection Actions

Rather than jump immediately to exploitation, spend time and money, while slowing things down, it is better to first find ways of fully exploiting the capacity of the bottleneck resource. For example, the SR-520 is observed to have a throughput only 20% of its potential at peak times. What actions might be taken to improve that throughput? Let's dream for a moment. If the throughput of the road at rush hour were achieving its potential of 3600 vehicles per hour, would it be necessary to replace the existing bridge with a new one? Would journey times be sufficiently short that Washington State taxpayers (like the author) might prefer to spend their tax dollars on some other more important and pressing matter? Such as more books in local schools? Perhaps!

So how would you go about exploiting the true potential of the road? The source of the problems actually lies with the humans driving the vehicles. Their reaction times and the actions they take are highly variable. As cars merge from the HOV lane, this causes vehicles in the center lane to slow to make space for a merging car. Some of the drivers react more slowly than others. Some stand on the breaks more vigorously than others and the net effect is that traffic backs up unpredictably. Some drivers disturbed by the fluctuation in the lane ahead and the reduced speed in comparison to the neighboring left lane, decide to switch and jump across merging into the left lane. The same thing is then repeated. So all the vehicles have slowed but the speed does not really affect throughput. It's the gap between the vehicles that is most important. What is desired is a smooth flow of traffic with a 2 second gap between vehicles. [Note: in some parts of California 1.4 seconds is observed as normal though not ideal from a safety perspective] However, the human element means that vehicles do not slow and accelerate smoothly and the gaps concertina. The reaction time of the individuals to press the accelerator and brake pedals, and the reaction times of the engines and transmissions and gear boxes in the vehicles means that gaps continue to widen as traffic builds up. Variability in the system is has a huge impact on throughput.

Fixing this problem for the SR-520 takes us into fantasy land in terms of vehicle control though some German manufacturers have experimented with such systems. Systems that use radar or lasers to judge the distance between vehicles and keep traffic moving in a smooth convoy, can remove the variability that is observed on the SR-520. Such systems have the ability to slow whole chains of vehicles smoothly maintaining the gap between them and as a result the throughput of traffic remains high. However, eliminating variability from private vehicles driven by the occupants has its limits. If you want low variability transport you have to chain the passenger cars together and put them on rails. That's fundamentally why mass rapid rail transport is more effective at moving large quantities of people quickly.

The good news is that in our office, our capacity constrained resources are affected by variability that we can do something about. We've talked a lot about coordination activities and transaction costs of doing value-added work in this book. If we have a capacity constrained user experience designer then we can seek to keep that individual busy working on value-added work by minimizing the non-value-added (wasteful) activities required of that person.

In one example, I had a capacity constrained test team in 2003. To maximize the exploitation of their capacity, I looked for other slack resources and found them with the business analysts and project manager. The test team was relieved of bureaucratic activities such as time sheet completion. They were also relieved of planning future projects. We allowed analysts to develop test plans for future iterations and projects while the testers busied themselves with performing tests on the current work-in-progress.

A long term fix may have been to invest heavily in test automation. The keyword in the last sentence was "invest." If you find yourself saying "invest" you are generally talking about an elevation action. Adding resources is not the only way to elevate capacity. Automation is a good and natural strategy for elevation. The Agile Software Development community has done a lot to encourage test automation in the last decade. So as a general rule, consider automation as an elevation strategy.

A wonderful side-effect of automation is that it is also reduces variability. Repeatable tasks and activities are repeated with digital accuracy. So automation also reduces variability in the process and may help to improve exploitation of capacity at another bottleneck.

The next way to insure the maximum exploitation of our capacity constrained user experience designer is to insure that she can always be making progress on current work. If the user experience designer reports that she is blocked for some external reason then the project manager and if necessary the whole team should swarm on the issue to get it resolved. A strong organizational capability at issue identification, escalation and resolution is essential for effective exploitation of capacity constrained bottlenecks.

If there are several issues blocking current work then the issues impeding the capacity constrained resource, in this case our user experience designer, should get highest priority.

The transparency in the kanban system will help to raise awareness of both the location of the capacity constrained (bottleneck) resources and the impact of any issues impeding flow at that point in the system. With everyone on the project aware of the system level impact of an impediment on the bottleneck, the team will gladly swarm on a problem to resolve it. Senior management and external stakeholders with a vested interest in a release arriving on-time will also give of their time more freely when they understand the value of that time and the impact that swift resolution of an issue will provide.

Hence, developing an organizational capability of transparently tracking and reporting projects using a kanban system is critical to improving performance. Transparency leads to visibility of both bottlenecks and impediments and consequently to improved exploitation of available capacity to do valuable work, through a team focus on maintaining flow.

One more technique which is commonly used to insure the maximum exploitation of a capacity constrained resource is to insure that the resource is never idle. It would be a terrible waste if the capacity constrained resource was left without work to do because of an unexpected problem upstream. For example, a requirements analyst takes several weeks off work due to a family medical issue. Suddenly the constraint is moved. Or perhaps a large section of the requirements are recalled by the business which has made a strategic change. While the team waits for new requirements to be developed the user experience designer is idle. Perhaps the upstream activities are highly variable in nature? This is common with requirements solicitation and development. Hence, the arrival rate of work to be done may be irregular. There could be many reasons that the capacity constrained resource may become idle due to a temporary lack of work. The most common way to avoid such idle time is to protect the bottleneck resource with a buffer of work. The buffer is intended to absorb the variability in the arrival rate of new work queuing for user experience design. Buffering adds total WIP to our system. From a Lean perspective, adding a buffer of work is adding waste and it will increase lead times. However, the throughput advantage provided by insuring steady flow of work through our capacity constrained resource is usually a better trade. You will get more work done despite the slightly longer lead time and the slightly greater total work-in-progress. This principle is explained fully in Chapter 21.

Using buffers to insure a bottleneck resource is protected from idle time is often referred to as protecting the bottleneck or a protection action. Before considering elevating a bottleneck you should seek to maximize exploitation and protection to insure that as much of the available capacity is utilized.

In our traffic management example with SR-520 where the actual throughput was less than 20% of potential turns out to be quite common with knowledge work problems such as requirements analysis and software development. It is often possible to see improvements of up to 4 times in delivery rate by exploiting a bottleneck.

In the example from Microsoft in Chapter 3, a 2.5 times improvement was achieved through better exploitation and protection to remove variability from the system.

Subordination Actions

Once you've decided how to exploit and protect a capacity constrained resource, you may need to take actions, to subordinate other things in the system, to make your exploitation scheme work effectively.

Let us revisit our fantasy traffic system, in a future world. Here we decide not to build a new floating bridge across Lake Washington, instead we decide to fit all vehicles traveling on SR-520 at peak times with a new velocity management system that uses radar and wireless communications to regulate the speed of traffic on a 7 mile stretch of freeway. This new system would act like the cruise control and override the manual use of the accelerator and brake pedals. Citizens would be incentivized to fit the system to their vehicles with tax breaks. Once enough cars have the system it would be switched on and cars without it would have to find an alternative route or choose to cross outside of peak times. The result would be smoother flowing traffic and greater exploitation of capacity at the bottleneck. My guess is that such a system if it could be effective would reclaim about 50% of the lost capacity. Or put another way, it would increase throughput across the SR-520 in peak times by about 2.5 times.

So what have we done with this example? We have subordinated the driver's right to affect and control their own speed in pursuit of the greater common goal of faster journey times facilitated by greater throughput across the bridge. This is the essence of a subordination action. Something else will need to change in order to improve the exploitation of the bottleneck.

For those knowledgeable about The Theory of Constraints it is often counter-intuitive to realize that the changes required to improve performance in a bottleneck are usually not made at the bottleneck. While reviewing the manuscript for my first book[2] a now well known member of the Agile Software Development community suggested that using the Theory of Constraints as an approach to improvement would lead to everyone on the team wanting to be part of the bottleneck resource because they would get all the management attention. This is an easy mistake to make. Counter-intuitively most bottleneck management happens away from the bottleneck. Many of the changes focus on reducing failure load to the bottleneck in order to maximize its throughput. As a general rule, expect to maximize exploitation of bottleneck capacity and hence maximize throughput and as a result, minimize the delivery time on your project, by taking actions all over the value stream and most likely not at the bottleneck itself.

Non-instant Availability Resources

Non-instant availability resources are not strictly speaking bottlenecks. However, by and large they look and feel like bottlenecks and the actions we might take to compensate for them are similar in nature to those for a bottleneck. Anyone who has ever driven a car and stopped at a traffic light understands the concept of non-instant availability. While stopped at a red light the car cannot flow down the road. The lack of flow is not caused by capacity constraints on the road but by a policy that allows cars traveling on another road, the right to cross that road.

A better example, and sticking to our theme in this chapter of transport in Washington State, would be the ferry system that operates across the Puget Sound linking the Kitsap and Olympic Peninsulas with the mainland around the city of Seattle. There are three ferry crossing, two that leave from Seattle crossing to Bremerton and Bainbridge and my favorite the SR-104 crossing between Edmonds on the east-side to Kingston on the west. When reading a map, the ferry route is actually shown as part of the SR-104 road. It is often marked as "toll" rather than explicitly saying "you have to get on a boat here ;-)." The transportation people think of the ferry as a non-instant availability road.

When you show up for the ferry, you pay some money and are asked to wait in a holding area. A typical wait time is about 30 minutes as the ferry takes 30 minutes or so to cross the Puget Sound and there is a 10-15 minute period to unload and another to load all the vehicles on before setting sale. Usually the ferry company is

[2] Anderson, David J., Agile Management for Software Engineering – Applying the Theory of Constraints for Business Results, Prentice Hall PTR, Saddle River New Jersey 2003

operating two boats, so boats sail every 50 minutes or so. At peak times they may operate 3 boats on the route shrinking the header between sailings to around 35 minutes.

Most of the time, the ferry will sail close to full but the system is not capacity constrained. The fact that cars build up in a holding area, a buffer, and are loaded onto the ferry for sailing, batch transferred, does not indicate a capacity constrained resource. It does however indicate a non-instant availability resource. Ferries only sail once or twice an hour with a capacity of around 220 cars per sailing.

At peak times, the ferry system does become capacity constrained. When this happens, the arrival rate of cars wishing to cross exceeds the capacity to transport them. The capacity is roughly 300 cars per hour. Cars begin to back up, queuing outside the holding area and before the toll booth. Often a tail back of vehicles for 2 miles through Edmonds or Kingston can be observed during these peak demand times. There is little that can be done. Cars just have to wait. It isn’t easy to elevate the constraint by bringing on another ferry. The timetable and schedule of ferry sailings is designed to provide a reasonable level of service a reasonable amount of the time. To always have excess capacity would be excessively expensive on the state taxpayers who subsidize the ferry service.

Moving back to software development and knowledge work, non-instant availability tends to be a problem with shared resources or people who are asked to perform a lot of multi-tasking. As we all know, there really is no such thing as multi-tasking in the office, what we do is frequent task switching. If we are asked to work on three things simultaneously, we work on the first thing for a while, then switch to the second, then to the third. If someone is waiting on us finishing the first thing while we are working on the second or third then we would appear to be non-instantly available from that persons and the first tasks perspective.

One example of non-instantly availability that I observed occurred with a build engineer. The company had a policy that only configuration management team personnel were allowed to build code and push it into the test environment. This policy was a specific risk management strategy based on historic experience that developers were often careless and would build code that would break the test environment. The test environment was often being shared between several projects and hence the impact of a bad build would be significant. The technology department did not have a good program level coordination capability and the chances that one team and one project was working in an area of the aggregate IT systems that might affect that of another project was quite high. The coordination function of knowing what was happening at a technical code level between and across projects was given to the configuration management department. These professionals were known as build engineers. The build engineer was responsible for knowing the impact of a set of changes in a given software build and avoiding breaking the test environment so that the flow of all projects was not affected by an outage in the test environment.

Generally, a project had a build engineer assigned from the shared pool of configuration management team resources. However, the demand from a single project for code builds into test was not sufficient to keep a build engineer busy for a full day. In fact, it generally wasn't enough to keep him or her busy for more than an hour or two each day. Hence, build engineers were asked to multi-task. They were either assigned to several projects or were assigned other duties.

In the example of Doug Burros at Corbis, he was assigned as the build engineering for the sustaining engineering activity. He was also assigned two other duties. He had responsibilities for building out new environments and also for maintaining existing environments. He was the configuration management engineer with full responsibility to keep the configurations current. This included applying operating system and database server patches and upgrades, middle-ware patches and upgrades, system configuration and network topology and so forth. He allocated about 1 hour per day to perform the build engineering duty. Typically, this would be in the morning approximately 10am to 11am. If developers found themselves at 3pm in the afternoon requiring a test build they would typically have to wait until the next day. The build engineer was non-instantly available. Work would block and as the sustaining engineering was operated using a kanban system, work would quickly backup along the whole value-stream causing idle time for many other team members.

The actions taken in response to non-instant availability problems with flow are remarkably similar to those for a capacity constrained resource.

Exploitation/Protection Actions

The first thing was to recognize that Doug was a non-instant availability resource and to observe the impact that this was having. Work was backing up when he wasn't available because the kanban limits were tightly defined. As Doug was a source of variability in flow, the correct course of action was to place a buffer of work in front of Doug. The trick was to make this buffer big enough to allow flow to continue without making it too large that Doug would become a capacity constrained resource. I had a discussion with him about the nature of the build activity. It turned out that he could reasonably build up to 7 items in his one hour per day availability. So we created a buffer with a kanban limit of 7. We introduced this to the value-stream and the card wall by introducing a new column called "build ready." We had actually increased the potential total WIP in the system by around 20% but it worked. While builds were not instantly available, the upstream activities were able to keep flowing during the day. The result was a significant boost in throughput and shorter cycle times, despite the increase in WIP. See Chapter 21 for more background on why decisions to maximize flow should trump decision to minimize waste (in this case WIP.)

Subordination Actions

As we learned earlier, subordination actions generally involve making policy changes across the value-stream to maximize the exploitation of the bottleneck. What options were available as subordination actions with our non-instantly available build engineer?

The first thing was to examine the policy of asking Doug to perform three different functions. Was it the best choice? I discussed this with his manager. It seemed that on her team, the engineers liked and needed diversity of work to keep it interesting. Also by asking team members, to perform system build out, system maintenance and build engineering, it maintained a generalist set of skills across all team members and kept the resource pool flexible. This provided the manager with many more options and avoided the potential for capacity constrained resource bottlenecks due to a high degree of specialization. Generalization was also appealing to team members from a career and resume perspective. They didn't want to become too narrowly skilled. So asking the team to work in only one area such as build engineering was not desirable.

Another option might have been to abandon the idea of multi-tasking and dedicate Doug to the sustaining engineering team effort. This would have provided him with a lot of idle time. He would be sitting around waiting for work, like a firefighter in the firehouse, sitting around waiting for a call that there is a fire to put out. Keeping Doug on constant standby would certainly cure the flow problems but was it a reasonable choice?

Budgets were tight and adding people to the configuration management team to handle the system build out and maintenance that Doug was doing would be expensive and perhaps impossible. I would need to ask my boss for budget to get another person because I wanted to keep someone idle most of the time. Was this a good risk management tradeoff?

To decide that we need to look at the cost of delay for the sustaining engineering effort and compare the cost of another member of staff in comparison to the cost of other alternatives to maintaining flow. The reality was that very few items in the sustaining engineering queue had a strategically significant cost of delay. So the idea that we'd keep someone idle, waiting for work, in order to optimize flow was not a viable alternative. Clearly, the exploitation action of adding a buffer of work to maintain flow was a cheaper and better alternative.

However, the discussion of what to do about Doug's lack of instant availability did create a debate on the team about the policy of having build engineers perform this work. The option of ending the policy and allowing developers to build code and push it to the test environment was discussed. It was rejected because the organization had no viable alternative method of coordinating the technical risks across projects. One option of providing a dedicated test environment for that project was rejected for cost reasons and wasn't a practical viable alternative in the short or medium term. Everyone continued to see the value in the build engineering function and the configuration management team.

Elevation Actions

However, buffering and adding WIP to solve the problem felt like a bandaid. It felt like a workaround. And you could view it that way. It was a tactical fix - an effective tactical fix, but nevertheless a tactical fix. Because the kanban system had exposed the non-instant availability bottleneck, and allowed the team to have a full debate around its cause and the possible fixes, the discussion inevitably came around to whether having a human build the code was the right answer. Would it be possible to automate this build process? The answer was "yes" though the investment was heavy. Some considerable development of capability in configuration management and cross project coordination would be needed. In addition, some specialists in automation had to be hired for a period of time to create the system to make it work.

It took around 6 months in total elapsed time, and two contractors for 8 weeks. The total cost in money terms was around $60,000. However, the end result was that Doug was no longer required and builds were instantly available when developers needed them. At this point, it was possible to eliminate the buffer and reduce the system WIP. That in turn resulted in a slight reduction in cycle time.

Automation was ultimately the route to elevation of the non-instant availability bottleneck. Adding capacity, that is, hiring another engineer was not a good choice.

One other path involving automation was also pursued – virtualization of environments. While virtualization is already commonplace, at Corbis, the test environments were still physical. Virtualization was not an organizational capability. By taking time to make it so, the test environment could be easily configured and restored. This reduced the impact of a build breaking the environment – a mitigation strategy. And it also enabled dedicated environments reducing or eliminating the risk of a build breaking another project's configuration.

So buffering was used as a short-term tactical exploitation strategy while automation was pursued as a long term elevation strategy.

And what about our Edmonds to Kingston ferry example? How might that be elevated? Well the State of Washington government is currently considering two options. One would be to replace the current aging fleet of ferries with a newer set of larger, more efficient boats. However, Washington has a lot of experience with floating bridges. There are two across Lake Washington including the SR-520 which is apparently the World's longest such bridge, and another across the Hood Canal on SR-104. What is now under consideration is the possibility of building a new record breaking floating bridge across the Puget Sound as part of SR-104 and replacing the ferry service entirely. The planned bridge would not only solve the problem of capacity constraints at peak times, it would also solve the non-instant availability issues that hamper all ferry services as a option for traffic flow. Such a bridge would open up the Kitsap and Olympic Peninsulas for faster economic growth. Perhaps 50 years from now someone else will be writing a book discussing how the SR-104 floating bridge across the Puget Sound is now a bottleneck and capacity constrained resource during peak commuting hours?

Takeaways

- Bottlenecks constrain and limit flow of work
- Bottlenecks come in two varieties: capacity constrained – unable to do more work; and non-instant availability – limited capacity due to limited (but usually predictable) availability.
- We manage bottlenecks using The 5 Focusing Steps from the Theory of Constraints
- Increasing the capacity at a bottleneck is known as elevation.
- The actions taken to elevate a capacity constrained resource will typically be different from the actions taken to elevate a non-instant availability resource
- Elevation may involve adding resources, or automation, or policy changes that make a previously non-instantly available resource, instantly available.
- Elevation actions typically cost money and take time to implement. Elevation actions are often considered strategic investments in process improvement
- Often bottlenecks in the process are performing well below their potential capacity – below the theoretical capacity constraint.
- Throughput at a bottleneck can be improved up to the limit of the theoretical capacity constraint through the use of exploitation and protection actions
- Typically protection involves adding a buffer of WIP in front of the bottleneck. This is true for capacity constrained resources or non-instant availability resources.
- Exploitation actions typically involve policy changes that control the work done by the bottleneck resource.
- Classes of service can be used as exploitation actions
- Subordination actions are actions taken elsewhere in the value stream to enable the desired exploitation or protection actions. Subordination actions are typically policy changes.
- Exploitation, protection and subordination actions are often easy and cheap to implement as they primarily involve policy changes. Hence, maximizing the throughput of a bottleneck by fully exploiting it can be viewed as tactical process improvement.
- Despite the tactical nature of exploitation of a bottleneck, the gains can often be dramatic. 2.5x to 5x improvements in throughput with consequent drops in cycle time can often be achieved at little to no cost over a short period of months
- Exploitation should always be pursued first before Elevation is attempted
- It is not unusual for a tactical set of exploitation and subordination actions to be implemented while a plan for a strategic change to elevate a constraint is implemented over a longer period of time.

Chapter 18 An Economic Model for Lean

"Waste" (or "Muda" in Japanese) is the metaphor used in Lean (and the Toyota Production System) for activities that do not add value to the end product. The metaphor of waste has been proving problematic with knowledge workers. Often tasks or activities that are costs or overheads but necessary or essential to completing the value-added work are hard to accept as waste, for example, daily stand-up meetings are essential to coordinating most teams. However, these meetings do not directly add value to the end product, so technically, they are "waste" but this has been hard to accept for many agile development practitioners. Rather than have people wrapped up in arguments about what is or is not waste, I concluded it would be better to find an alternative paradigm and alternative language that is less confusing or emotionally evocative.

Redefining "Waste"

Following the lead of writers such as Donald G. Reinertsen, I have adopted the use of the language of economics and refer to these "wasteful" activities as costs. I classify costs into 3 main abstract categories: Transaction Costs; Coordination costs; and Failure Load. Figure xx show this pictorially.

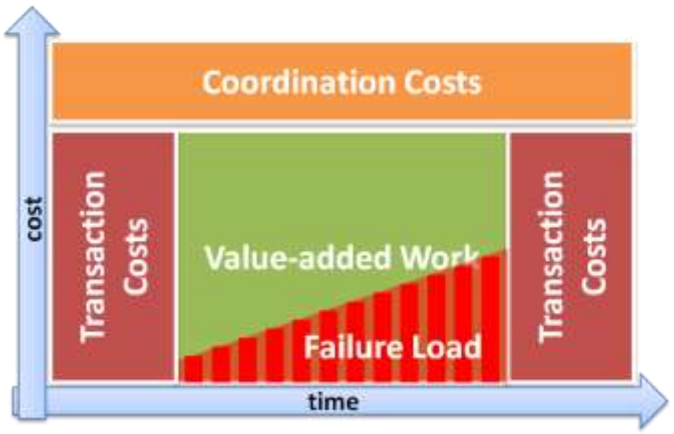


Figure xx. The Economic Cost Model for Lean Software Development

In addition to three sources of cost, the value-added work may depreciate in terms of its expected future value. This is partly because future value is uncertain and as time goes by, more information becomes available and the future value can be predicted more accurately. More problematic is when changes in the market mean that the future value is reduced, for example, a competitor launching the same feature at a lower than anticipated price. This loss in future value represents an opportunity cost. This concept of work-in-progress inventory depreciation is shown in figure xy.

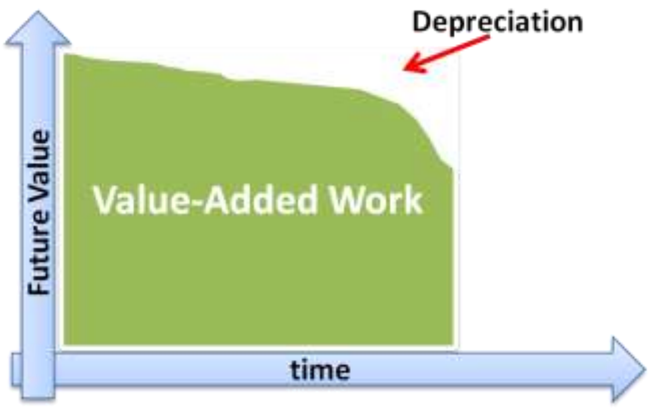


Figure xy. The Economic Future Value Model for Lean Software Development

Figure xx shows that over time there are a number of value-added activities (the green area) for an iteration or project. Surrounding those activities are transaction (red area) and coordination (orange area) costs. The capacity for value-added activities can be displaced by work that can be considered failure load (the red barred area over the green area), that is, work that is either rework or demand placed on the system because of a previously poor implementation. The value-added activities also have a time related value as shown in figure xy. This can be calculated from the opportunity cost of delay curve and from the likelihood of change that would either modify or obviate the need for the particular value-added function. The essence is that valuable customer requirements are both perishable and time critical. Hence, delivering them earlier is nearly always better. Delay reduces value and can be considered waste in the same sense that a fruit-seller who has to dispose of a quantity of over-ripe bananas incurs a cost of the wasted inventory of bananas. Hence, I call this fourth (opportunity) cost, inventory depreciation. It specifically refers to the time related value and likely depreciation of the value of a requirement.

Each of these 4 costs is discussed in detail in the following sections. I will describe them using a simple real world example – the activity of painting the fence at my home in Seattle with preservative wood stain.

Transaction Costs

The fence has 21 sections. The customer value is delivered when a section of fence is painted with the wood stain. Full value is delivered when all 21 sections are stained on both sides.

Before I can start the job, I must first procure all the materials. This involved a trip to Home Depot. There was also some preparation work required on the fence: some repairs; some sanding; and the trimming of plants and bushes to allow access for painting. None of these activities could be described as adding value. The customer does not care that that I have to make a trip to Home Depot. The customer does not care that this activity takes time. In fact, it is annoying as it delays the start and end of the project. These activities delay the delivery of customer value.

So the project has some setup activities that are essential before the value-added work can start.

There may be others. There might be some planning. There may also be some estimation activity and some setting of expectations. The customer may be quoted a price for the job and a delivery date. [In this case, the customer for the project was my wife.]

When it comes to the actual painting of the wood stain, it turns out that 42 sections of fence is too many to attempt in a single session of painting. The velocity of painting was approximately 4 sections per hour. So the job was split up into 6 sections. If this were software development we might have called these iterations or sprints. If it were manufacturing we might call them batches. When I went to start a single section of painting, I also had some setup activities. The first was to change clothes. Then I would setup up the materials. I would move all the paint, brushes and other tools from the garage to the location for the painting that day. Only then could I start painting.

So project and iterations both have setup activities.

After a couple of hours of painting, I might want to take a break. Perhaps it is time for lunch. I can't just drop everything and start eating. First I must secure the paint by replacing the lid on the tin and then I must secure the brushes by either cleaning them or dropping them into a jar of water to prevent them from hardening while I take a break. Next I must personally clean up. I wash my hands and I change out of the work overalls. Only then can I go and eat.

When the whole project is complete, I might have some extra wood stain remaining and any full tins can be returned to Home Depot for a refund. So another trip is required.

It seems that both iterations and projects have a set of cleanup activities.

In economic terms these setup and cleanup activities are referred to as transaction costs. Every value-added activity has associated transaction costs. These transaction cost activities are things which the customer may not see, most likely does not value and to which they are ambivalent at best. The customer may be forced to pay the costs of these activities but they would prefer not to. How often have you called a plumber to fix a washing machine or dishwasher and been asked for a $90 call out fee? This is a transaction cost. Would you prefer a lower fee? Would you choose a plumber who did not charge such a fee? The transaction costs do not add value. They may be necessary but in Lean terms, they are waste.

So the first two types of waste are transaction costs, specifically: the setup or front end; and the clean up or backend transaction costs.

If you consider this for software development activities, you will realize that all projects have a number of setup activities, such as project planning, resource planning and recruitment, budgeting, estimating, risk planning, communication planning, facilities acquisition and so on. Most projects also have clean up costs and other related back end transaction costs, such as delivery to the customer, tear down of environments, retrospectives, reviews, audits, user training and so on.

Iterations too have transaction costs including the iteration planning and backlog selection (or requirements scoping), perhaps estimation, budgeting, resourcing, environment setup. On the backend they will have transaction costs including integration, delivery, retrospective, and environment tear down.

Coordination Costs

Coordination is necessary as soon as two or more people try to achieve a common goal together. We invented language and communications systems in order to coordinate between human beings. When we agree to meet with friends, have drinks, dinner and watch a movie on a Friday evening, all the emails, text messages and phone calls that are required in order to make that social evening happen, are the coordination costs of spending some valuable quality time with friends and seeing a movie.

So coordination costs on projects are any activities that involve communicating and scheduling. When people on project teams complain that they can’t do value-added work such as analysis, development or testing, because they are doing email, they are performing a set of coordination activities – each email read and answered is a coordination activity. When they complain that they can’t do value-added work such as analysis, programming or testing, because they are always in meetings, then these meetings are coordination activities.

In general, any form of meeting is a coordination activity including favorites of the Agile community such as daily standup meetings, unless they meeting is designed to produce a customer valued deliverable. If 3 developers get together at a white board and model a design for code they are about to implement, this is not a coordination activity, it is a value-added activity. Why? Because it produced information that builds towards a complete customer-valued function.

If we view software and systems development as an information arrival process, where our starting point is no information, and complete information represents working functionality that meets the customer's needs and intent, then any information that arrives between the starting point and end point, that moves us closer to that end point, of working functionality that meets the customer's needs and intent, is value-adding information.

If team members meet in order to create value-adding information, a design, a test, a piece of analysis, a section of code, then that meeting is not a coordination cost, it is a value-added activity.

However, if team members meet in order to discuss, status, or task assignment, or scheduling that helps coordinate team members, their actions and the flow of deliverables, then that meeting is a coordination cost and should be regarded as wasteful. As such, you should seek to reduce or eliminate coordination meetings.

Hence, a 5 minute standup is better than a 15 minute standup if it achieves the same amount of coordination. A 15 minute standup is better than a 30 minute standup, if achieved the same amount of coordination.

You can think about reducing coordination activities by finding other better ways of coordinating people.

One way is to empower team members to self-organize. Command and control type management where people meet in order to assign tasks to individuals in advance are wasteful. It is better to let team members self-assign tasks. It generally reduces the coordination costs on a project. However, self-organization requires information in order to work. Techniques within Kanban such as the use of visual tracking of the value-stream and visualization of work using a card wall and electronic tools and reports provide coordination information that enables self-organization and reduces coordination costs on a project. Use of classes of service and visualizing them with colors of cards or swim-lanes on the card wall, together with the associated set of policies for the class of service, enable self-organization of scheduling and automatic prioritization or self-expediting (a term I first saw used by Eli Goldratt referring to a buffer management system).

In general the more information that can be made transparent to the knowledge workers on the team, the more self-organization will be possible and the fewer coordination activities will be required. Let transparency of work, process flow, and policies related to risk management, displace coordination activities. Reduce waste through wider use of transparency.

How do you know if an Activity is a Cost?

I have discovered that many people have trouble identifying wasteful activities. I have seen for example, Agile advocates argue that daily standup meetings are value-added. I do not subscribe to this view. I cannot fathom that a customer cares less whether a team holds standup meetings or not. What the customer wants is functionality that enables their goals, delivered in a timely manner with high quality. Whether or not a team needs to hold daily standup meetings to enable such delivery is neither here nor there from a customer perspective.

So how do you identify wasteful transaction costs or coordination activities?

I believe that you ask yourself, "if this activity is truly value-adding, would we do more of it?"

When you ask a Scrum advocate who is vehemently arguing that daily standup meetings are value-added, whether they would hold the standup twice daily, or whether they would lengthen it from 15 minutes to 30 minutes, then they will surely reply,
"No!"
"Well, if standup meetings are truly value-added," I reply, "then surely doing more of them would be a good thing?"

This is really the acid test that demonstrates the difference between a truly value-added activity and a transaction or coordination cost. Developing more customer requirements is clearly value-adding. You would do more of it if you could and the customer would gladly pay for it. Planning is clearly not value-added. The customer would not pay for more planning if he could avoid doing so.

So ask yourself, would we do more of this? Challenge others with the same question about activities they undertake. If the answer is "No" then consider how you might go about minimizing the time and energy spent on the activity or how you might make the activity more effective, and hence, reduce the duration, frequency or quantity of the activity.

It can sometimes be difficult to determine whether an activity is a transaction cost or a coordination cost. Some activities often look like both. I see this confusion when I teach Kanban classes all the time. As I do with class participants, I would urge you not to waste too much energy trying to determine the difference. What is important is that you have identified an activity as non-value-adding and therefore wasteful and you know that you want to reduce or eliminate that activity as part of a program of continuous improvement.

Failure Load

Failure load is demand generated by the customer that might have been avoided through higher quality delivered earlier. For example, a lot of help desk calls generate costs for a business. If the software or technology product or service were of higher quality, more usable, more intuitive, more fit for purpose, then there would be fewer calls. This would enable the business to reduce the number of call center personnel and reduce costs.

Lots of calls to a help desk tend to generate lots of production defect tickets. When selecting the functions in scope for a project or iteration, the business must choose between new ideas and production defects. Production defects aren't just software bugs, they include usability problems and other non-functional issues such as poor performance, lack of responsiveness under load or certain network conditions and so on. The fix for a production defect against a non-functional requirement may appear like new functionality – a design for a new screen perhaps – but it truly isn't. It is failure load. That new screen design came about because of a usability defect in a previous release.

Failure load doesn't create new value it enables value left on the table from a previous release. More than likely the earlier release of the product or service failed to deliver on its projected payoff function. While some of this may be due to market variability or unpredictability, some of the shortfall will have been discovered to come from problems with the earlier release. Perhaps a bug in the product prevents the usage of some functionality. Because of this, potential customers are switched off the product and defer purchase or switch to a rival product.

So the picture is muddy. Failure load still adds value. But what is important is that it adds value that should have been there already. Reducing failure load reduces cost of delay. Reduced costs mean more profits sooner. Reduced failure load means more of the available capacity can be spent on new functionality. Reduced failure load enables a business to pursue more market niches, with more product offerings. Reduced failure load enables more options. Reduced failure load may enable a reduction in team size and a reduction in direct costs.

Inventory Depreciation

There is a time value to ideas. We capture the time value of ideas for product, service or systems functionality with the opportunity cost of delay curve. That graph is subject to uncertainty and the spread of uncertainty widens the further out in time we go from the present. As time goes by, new information arrives that will cause us to redraw the cost of delay curve. For example, a competitor might launch a rival product that will reduce anticipated demand for our new product. Perhaps, a government will change regulatory requirements and obviate the need for a requirement in our system. If this is the case, then we would prefer that we hadn't started work on that requirement yet. If we have started work then all of that code will need to be removed.

Because of the time value of ideas, we must treat the backlog of requirements, and all work-in-progress as a depreciating asset. Some amount of our requirements will be obviated and some amount of them will turn out to be worth much less than we believe them to be worth when we start on the project.

To put hard numbers of the cost inventory waste would require significant amounts of historical data. We may not have that data available. However, it should not stop us from doing the right thing. If inventory, work-in-progress, is depreciating then it would make sense to have less of it. Less customer valued work that has been started and has incurred costs and energy means less waste. While we may not be able to quantify the amount, the principle is sound. Reducing work-in-progress results in less waste and makes our process more efficient.

Takeaways

- Waste can be classified into 4 categories: transaction costs; coordination costs; failure load; and inventory depreciation.
- The concept of waste is a metaphor
- The waste metaphor does not work well for everyone as waste is often necessary though not specifically value adding as a result I have replaced it with an economic cost model
- To determine if an activity is truly wasteful ask, "Would we do more of this if we could?" If the answer is no, then the activity is some form of waste
- Transaction costs come in two types: setup activities; and cleanup or delivery activities
- Coordination costs are activities that are performed in order to assign people to tasks, schedule events, or coordinate the work of two or more people towards a common outcome
- Failure load is new value-added work that is generated because of some earlier failing such as a defect in the software, or a poor design or implementation that led to lack of customer adoption, a failure to realize a target payoff function, or a significant rise in help desk calls or service requests.
- Failure load uses capacity that could have been used for new, innovative, additional customer valued, and revenue generating features.
- All ideas on the backlog and customer valued work-in-progress is inventory.
- An inventory of ideas has a shelf life
- The value of the ideas depreciates due to changes in the market
- Reducing inventory and with it cycle time reduces and depreciation in the potential value of the ideas captured in the backlog
- Turning ideas into working, customer-deliverable code quicker maximizes its potential value and minimizes waste

Chapter 19 Sources of Variability

Variability in industrial processes has been studied since the early 1920s. The pioneer of this subject was Walter Shewhart. His techniques became the foundation for the quality assurance movement and are foundational elements of both the Toyota Production System and Six Sigma methods for quality and continuous improvement. Shewhart's techniques were adopted, developed and evolved by W. Edwards Deming and Joseph Juran. Their work was inspirational for Watts Humphreys and the founding members of the Software Engineering Institute at Carnegie Mellon University who held the belief that the study of variation and its systematic reduction would bring great benefits to the software engineering profession.

There is a great deal of material published by Shewhart, Deming and Juran on the study of variation and its use as a management technique and the foundation of program of improvements. In addition, there is much published on the quantitative assessment method known as Statistical Process Control (SPC) that emerged as the main tool for studying variation and acting upon it. Use of SPC is being observed as emerging with teams adopting (at the time of writing), however, use of SPC is considered an advanced and high maturity topic that will be addressed in a later book. Here, we will talk about variation in the most general of terms and simplest form of understanding.

Shewhart classified variability and variations in process performance into two categories: internal; and external.

Internal sources of variation are variations which are under the control of the policies that govern the process and can be directly affected by individuals, the team and the management making policy changes that affect the process in use. A change to the process represents a change that affects the internal sources of variation. Somewhat ironically, Shewhart, named these internally generated variations "chance cause variations." "Chance" implies that the variation is random and the randomness is a direct consequence of the process design. It does not imply that the randomness is evenly distributed or follows a standard distribution. Changes to the process design via changes in internal policies will affect the mean, the spread and the shape of the distribution, of any variation.

To use a general example, a batter in the game of baseball will have a hit ratio indicating how often he managed to hit a pitch that resulted in achieving first base or better. Different batters will exhibit different ratios with a typical range of about 0.100 to 0.350. On any given day, an individual batter may not achieve his typical hit ratio. This is determined by a number of factors such as pitcher selection, how well other players hit the ball, and the specific pitches thrown by the pitcher.

If we changed the rules of baseball to say allow 4 strikes before a batter is out then we change the odds in favor or the batter against the pitcher. The batter average will increase as a result. Some better player may well achieve hit ratios in excess of 0.500 as a result of such a rule change. This is an example of modifying the system to modify the chance cause variation within the system.

If we want to interpret this for a software development specific example, an internal, chance cause variation, would be the number of bugs created per line of code, per requirement, per task, or per unit of time. The mean number, the spread and the distribution, of the bug (or defect) rate can be affected by changing the tools and process such as insisting on unit tests, continuous integration and peer code reviews.

The process definition in use on your team, expressed as policies, represent the rules of the collaborative game of software development. The rules of the game determine the sources of and quantity of internal variation. The irony is found in the notion that the "chance cause" variations are actually directly under the control of the team and management through their ability to modify policies, change the process and affect sources of internal variation.

External sources of variation are things that happen that are out with the control of the immediate team or management. They are randomizations that come from other teams, suppliers, customers, and random "acts of God," as they are known in the insurance industry. For example, a 2 week outage of a server farm is caused by excessive flooding that resulted from unusually wet, stormy weather. External sources of variation require a different approach to management. They cannot be directly affected by policies but a process can be put in place to effectively deal with external variations. The body of knowledge that relates directly to this field is the issue and risk management body of knowledge.

Shewhart named external variations, "assignable cause variations." By "assignable" he implied that someone (or a group of people) could easily point at the source of the problem and consistently describe it. Such as "There was a storm. It rained really hard and our server farm was flooded." Assignable cause variations cannot be controlled by the local team or management but they can be predicted and plans made and processes designed to cope with them gracefully.

Internal Sources of Variability

The software development and project management process in place coupled to the organizational maturity and capability of the individuals on the team determines the number of internal sources of variability and the degree of that variability.

To avoid confusion Kanban must not be thought of as a software development lifecycle process or a project management process. Kanban is a change management technique that requires making alterations to an existing process: changes such as adding work-in-progress limits to the existing process.

Work Item Size

The method of analysis used to breakdown requirements and itemize them for development will have its own degree of variability. One dimension to this will be the size of work items. Early literature describing the Extreme Programming method described user stories as a narrative description of a feature as implemented and used by an end user, written on an index card. The only constraint was the size of the card. The effort required to create a user story was described as anything from a half day to 5 weeks of work. Within a couple of a template for writing user stories had emerged from the community in London.

As a <user>, I want a <feature>, in order to <deliver some value>

The use of this template greatly standardized the writing of user stories. One of the creators of this approach, Tim McKinnon, reported to me in 2008 that he now had data to show that the average user story was 1.2 days of effort and the spread of variation was half a day to about 4 days.

This is a specific example of reducing the chance cause variation in the extreme programming method by asking the team to standardize user story writing around a given template. By doing so, Tim changed the rules of the game. The original rules asked team members to write stories on index cards in narrative, and the new rules asked them to continue with index cards but to follow a specific sentence format. These changes are quite clearly under the influence and control of local managers. They are internal to the system. The size of a user story is controlled by chance cause variation.

Work Item Type Mix

When all work is treated the same and perhaps called by a single type there is likely to be greater variation in size, effort, risk, or other factors. By breaking work out by specific type, it is possible to treat different types differently and to provide greater predictability.

For example, the Extreme Programming community developed type definitions for different sizes of stories. These gained names like “epic” and “grain of sand.” An epic might be a larger story that would take several people several weeks to develop, while a grain of sand might be a small story that could be completed by a single developer or pair of developers in a few hours. By adopting this nomenclature of Epic, Story and Grain of Sand, we now have three types. For each individual type the spread of variation will be lower than the spread when all work was treated as a story.

Work in a typical software development department typically involves several types. There might be customer valued new work with a name such as ”feature,” “story,” or “use case.” As just described, these might be stratified into size elements, or by some domain subtype, or risk profile. There might be defect removal work such as “production bug,” “(internal) bug.” There might be maintenance work described as “refactoring,” “re-architecting” or simply “upgrading.” Software operating systems, databases, platforms, languages, APIs, and service architectures change over time and the code base needs to be updated to address the changes.

By using techniques to identify different work item types we can change the mean and spread of variability and improve the predictability in the system for any one type of work.

An additional strategy to improve predictability is to specific allocate total WIP capacity to specific types. For example, with my maintenance team at Corbis, only two IT Maintenance items were permitted at any given time. This limited the capacity spent on API and database upgrades. This strategy is particularly useful when the types are divided out by size or effort required such as epic, story and grain of sand. By allocating specific capacity to each type, the responsiveness of the system is maintained and the predictability will be greater.

Consider a team with a Kanban board where there is a limit of two epics, eight regular stories and four grains of sand. Two epics are in progress. A slot opens up in the queue for a regular story but there are none in the immediate backlog ready to start. The team has a choice of starting an epic, or a grain of sand, or sticking to the type allocation and incurring some idle time.

If they start an epic, and a few days later a regular story shows up in the backlog, then they would be unable to start the regular story for quite a while. This will increase the cycle time spread for regular stories.

Starting a smaller grain of sand is a better choice as it might be finished before another regular story is ready to start. In this case, there is no impact but there is a benefit in additional throughput. However, if they don’t get lucky and they fail to complete the smaller item before a story is ready to start, then the cycle time spread for regular stories will be affected adversely, though not as badly in the epic scenario.

Predictability and risk management should typically trump an opportunity to increase throughput as business owners and senior managers value predictability more than throughput. Predictability builds and holds trust, a core agile value, better than delivering more with less reliability.

Class of Service Mix

If we consider the classes of service described in chapter 11, we can anticipate how variability might be affected by the mix of items. If an organization suffers from a lot of expedite requests, these will randomize everything else, increasing the cycle time mean and spread of variation reducing predictability of the whole system.

Expedite requests are essentially external variations and are described in the next section.

If the demand for other class of service types is fairly steady then the cycle time performance for each type should be fairly reliable. The mean and spread of variation should be measurable and should remain somewhat constant. This provides predictability. You can achieve this if the backlog is sufficiently large and full with a strong mix of each class. Allocate a WIP limit to each class of service. This will enable the mean and spread of variability for each class to settle down and the system will be predictable.

If the demand is variable, for example, there are only a few fixed delivery date items and they tend to be seasonal, then some action should be taken to either shape or control demand, or changes to the allocation of WIP limits across types should be instituted seasonally to anticipate demand changes, or alternatively changes to pull policies should be made seasonally to cope with demand.

Consider a team with a WIP limit of 20, allocated as 4 fixed date items, 10 standard class items and 6 intangible class items. You can have a policy that these limits must be strictly adhered to, or you can loosen the rule and allow a standard or intangible item to fill a slot for a fixed date item when there is insufficient seasonal demand for fixed date items. These policies can be switched over at different times of year to improve the overall economic outcome and insure that the system remains fairly predictable.

Irregular Flow

Irregular flow of work can be caused by both external and internal sources of variability. Every single item pulled through the Kanban system will be different. Different in nature to some degree and different in size, complexity, risk profile and effort required. The natural randomness of this will cause ebbs in flow. A Kanban system naturally copes with this so long as the WIP limits are enforced. However, greater variability from other sources such as work item size, demand patterns, type mix, class of service mix, and external sources will require suitable buffering to absorb the ebbs and surges in flow through the system. Additional buffers may be required and WIP limits will need to be larger when there is more variability in the system. Greater WIP limits will result in longer cycle times but the smoother flow should reduce variability. So an increase in a WIP limit to smooth flow with increase the mean cycle time while reducing the range of cycle time variability. This is generally a more desirable outcome as managers, owners and usually customers value predictability over the random chance of a shorter cycle time or greater throughput.

Rework

Rework whether it is internal bugs being fixed before release or production defects displacing new customer-valued work affects variability. If a defect rate is known, regularly measured and fairly constant then the system can be designed to cope gracefully with it. It will be an economically inefficient system but it should be reliable. What causes a lack of predictability is when the defect rate is not anticipated correctly. Unplanned for bugs will lengthen cycle times, tend to increase the spread of variation and greatly reduce throughput. It seems to be very hard to plan for a specific defect rate, e.g. 8 bugs per user story, and for those defects to be of a known or predictable size and complexity. The best strategy for reduction of variability due to defects is to relentlessly pursue high quality with very low defect counts.

Changes to the software development lifecycle process can be made to greatly affect defect rates. Use of peer reviews, pair programming, unit tests, automated testing frameworks, continuous (or very frequent) integration, small batch sizes, cleanly defined architectures and well factored, loosely coupled, highly cohesive code design will greatly reduce defects. Changes that directly affect defect rates and indirectly improve the predictability of the system are under the control of the local management and the team directly.

External Sources of Variability

External sources of variability come from places that are not directly controlled by the software development process or project management method. Some of these will be from other parts of the business or the value stream such as suppliers or customers. Other external sources might involve elements of the physical world that can't be easily anticipated, predicted or controlled. For example a piece of equipment failing, or adverse weather conditions.

Requirements Ambiguity

Poorly written requirements, ill-defined business plans, lack of strategic planning, vision or any other context setting information may mean that a team member is unable to complete a piece of work. The ambiguity in the context setting material means that a team member such as a developer or a tester is unable to make a decision. The work item becomes blocked due to this inability to make a decision. New information is required to clarify things so that the team member can make a good quality decision and the work-in-progress will flow towards completion.

In order to reduce the impact of such blockages, the team and direct management need to implement an effective issue management and resolution process as described in Chapter 20.

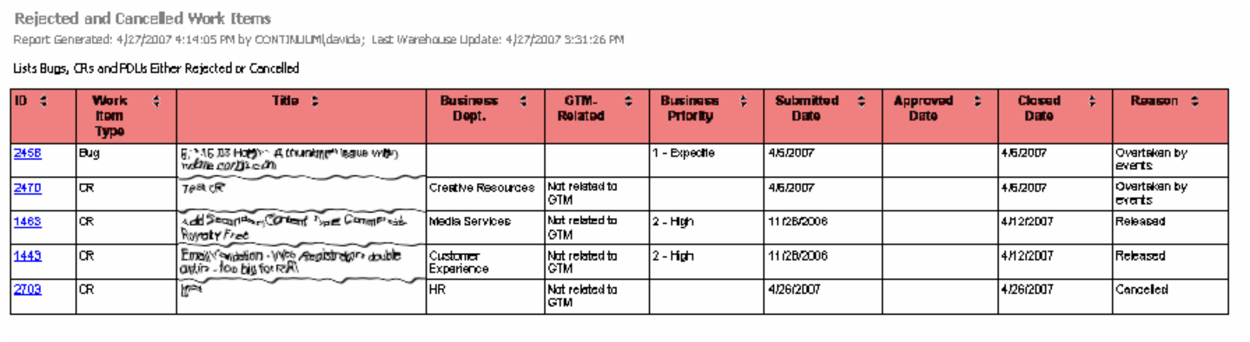
As a team and organization matures it may be possible to discuss root cause analysis and elimination. Blocking issues due to ambiguous requirements can be addressed by directly influencing the analysis processes used to develop requirements and by improving the capability and skill level of those defining them. Measures such as these typically require the collaboration of other departments and managers and a will on the part of the business to improve.

At Corbis in 2007 this was achieved through a gradual process. First the Kanban system was implemented including a visual board, electronic tracking system and the transparency that comes with it. The business became more and more involved and interested in the software development activity and in monitoring the process performance. A report was generated showing the number of open issues, the number of work items blocked and the average time to resolve. [Reference Issues & Blocked Work Items report in Chapter 12.]

When a requirement made it the whole way through to acceptance testing before it was rejected as not what the business really needed, the team reacted by creating a waste bin on the board and placing the ticket in it. Management then asked for a small set of electronic reports that showed work which had entered the system but failed to make it the whole way through.

Picture xy. Board with waste bin

The combination of transparency, reporting and building awareness of the impact and cost of poor requirements resulted in the business voluntarily changing its behavior. The waste report that showed the effect of poor requirements initially showed 5 to 10 items per month. By the fifth month it was empty. The business had come to appreciate that by taking more care they could avoid wasting capacity. They voluntarily collaborated to make the outcome of the system better. The net effect was root cause elimination of the assignable cause variations from poorly written requirements or ill-defined context information.


Rejected and Cancelled Work Items

Report Generated: 4/27/2007 4:14:05 PM by CONTINUUM\davida; Last Warehouse Update: 4/27/2007 3:31:26 PM

Lists Bugs, CRs and PDUs Either Rejected or Cancelled

ID	Work Item Type	Title	Business Dept.	GTM-Related	Business Priority	Submitted Date	Approved Date	Closed Date	Reason
2456	Bug	[illegible]			1 - Expedite	4/5/2007		4/5/2007	Overtaken by events
2470	CR	Test CR	Creative Resources	Not related to GTM		4/5/2007		4/5/2007	Overtaken by events
1463	CR	Add Secondary Content Type Commercial Royalty Free	Media Services	Not related to GTM	2 - High	11/28/2006		4/12/2007	Released
1443	CR	Email validation - Web Registration double entry - too big for R/R	Customer Experience	Not related to GTM	2 - High	11/28/2006		4/12/2007	Released
2703	CR	[illegible]	HR	Not related to GTM		4/26/2007		4/26/2007	Canceled

Picture xz. Rejected & Cancelled Work Report showing abandoned items in last month

While the software development team had taken actions to provide greater transparency and awareness, the actions taken did not directly affect the requirements development process. The issue management and resolution process merely mitigated the impact of blocking issues by raising awareness and the time to resolve. The result was lower impact on the mean cycle time and its spread of variation. The effects of the transparency and reporting that eventually resulted in an external change in process eliminated the root cause of the problem.

This is anecdotal evidence that actions can be taken locally that will have an indirect effect on assignable cause variations

Expedite Requests

Expedite requests happen because of external events such as an unexpected customer order, or due to some breakdown in a company's internal process, for example, lack of communication that results in late discovery of some important requirement. Expedite requests are by definition assignable cause variations as the reason for the request is always known and therefore always "assignable."

Expediting is known in industrial engineering to be bad. It affects predictability of other requests. It increases mean cycle time and the spread of variability and it reduces throughput. Evidence collected at Corbis throughout 2007 demonstrated that this industrial engineering result held true for software development processes. So expediting is undesirable even if it is being done to generate value.

The need for expediting can be reduced. Increasing slack capacity through improvements in throughput, automation or increased resources will improve the ability to respond. Shorter cycle times, greater transparency and improving organizational maturity will reduce the need for expediting. Good Kanban teams have been shown to do very little expediting. In fact at Corbis during 2007 there were only 5 requests in total.

As with poor requirements we can hope that transparency of process and good quality information regarding throughput, cycle time and due date performance, will influence upstream behavior. We hope that demand will be shaped so that it is effectively understood early enough so that it can be handled using a regular class of service rather than as an expedite request.

One method of provoking this change is to agree a limit on the number of expedite requests that will be processed at any given time. At Corbis this limit was one. By denying the business the ability to expedite anything they feel like, you force upstream people such as sales or marketing people to explore opportunities early and assess them effectively. If sales people are paid on commission and measured on revenue generated then failure to expedite something will hurt them. If they failed because the WIP limit for expedite requests had been reached then they will try harder in future to gather enough information to post a request early enough that it is met with a regular class of service. Again this is an example where some internal actions can be taken to indirectly affect an assignable cause of variation.

Irregular Flow

Irregular flow of work can be caused by both chance and assignable cause variations as mentioned above. Assignable cause variations that affect flow all result in blocked work. Problems such as ambiguous requirements, and environment or specialist shared resource availability are all common reasons for assignable cause blocked work.

Blocked work items require a strong discipline and capability at issue management and resolution as described in chapter 20. There are two approaches to dealing with this.

The first will ease flow but at the expense of lead time and possibly quality. You can improve flow by having a larger overall WIP limit achieved through either explicit buffering or by using a policy with less restriction on WIP for example three things per person, rather than 1.2 things per person on average. The greater WIP limit means that while something is blocked the team members can be working on other items. I recommend this approach for immature organizations. The effects should be simple and lacking in drama. Lead times will be longer but this may not be an issue in many domains. The spread of variation might be greater so lead times will be less predictable. However, they may still be more predictable through the use of a Kanban system than before. The biggest negative to using greater WIP limits is that there is no tension to provoke discussion and implementation of improvements. There is no pressure to improve.

The second approach is to relentlessly pursue issue management and resolution and as the team matures to move towards root cause analysis and elimination with specific improvements designed to prevent assignable cause variations happening in future. In this approach you leave the WIP limits, buffer sizes and working policies, fairly tight and you cause the work to stop when things become blocked. Idle time for those blocked raises awareness of the blocking issue. It may cause a swarming behavior to try and fix the issue and it has been seen to encourage those idle team members to think about root causes and possible process changes that will reduce or eliminate the possibility of re-occurrence. Keep the WIP limits tight and pursuing issue management and resolution as a capability has been seen to create a culture of continuous improvement. I first saw this at Corbis in 2007 but there have been several other reports that emerged in 2009 at firms such Software Engineering Professionals in Indianapolis, and IPC Media and BBC Worldwide, both in London. There is now sufficient evidence to suggest that Kanban does provoke a culture that is focused on continuous improvement. The consistent process elements between the examples seem to be a willingness to enforce tight WIP policies, to mark work as blocked, to allow the line to stop, to incur idle time, and to pursue issue management and resolution as an organizational discipline. What results from this is a focus on root cause analysis and elimination and the gradual introduction of improvements that both reduce assignable cause variations and ignite a wider culture of continuous improvement.

Environment Availability

Environment availability is quite a typical assignable cause issue that can have a significant effect on flow, throughput and predictability. Environment outages often cause entire workflows to stall. A kanban system will bring visibility to the problem and its impact. The idle time incurred by enforcing a WIP limit has been seen to encourage collaboration on resolving the outage. When upstream folks such as developers and testers help systems maintenance people to recover an environment this behavior is often referred to as swarming. Swarming implies the concept that the team swarm together to work on a single problem until it is resolved. The nature of Kanban encourages teams to focus on lead time, throughput and flow through the value stream. By aligning all the groups up and down the value stream with the same goals, there is an incentive for this swarming behavior to emerge. Everyone wins when idle people volunteer to collaborate to resolve an issue when it affects them even though it is not in their immediate work area of area of responsibility.

Other Market Factors

In October 2008, banks and investment banks in leading financial centers such as London and New York started to cancel or significantly modify IT projects in development. The reason was that their world had been turned upside down. They were fighting for their survival. Suddenly they needed to better understand their and the market's liquidity. It was no longer important to be delivering the latest exotic commodity product. The market could care less about investments. In the fall of 2008, the financial industry was interested solely in solvency or insolvency depending on how lucky they were.

This is a severe example but it is a real example that shows how project portfolios and requirements for projects in progress can change dramatically. Reacting to these kinds of changes tends to distract teams and the results are significant drops in throughput, dramatic increases in lead times, often drops in quality, and a loss of predictability as the team recovers from the randomization that a fluctuation in the market caused to the internal workings of the project.

Clearly such events are assignable cause variations. They need to be accommodated using risk management strategies and tactics. There is a considerable body of knowledge on assignable cause variation or event-driven risk. Building a strong risk management capability as part of an overall organizational maturity goal will improve the predictability of a software engineering function whether it is using Kanban or not. However, kanban systems will exhibit greater predictability and the value-stream trust that comes with it, when risk is managed well.

Kanban systems have a number of other elements which assist in risk management. The WIP limit reduces risk as only a small fraction of work is in progress at any given time. Allocation of WIP limits across work item types and classes or service help to manage risk and absorb assignable cause variations. Other strategies are emerging and it is likely that a subsequent book will emerge detailing advanced methods for improving Kanban and better managing risk.

I've presented some material on managing risk with kanban systems that has emerged from the use of Kanban at conferences in 2009 and this can be found online[xxiv].

Staffing Resource Factors

In larger organizations, it is common for specialist staff to be shared across projects. I've seen specialists such as database architects, user interface designers, graphic designers, security analysts, auditors (of various kinds such as Sarbanes-Oxley), shared across teams and projects. Specialists like this are at best non-instant availability resources and apt to be full blown capacity constrained bottlenecks.

There are several ways to cope with shared resources in a kanban system. One example that emerged at Corbis was simply to tag tickets with the name of the required shared resource on a small orange ticket attached to the larger work item ticket.

Picture xx. Small Orange tickets with name of shared resource required are attached to Feature work item yellow tickets

This tactic was enough to alert management to specialists who were overstretched and causing work to drag.

Another approach is to treat non-instant availability of a shared resource by marking an item as blocked awaiting attention from the specialist. This does have the affect of bringing the number of instances and the average length of delay time to the attention of managers and the team as a whole. This may lead to discussions about how to change the process to minimize the impact.

Another strategy, and this will be most effective when the specialists is a capacity constrained resource, is to give the specialist his or her own Kanban board. The input to this board then provides an opportunity for the competing customers – the projects across the portfolio – to vie for attention based on class of service, cost of delay, work item type and other prioritization policies. It makes the work of the specialist transparent and the policies that control that work explicit. This allows the specialist to be utilized better.

Another strategy that may emerge is to tie use of the specialist to work item type or class of service. Based on the risk involved in a particular type of work or a particular class of service, the specialist is either needed or not needed. For example, would it be possible to analyze and classify work that had significant versus minor impact on the database schema? If so, then it would be possible to route only items with significant impact to the specialist database architect and to manage the quantity of this work. This would enable the managers to control the process so that the database architect specialist would or would not be the system bottleneck, depending on the quantity of work allocated that required attention from him or her.

This is a similar scheme to the one described in chapter 11 regarding the use of a temporary class of service. This approach that delineates a type of work or a class or service according to the risk profile is one of the more advanced ways that kanban systems can manage risk. The approach is not exclusive to Kanban but is better implemented in a system with an explicit WIP limit.

Difficulty Scheduling Coordination Activity

Another common source of assignable cause variation that causes work to block and flow to irregular is the challenge of coordinating external teams, stakeholders and resources. One common reaction to coordination challenges is to schedule meetings with a regular cadence. In some instances this is very efficient. However, it won't always be possible.

Flow may be interrupted by a governance or regulatory requirement that requires an audit or signoff. The people required to perform this function may not be instantly available or may be difficult to schedule.

In the first instance, assignable cause variation of this nature should be addressed by raising awareness and drawing attention to it with visibility and transparency. By marking items as blocked and raising visibility on to the source of the blockage the management, team and value stream stakeholders will become aware of the impact of such coordination challenges.

This should lead to some behavioral changes that improve the situation.

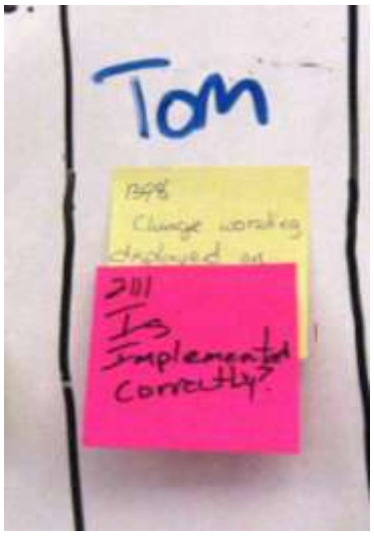
One tactic might be to exam the governance and regulatory rules and decide whether everything needs to be assessed, approved, audited, examined, or whatever the source of the blockage. Assuming that some risk profiling allows work to be bifurcated into two categories that do and do not need such a meeting to happen, then either work item type or class or service can be used to separate out the work. You can then use allocation of the WIP limit to both types to insure smooth flow.

Takeaways

- The study of variation in industrial processes started in the 1920s with Walter Shewhart and evolved through the work of W. Edwards Deming, Joseph Duran and David Chambers through the mid to late 20th Century.
- A study of variation and its statistical analysis method is at the heart of both the Toyota Production System (and hence, Lean) and Six Sigma methods for process improvement
- The work of W. Edwards Deming and Joseph Duran were a major inspiration for the work of the Software Engineering Institute at Carnegie Mellon University and the Capability Maturity Model (now Capability Maturity Model Integration or CMMI)
- Shewhart divided sources of variation into two categories: those internal to the process or system; and those external to the process or system.
- Internal variations are referred to as "chance cause" variations
- External variations are referred to as "assignable cause" variations
- There can be many sources of chance cause variations in the value stream of a software development lifecycle. Typical examples include work item size, type, class of service, irregular flow, and rework.
- There is a possibly endless list of sources of assignable cause variations. Typical examples include requirements ambiguity, expedite requests, environment availability, irregular flow, market factors, staffing factors, and challenges scheduling coordination or overhead activities
- Chance cause variation can be controlled through the use of policies that define the software development lifecycle and project management processes in use
- Assignable cause variations can be managed through the use of issue management and resolution and risk management capabilities and reduced or eliminated through root cause analysis and elimination capabilities
- Kanban systems produce better economic outcomes when coupled to a solid event-driven risk management capability.
- Kanban also offers additional ways to manage risk such as allocation of WIP limit across classes of service and work item types and use of risk profiling to separate work into types or classes and process accordingly
- Further work on advanced risk management strategies and tactics with Kanban is on-going and will be the topic of a future book

Chapter 20 Issue Management & Escalation Policies

When work in our kanban system becomes blocked for any reason, it has become the convention to indicate this on the card wall by attaching a pink sticky note to the card indicating the reason for the blockage. In electronic systems, there may be other ways to indicate that a work item is blocked and preferably the features of the electronic system should allow the reason for the blockage to be tracked separately.



Picture xx. Pink blocking issue (or impediment) item attached to Change Request work item directly affected

During the writing of this text I have noticed that some people attempting Kanban for the first time refer to these blocked items as bottlenecks. This is wrong. A block item might be clogging the pipe and restricting flow but it is not a bottleneck as described in chapter 17. It is neither a capacity constrained resource or a non-instant availability resource. In the same manner, a cork in a bottleneck is not a bottleneck. If you want to restore flow from the bottle, you simply remove the cork.

It's dangerous to think of blocked work items as bottlenecks because it leads to the wrong kind of thinking in order to resolve the problem. Blocked work items should be treated as special cause variations rather than bottlenecks. What is similar is the desired outcome. In both the case of a bottleneck and the case of a blocked work item, we want to resolve the issue in order to improve flow.

Blocked work items require an organization to develop a capability at issue management and resolution to restore flow as quickly as possible and prevention of recurrence requires the development of a root cause analysis and resolution (improvement opportunity) capability. The latter capability was discussed in chapter 19 as removal of special cause variations. The former is discussed here.

Managing Issues

It isn't good enough simply to mark and track work as blocked. Many early Agile software development tools, allowed for this. While it is useful information to know that an item, a story, a feature, a use case, is blocked, my observation of teams around the world has been that knowing something is blocked does not lead to the development of a strong capability at getting them unblocked.

It is essential to track the reason for the blockage and to treat it as a first class work item, albeit a failure load work item. A special work item type, Issue, is set aside for this purpose. Issues are tracked using pink tickets. They should be assigned a tracking number, and a team member, usually the project manager (or Scrummaster) should be assigned to insure resolution.

When a team member working on a customer valued item is unable to proceed, they should mark an item as blocked by attaching a pink ticket to it describing the reason for the blockage and creating an issue work item in the electronic tracking tool and linking it to the original work item. For example, there may be ambiguity in the requirements and a knowledgeable person was not available to instantly resolve the ambiguity. Or perhaps an environment setup is required and is currently unavailable. Or a specialist is required to work on the item and that person is unavailable due to vacation or sickness or other out of office time. And so on…

As discussed in Chapter 7, maintaining flow should be the main focus of discussion at the daily standup meeting. Hence, the meeting should be focused on discussing blockages and progress towards the resolution of the issues. The meeting should focus heavily on the pink tickets. Questions should be asked about who is working on resolution of the issue and the status of progress on resolution. Does the issue need to be escalated? If so, to whom?

Idle team members should be encouraged to volunteer to run issues to ground and generally to swarm on problems and assist however they can to resolve them and restore flow to the system. A team with a strong self-organization capability will tend to do this naturally, team members will volunteer to help resolve issues. Where that self-organization capability is yet to emerge, the project manager may need to assign team members to work issues to resolution.

Like all other work items, issues should be tracked. The tracking should include a start date and an end date and a link to all affected customer valued work items. Note that a single issue may be blocking more than one customer valued item. This is another sound reason for tracking issues as an independent work item and for having Issue as a distinct work item type. When choosing an electronic tool for tracking your kanban system, be sure to pick one that supports issue tracking as a first class type or a tool that is sufficiently customizable that you can create a type of work called Issue and designate that it will be displayed using pink (or red) cards in the visual display.

Escalating Issues

When the team is unable to resolve an issue on its own, or an external party required to resolve an issue is unavailable or unresponsive, then the issue must be escalated to a more senior manager or other department.

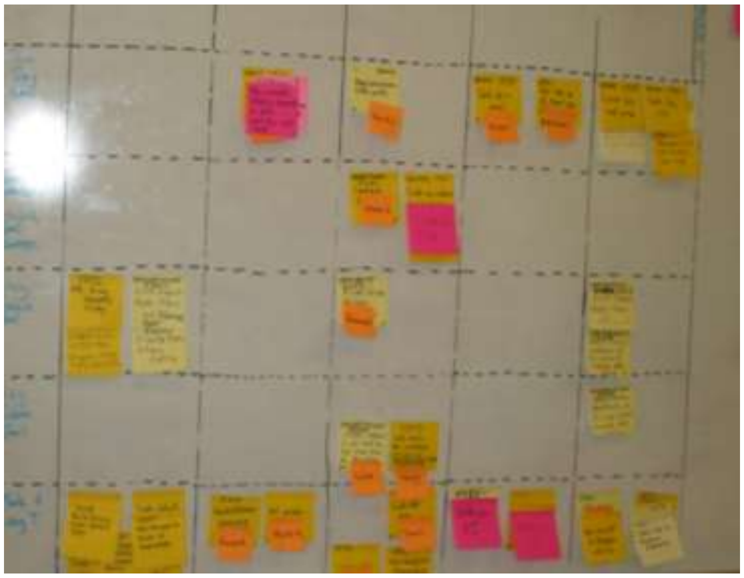
It is important for the organization to develop a strong capability at issue escalation. Without it, maintaining and restoring flow after a blockage can be problematic.

The foundation of a good escalation capability is a documented escalation policy or process. In Chapter 5 I discussed the power of developing organizational policies in a collaborative fashion. The escalation policies should be developed in such a collaborative fashion and a consensus agreed amongst departments involved in the value stream. The escalation policies should be widely known and understood and a document describing them should be readily available to all team members. There should be no ambiguity over how and where to escalate a problem. By taking the time to define escalation paths and write policy around it, the team knows where to send issues for resolution. This saves time discovering to whom an issue should be escalated and it sets expectations of those more senior individuals that they are expected to be a part of the process and that their responsibility to resolve issues will help to maintain flow and ultimately to minimize cost of delay (or optimize payoff from speedy delivery.)

Tracking and Reporting Issues

As I stated earlier, issues should be tracked as first class work items with their own work item type. The convention has evolved to use pink or red cards or sticky notes to visualize issues. A start date, an end date, as assigned team member, a description of the issue and links to the blocked customer valued items are the minimum requirements for an issue tracking system. Some history of efforts made in resolution, a history of assigned individuals, an indication of the escalation path, an estimated time to resolution, an impact assessment, suggested root cause fixes for future prevention, may also be useful fields to track.

While pink tickets on the card wall provide a strong visualization of how many items are currently blocked, it is also useful to track and report issues in other ways. A cumulative flow diagram of issues and blocked work items provides a strong visual indicator of the organizational capability at issue management and resolution. The trend in blocked work items over time indicates whether a capability of root cause analysis and resolution – improvement opportunities to eliminate special cause variations – is developing. A tabular report of current issues, assigned individuals, status, anticipated resolution date, affected work items and potential impact may also prove useful for day-to-day management on larger projects.



Picture xy. Board showing several blocking issues affecting multiple features

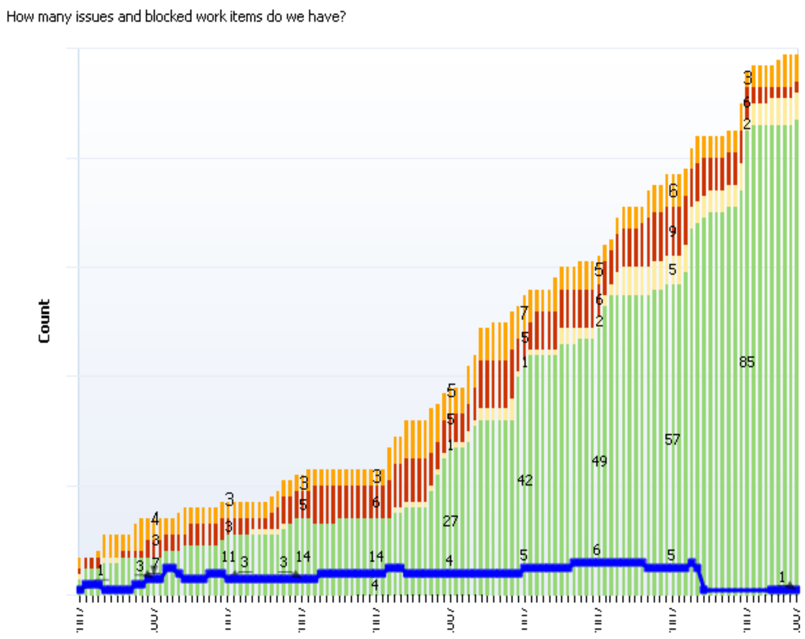


Figure xz. Issues CFD with overlaid Blocked work items graph

These reports should be presented at each operations review and time should be set aside to discuss the emergence and maturity of organizational capability at issue management and resolution and root cause analysis and resolution. The organization should be aware of the failure load impact of blocking issues. This will enable objective decisions about improvement opportunities and the likely benefits of investment in root cause fixes to prevent special cause variations.

Takeaways

- Kanban systems should have a first class work item type, Issue, used to track problems causing other customer valued work to block
- It has become the convention to use pink (or red) sticky notes on a card wall to visualize issues
- Pink issue tickets are attached to the items that are blocked
- A strong capability at issue management and resolution is essential to maintain flow
- Blocked work items and issues are not bottleneck. They should be managed as special cause variations rather than as capacity constrained resources or non-instant availability resources
- Issue management should be a strong focus of daily standup meetings
- A strong capability at issue escalation is essential as part of a strong capability at issue management
- Escalation policies should be clearly defined and documented and all team members should be aware of them
- Escalation policies work better when they were agreed collaboratively by all departments involved in the value-stream
- Issues should be tracked electronically
- Some reporting based on electronic data will facilitate day-to-day issue management and resolution on larger projects
- Use an Issues & Blocked Work Items cumulative flow diagram to visualize the development of capability at issue management and resolution and root cause analysis and resolution

Chapter 21 An Operational Decision Framework

Hopefully, by now you've learned that the Kanban approach requires teams and individuals to think for themselves. They are empowered to make process change decisions and to make day-to-day operational management decisions that hopefully maximize the value delivered while managing risk appropriately.

When making these day-to-day decisions it would be good to have some simple guiding principles. These principles should enable management to trust the team to make good quality decisions. I've developed two of these little sets of principles. The first I call The Lean Decision Filter and the second The Agile Decision Filter. When implementing Kanban I recommend that you teach both of these simple decision filters to your team and the management hierarchy. They will help instill a culture that delivers the Lean and Agile organization you are probably seeking. They will enable Kanban as Kanban enables them in a virtuous cycle.

The Lean Decision Filter

1. **Value trumps Flow**
2. **Flow trumps Waste Reduction**
3. **Reduce Waste to Improve Efficiency (Do not pursue economy of scale)**

This simple set of principles is based on some assumptions. The underlying assumption is that the goal of the business is to maximize value delivered. The whole Lean movement and its concepts, principles and paradigm are really based on this assumption. If your business has other goals then you may need a variant on this set of principles or perhaps Lean isn't for you.

Earlier in the book we saw examples of where it makes sense for value to trump flow. We introduce an Expedite class of service to make it explicit that Value trumps flow and the policies around qualification for expediting are explicitly value related. We know that expediting behavior will impact flow, increase WIP, reduce item throughput, and lengthen cycle time. Everything about expediting is bad, other than the fact that it gives us the possibility to deliver more value. It gives us an option to be opportunistic.

In another example, we read the story of Doug Burros, the build engineer, who wasn't instantly available and as a result, he became a bottleneck and the gating mechanism on flow. The tactical solution was to put a buffer in front of Doug. Buffers increase inventory and notionally increase cycle time. Buffers are waste.

However, the buffer was sized so that flow could be sustained without Doug becoming a capacity constrained bottleneck. The result was increased flow despite the increased waste. By increasing waste, we increased flow and by implication increased throughput and value delivered.

By gently challenging team members to justify day-to-day operational decisions and process changes against the first two statements in the Lean Decision Filter, we insure that the process is being optimized in a Lean fashion. This builds trust with managers that team members are using their empowerment wisely and can be trusted with it. This in turn frees managers to focus on more strategic problems and not get involved in day-to-day operational decisions. Only in circumstances of uncertainty where team members can't decide the obvious choice should they escalate, or in circumstances where the decision is strategic and significant to the business should a day-to-day decision be escalated. However, by their nature day-to-day decisions are rarely strategic or significant (in accounting terms) to a business, so escalation should be rare.

It's also important to understand the motivation for reducing waste. Reducing waste should enable improved efficiency in Lean terms, i.e. the cycle time to assigned (or touch) time ratio should be improved, or put another way, local cycle time should be improved, increasing the idle time for the workers. This has been described as Road Runner Behavior by the Theory of Constraints community after the cartoon character. The idea is simple. When you have work do it as quickly as possible, without sacrificing quality, then stop and wait for the next job to come along. Idle time for knowledge workers is not waste. Do not measure waste as time spent by individuals. Do not fall into the 20th Century trap of measuring efficiency around people and their time in motion and task loading, measure efficiency as time in motion for the work as it passes along the value stream.

Any process change made in the name of waste reduction should demonstrate a reduction in total cycle time, and an improvement in the cycle time to assigned/touch time ratio and an increase in idle time for some of the knowledge workers along the value chain. This idle time may be put to other uses, on lower class of service tasks or process improvement initiatives. This improvement need not be shown instantly. It may take a few weeks of months to come through in the metrics. The reason for this is that cycle time is a lagging indicator.

So when someone proposes a change, challenge it with, "Why are we making this change?" The answer may be "Because it will eliminate [this] wasteful activity?" Challenge that with, "And how will this improve cycle time, cycle time : assigned time ratio, and individual contributor idle time?" Expect a full answer to all three questions and with a more mature organizations ask for numbers. What is the expected cycle time reduction (in hours or days), for example?

At Corbis in September 2007, we knew that we had a 14 day queuing delay between our business analysis activity and our system analysis (detailed product specification) activity. We had already politically merged the two departments under the one manager 5 months earlier. During the summer preparation and cross-training had taken place to enable most of the combined department's staff to perform both roles. The result would be a combined business and systems analysis activity that would be performed generally by one individual without a hand-off. Only in a few exceptional circumstances where a specialist was required would there be a handoff. We knew from data that this change should produce an optimal 14 day reduction in end-to-end cycle time from 55 days to 41 days. We hoped for about 11 days down to 44 days. We made the change and we observed the metrics as they came in through the fall. The result was actually a full 14 day reduction in total cycle time.

This showed two things. Firstly, that we made a significant change that involved job title changes, employee training and role and responsibility changes, for the right reason and that it delivered the expected results. It also showed that we have an underlying model for Kanban and Lean and that this model can be used to make such predictions with accuracy. While I include only one example in this text, there have been many others.

The second part of the third statement in The Lean Decision Filter states that we should not pursue economy of scale. Economy of scale is the notion that we should increase batch size and quantity of WIP in order to efficiently amortize the transaction and coordination costs of receiving, developing and delivering the valuable working software. Pursuing economy of scale will typically manifest itself in decisions that increase batch size, increase cycle time, or reduce cadence. For example, a daily meeting may be considered inefficient, so the suggestion might be to move to every 2 days or twice per week. This would be an economy scale style decision. The team should challenge the suggestion and ask, "Why is the daily meeting so costly and inefficient? What could we do to our system to make the daily meeting more effective?" This approach is an approach that looks to reduce waste to improve efficiency while slipping the cadence of the meeting would be the opposite, an economy of scale approach. So as a general rule, challenge any process change that increases hand off batch size, increases cycle time, or reduces cadence. These are the indicators of hidden economy of scale thinking and the pursuit of 20th Century style efficiency.

The Agile Decision Filter

1. **Are we making progress with imperfect information? (or are we delaying in the pursuit of perfection early in the value stream?)**
2. **Are we increasing the level of trust and social capital, driving out fear? (or are we adding fear, uncertainty, doubt, bureaucracy and permission, reducing the level of trust and social capital?)**
3. **Are we treating WIP as if it were a liability? (rather than an asset?)**

I developed The Agile Decision Filter as a set of actionable management guidance based on the principles and values behind The Manifesto for Agile Software Development. I wanted to give managers rather than software developers a way of understanding how to act in a manner that was aligned with Agile values and would encourage the adoption of Agile software development techniques and lead to a cultural change that would ultimate enable improved business agility.

The first statement really underscores the nature of knowledge work – that's it's an information arrival process where we need to be doing it in order to discover all the information we need to know. We cannot discover all the information in advance. In mathematics it's the concept of a wicked problem - a problem which is changing as we are solving it. Delaying to acquire perfect information is waste. The bargain we are striking is that any rework incurred through new information arrival later in the process will be less than the waste incurred through a pursuit of perfection strategy. There are clearly some circumstances and contexts where this assumption will not be valid. Those tend to involve activities with huge cost of failure where lives may be lost or where the cost of rework would be astronomical in both time and money. This tends to apply only in large scale systems engineering activities involving delicate expensive hardware such as spacecraft, satellites and military equipment. So generally in most software development activities it makes sense to make progress with imperfect information.

This decision filter will primarily manifest itself with Kanban when we consider the pull criteria that an item must meet in order to be considered available for pull to the next stage in the value stream. It also has implications for how we treat rework as new information arrives. Do we sent the item back up the value stream? Do we leave it where it is and allow multiple people from different stages of the value stream to work on it? Do we work on it in situ, or do we send it back to queue and await reprioritization? All of these are choices. There is no correct answer. A team must decide its rework or refactoring policies for itself. What is important is that the kanban system should accommodate making progress with imperfect information and subsequent rework as new information arrives. By keeping The Agile Decision Filter in mind when establishing the working policies of the system, it can adapt and enable an Agile way of thinking. The long term result should be less waste and better business agility (through reduced cycle time and less WIP.)

The second statement in The Agile Decision Filter reflects the Lean principle of respect for people but goes further and really asks for the establishment of a high social capital culture within the organization and potentially with external value-stream partners. Japan is a naturally high trust country and has a high social capital culture. The concept of trust is not widely explored in Lean literature and it is likely that it gets overlooked because it is so embedded in the underlying culture.

A low trust culture tends to require a lot of contracts that require negotiation to agree in the first place. They then require audit to verify compliance and some form of arbitration in the event of non-compliance. All of these activities are wasteful. They do not add value to the end product even if they may enable a working relationship between teams, organizations and businesses. By enabling a high trust culture we eliminate a lot of waste. By challenging day-to-day decisions with respect to whether they increase or decrease trust, we discourage the addition of wasteful activities to our process.

We also encourage a risk taking, failure tolerant organization without fearful employees. Fear can cause delays or inaction. Fear will reduce levels of innovation and process improvements. Fear will delay resolution of impediments and blocking issues. Fear will cause delay through inaction. Fear ultimately increases cycle time and reduces the ability to improve performance at all levels. You can fight fear in the organization by challenging day-to-day decisions with the 2nd statement of The Agile Decision Filter.

The 3rd statement of The Agile Decision Filter asks us to reduce WIP and focus on shortening cycle time. It pushes against the 20th Century notion that WIP was an asset. Until 2002, knowledge work WIP could be capitalized on the balance sheet of firms as an intangible asset, under GAAP accounting rules. Technically it still can but new rules mean the transaction costs of revaluation are prohibitive and the practice has largely stopped. So managers grew up in a world where hording WIP improved the financial numbers of the business and was generally encouraged by local management metrics. Managers were often rewarded for hording WIP because the economy of scale made their efficiency numbers look better. A common practice was to measure efficiency based on employee loading. So maintaining a large stack of WIP to keep everyone busy, regardless of whether they were working in a bottleneck or not, was good practice.

If we flip that mindset and ask managers to treat WIP as a liability then they will naturally want to get it off their hands. The best way to do this is to complete their work activities and all it to be pulled downstream. Treating WIP as a liability encourages cycle time reduction and Lean flow efficiency. It encourages the Road Runner style of behavior.

The reality is that knowledge work in progress is probably best thought of as a rapidly depreciating asset. The knowledge received is perishable and short cycle times are imperative to maximize the value delivered. However, as an amplification to drive appropriate behavior, asking managers and the team to think of it as a liability drives the desired behavior without ambiguity and debate over the rate of deprecation.

Together The Lean Decision Filter and The Agile Decision Filter provide a belt and braces approach to insuring day-to-day operational decisions are made in a fashion that drives the right behavior, culture and outcomes. By assimilating these ideas into the culture of the organization, individuals will keep others honest by challenging their proposals and decisions, and the outcome will be a self-organizing, Lean organization that delivers significantly improved business objectives.

Takeaways

- Trust in empowered employees by managers can be facilitated by giving everyone a small set of principles with which to challenge and align all day-to-day operational management decisions
- The Lean Decision Filter and The Agile Decision Filter were created to provide two short lists of three principles that boil down the essence of Lean and Agile thinking as actionable guidance in the form of decision filters
- Each operational decisions should be justifiable against the one or all of the criteria in the decision filters and should certainly not break any of the decision filter criteria
- Value trumps flow
- Flow trumps waste reduction
- Reduce waste to improve efficiency rather than pursue economy of scale
- Make progress with imperfect information and correct as new information arrives
- Encourage a high trust, high social capital culture and eliminate negotiation, contracts, verification, arbitration and as much bureaucracy as possible
- Treat WIP as a liability to shed quickly rather than an asset to be hoarded
- Inculcating The Lean and Agile Decision filters in your organization will change the culture and enable a highly empowered self-organizing Lean and Agile organization
- A Lean and Agile organization will deliver significantly improved performance over time and deliver significant better business outcomes

[i] Anderson, David J., Agile Management for Software Engineering – Applying the Theory of Constraints for Business Results, Prentice Hall PTR, 2003
[ii] Beck, Kent, Extreme Programming Explained: Embrace Change, Addison Wesley, 2000
[iii] Beck, Kent et al, The Principles Behind the Agile Manifesto, http://www.agilemanifesto.org/principles.html
[iv] Goldratt, Eliyahu M., What is this thing called The Theory of Constraints and how should it be implemented? North River Press 1999
[v] Anderson, David J., and Dragos Dumitriu, From Worst to Best in 9 Months – Implementing a Drum-Buffer-Rope Solution in Microsoft's IT Department, Proceedings of the TOCICO World Conference, Barcelona, November 2005
[vi] Belshee, Arlo, Naked Planning
[vii] Hiranabe, Kenji, First article on InfoQ
[viii] Hiranabe, Kenji, Second article on InfoQ
[ix] Augustine, Sanjiv, Managing Agile Projects, Prentice Hall PTR, 2005
[x] Highsmith, Jim, Agile Software Development Ecosystems, Addison Wesley Professional, 2002
[xi] The Nokia Test is attributed in origin to Bas Vodde, described here by Jeff Sutherland who has adopted and updated it, http://jeffsutherland.com/scrum/2008/08/nokia-test-where-did-it-come-from.html
[xii] Beck et al, The Principles Behind the Manifesto, http://www.agilemanifesto.org/principles.html
[xiii] Jones, Capers [Get title of book] 2002
[xiv] Ambler, Scott, Agile Modeling [get full reference]
[xv] Chrissis et al, CMMI – get full reference
[xvi] Sutherland, et al – Agile 2007 & 2009 papers on CMMI ML5 and Scrum adoption
[xvii] Larman & Vodde – get full reference
[xviii] Willeke, Eric, with David J. Anderson, Eric Landes(editors) Proceedings of the Lean & Kanban 2009 Conference, Wordclay 2009
[xix] Beck et al, Principles Behind the Agile Manifesto, 2001, http://www.agilemanifesto.org/principles.html
[xx] Chrissis et al, CMMI – get full reference
[xxi] Sutherland, et al – Agile 2007 & 2009 papers on CMMI ML5 and Scrum adoption
[xxii] Larman & Vodde – get full reference
[xxiii] Willeke, Eric, with David J. Anderson, Eric Landes(editors) Proceedings of the Lean & Kanban 2009 Conference, Wordclay 2009
[xxiv] Anderson, David J. New Approaches to Risk Management, Agile 2009, Chicago, Illinois, http://www.agilemanagement.net/Articles/Papers/Agile2009-NewApproachesto.html