Contrasting Scrum and Kanban

In This Chapter
There has been much talk about the difference between Scrum and Kanban. A simple comparison often leads one to think that Kanban is merely Scrum without iterations. But that would be like calling a Toyota Corolla with a bigger engine a Ferrari. The methods are different, but much more important, the mindset and paradigm from which they spring are different. And these differences add up to more than the sum of the parts.

This chapter compares Kanban and Scrum by looking at the differences in their paradigms and attitudes and then walking through how each would solve a common problem – management asking the team to expedite some work.

Takeaways
Key insights to take away from this chapter include:

- Scrum and Kanban are based on a different set of perspectives (beliefs), approaches and attitudes
- Scrum is more exclusive of management than Kanban
- This results in a less collaborative approach between managers and the developer team when exceptional events occur
- Kanban creates more of a team based approach because there is an explicit model of how the team is supposed to work

Perspectives / Beliefs
Black box Vs Requiring Feedback

Scrum is an empirical process (black box). “Black box” means that you study it from the outside. While we agree that software development is a non-predictive process in the sense that one must work within the plan do study act (PDSA) cycle this does not imply that the process itself is a black box. However, it does appear that most in the Scrum community believe that creating a defined process and continuing to question and improve it based on feedback is not a productive undertaking. This means that those doing Scrum must learn how to apply it within their own context based primarily on the experience they have acquired.

Kanban is a non-predictive process that uses feedback to guide itself. However, non-predictive does not mean that there is not a model underneath it. Playing poker, for example, is a process that is not well defined in the sense you can’t say – do this, then do that. But it has a model underneath it that can be used to improve one’s play. One can try certain strategies based on this model and then see what results
are achieved. If the results are good, that validates the strategy to some extent. If the results are bad, one should challenge the strategy.

**Approach to Improvement**

“Inspect and Adapt” Vs Improving the Underlying Model

In “The Scrum Papers”, Jeff Sutherland and Ken Schwaber\(^1\) say “In this paper we introduce a development process, Scrum, that treats major portions of systems development as a controlled black box.” The basis for this is “Concepts from industrial process control are applied to the field of systems development in this paper. Industrial process control defines processes as either ‘theoretical’ (fully defined) or ‘empirical’ (black box). When a black box process is treated as a fully defined process, unpredictable results occur.”

I had always interpreted this to mean that you need feedback in order to control a software development process. I agree with this. However, it actually says the process is a black box process – meaning one learns about it through observation.

We believe there are two different orthogonal issues. The first is the one Jeff and Ken mention – defined or empirical. The second, distinct issue is whether a process is predictable or requires feedback to control. These are not the same thing. Defined or empirical relates to whether there is a model underneath the system that one can learn. Predictable Vs Requiring Feedback means can you predict the outcome of the system. I’ll contrast these with some examples:

<table>
<thead>
<tr>
<th>Defined Vs Empirical</th>
<th>Predictable Vs Requiring Feedback</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defined</td>
<td>Predictable</td>
<td>Volume of water flowing through a pipe when flow rate is controlled.</td>
</tr>
<tr>
<td>Defined</td>
<td>Requires feedback</td>
<td>Playing poker, flight path of an airplane, mixing chemicals to create other chemicals, driving a car</td>
</tr>
<tr>
<td>Empirical</td>
<td>Predictable</td>
<td>Sorting beads coming from a thoroughly mixed batch of beads after 300 have come up the same color</td>
</tr>
<tr>
<td>Empirical</td>
<td>Requires feedback</td>
<td>Playing a sport like soccer</td>
</tr>
</tbody>
</table>

Scrum incorrectly defines software development as “an empirical & requires feedback” process when it is actually a “defined & requires feedback” process. This is not a subtle point. If software development is “empirical & requires feedback” then our best approach is to treat it like a black box. Do something, inspect and adapt, try something else – basically PDSA. However, if software development is a defined process that requires feedback to control it, then there are much greater possibilities available.

---

Kanban specifically is based on the notion of pull and flow. Part of its method for process improvement is to create a model for managing the amount Work in Process (WIP) to an appropriate level. Too much WIP causes thrashing and delays delivery of value. Too little WIP causes delays where no work is taking place. Kanban practitioners use their understanding of flow to manage WIP by creating agreements as to how much WIP can be present. This WIP is typically managed through the use of Kanban cards. The number of cards allowed, the agreements about these cards and the team’s understanding of flow and pull comprise a model of how the teams’ software development process is working.

This enables the team to think of the process they follow as a model of the best way they know how to do their work. They can see the results they get, use that to improve the model, see the results achieved from that, and so on. This is a classic Plan Do Study Act (PDSA). However, they can do more. When faced with a situation where they don’t know what to do, either a new situation or a situation where there is an impediment, they can use their understanding to make a prediction about how a new solution would work. Then, they can try the new solution and see if their actual results match their understanding (that is, their model).

**Attitudes**

**Avoid Management Vs Include Management**

In many ways Scrum has taken the path of avoiding management whenever possible. It is set up on the notion that the team must be free to choose whatever development approach necessary to best discover and build the software product. While we believe that software developers must be able to define and modify their own process, we don't believe this means a disassociation with management. Let’s look at the two different attitudes expressed towards management by Scrum and Kanban.

In Scrum, roles are split into two types – “Chickens” and “Pigs”. Chickens, which include management, are not part of the actual Scrum process. While chickens must be taken into account, they are not part of the team. This division makes it difficult for chickens to see how the pigs (the development team) are working.

In a sense, Scrum has taken the two main roles of management (team management and product management) and rolled them into two new roles – the Scrum Master and the Product Owner. The Scrum Master’s role is to facilitate the team. Their primary job is to remove impediments the team confronts. They are to act as a buffer between the team and any outside influences that might distract the team. The product owner represents the voice of the customer – ensuring that the team works on the right things. While these roles sound productive, there is an important side affect they have – one of insulating the team from management.

Kanban teams take a different approach. Two foundational legs of Kanban are Lean’s principles of flow and pull. Kanban is also consistent with Lean’s management philosophy that management plays a key role in product development. However, this role is neither one of command and control nor one of servant leadership. Rather, it is one of direction (setting objectives, not ‘how to’s), education and process improvement. In other words, a Lean manager’s role is to provide leadership. They are to set
the objectives, educate the team on better ways to do things and ensure the team follows what the
team has decided is their best way of doing things.

The bottom line is that Scrum excludes management, dealing with them only when necessary and in
prescribed ways. If management becomes overbearing, Scrum teams can pull out the big gun and
threaten to abort the sprint. This, unfortunately, is neither a cooperative nor viable alternative.
Kanban, on the other hand, is inclusive of management and creates a transparent process to
management. When things change and have to be expedited, management can see the impact of such
a change. This enables management and teams to work together.

A “Case” Study
Let’s see how these differences play out in the real world. Imagine two Agile teams at two different
companies. At company ABC they use Scrum while at company XYZ they use Kanban.

Agreements between the Scrum team and management is that the team expects the Product Owner to
prioritize the features on the backlog, but that once they are selected the team is free to do what it
wants in order to meet their commitment. There is not really much collaborative work between them.
Management prioritizes (through the PO), the team implements. If management tries to get heavy
handed and demand something, the team can decline to work (i.e., they can abort the sprint).
Managers are told if something urgent comes up they need to wait to the end of the sprint in order to
get it on the list. The team thinks this isn’t so bad – on average management will only have to wait a
week, or two at the most. Management, however, may get the sense that they used to be able to get
the team to work on something immediately.

The Kanban team, on the other hand, spells out its work flow and why they limit WIP to management.
This is done in the language managers understand – here’s where we figure out what the customer
wants (analysis), here’s where we figure out that we can verify we’ve done what they want (test
specification), here’s where we design it, here’s where we build it, here’s where we validate it we built it
right (acceptance test), … They make it clear that they are managing the work they are doing to limit
work to the capacity of the team. This keeps their efficiency high while enabling them to respond
quickly to requests.

They reach an agreement with managers that the team will always pick the top items put on their queue
and get it done as fast as possible using the best development methods they know for quality. If
management has to expedite something above work in process, they agree to a convention of the “silver
card” which always moves to the top of each queue.

At The Scrum Team
At ABC Company, when a VP puts some pressure on the development manager, the conversation may
go something like this:

Product Manager: Joe (one of their VPs) has just told me we need to get Feature X done immediately.

ScrumMaster: Great. We’ll move that to the top of the backlog and we’ll do it next sprint.
**Product Manager**: What part of “immediately” do you not understand? He wants it done now.

**ScrumMaster**: Yes, I understand what he wants, but that’ll be disruptive and overall have a negative effect. When we started Scrum we agreed that we could work without interruption during the sprint.

**Product Manager**: Well, that’s true, but that was intended for when things were going normal. Now I’m getting heat to get this done. I’m sorry, but just have the team work a little hard the next week to get it in. I don’t ask for this very often.

**ScrumMaster**: Well, I think if we put this thing in, we have to take something out.

**Product Manager**: You know we can’t do that. If you don’t keep your sprint commitment then that’ll impact the other teams that are depending upon you.

**ScrumMaster**: Can’t you just go back to Joe and tell him it’ll be disruptive?

**Product Manager**: Would you want to do that?

**ScrumMaster**: Well, no.

**Product Manager**: Good. Then we’re agreed. You’ll all just buckle up a little and get this small thing done. I really appreciate it.

**ScrumMaster (to himself)**: Great. I guess I could have told him about our agreement to abort the sprint but I know that’ll be a CLM (career limiting move). Well, having the team work an extra weekend is still better than the way it used to be.

**At The Kanban Team**

At XYZ Company, when a VP puts some pressure on the development manager, the conversation may go something like this:

**Product Manager 1 (PM1)**: Joe (one of their VPs) has just told me we need to get Feature X done immediately.

**Kanban Team Leader (KTL)**: Great. Just move it to the top of the queue and we’ll pull it next.

**PM1**: What part of “immediately” do you not understand? He wants it done now.

**KTL**: So you want us to drop everything we are doing and get to this?

**PM1**: Yes, that’s what immediately means.

**KTL**: Do you think it’d be ok for us to let people finish the current task they are working on so they can at least get closure on that? Most everyone would only need a day or so to complete that. Then we’d get on this by using the “silver card” and everyone appropriate would give it their full attention.

**PM1**: Yes, that’s ok. Joe’ll be pleased to know that he’s going to get his work done while not being too disruptive.
KTL: Oh, it’ll disrupt us, but if that’s the right business decision, no problem.

KTL puts white card on with Joe’s request and notifies other product managers/VPs of its presence.

Product Manager 2 (PM2) calling PM1: I see you’ve white-carded Feature X. Are you aware that this will slow down the 3 features currently being worked on? I need those features to be able to hit our release schedule.

PM1: Well, Joe said we needed to get this done. We just got a big account that needs feature X. If we get that done quickly we can make thousands of dollars.

PM2: OK, but we’ve made a lot of other promises as well. I don’t think we can knee-jerk react here.

... I could take this conversation either way. Perhaps the PMs resolve it, perhaps they go to Joe. Maybe Joe realizes his mistake, perhaps he decides the impact is too great, perhaps he stays stubborn and can outrank everyone else. The point is the conversation got elevated to where it should be – at the product management level. Scrum, unfortunately, typically deals with this conflict primarily at a single product manager – team level. The problem is not between the product manager and the team, it is the fact that the product managers aren’t prioritizing across themselves.

By creating visibility on the process, the impact of the demand can be seen across the organization. The Kanban team doesn’t have to take an strong and uncomfortable stand, they merely need to let the business side see the impact of the effect their decisions will have on the productivity of the organization. The software team is not a black box to management, so management can work with the team more effectively.

Challenges Scrum Teams Must Overcome That Kanban Doesn’t Create
We have heard that Kanban should be used for more mature teams. That is, start with Scrum, move to Kanban. We suspect that this attitude has been created because many mature Scrum teams have move to Kanban. But we actually believe that Kanban is easier to implement than Scrum in many cases. Some of the reasons for this is that Kanban does not cause a lot of problems that newbie Scrum teams need to overcome. Here are a few that come to mind.

Starting Too Many Stories
Coaches of new Scrum teams have come to expect that teams will end up st many stories. War stories between Scrum coaches virtually always retell of the team that finished 90% of 100% of their stories – only to realize at the end of the sprint that they got no value for doing so. Value is only given when a story is completed. While this lesson is learned reasonably quickly, Kanban, with its WIP limits, gives guidance to the team regarding the number of stories to have in play at any one time.

Assigning Resources During the Sprint
Scrum attempts to solve this problem by having a cross-functional team. However, cross-functional does not guarantee that any specialized person will be available at the times they are needed. Many
Scrum practitioners take it a step further and say the team should be comprised of generalists. In many situations, this is just not practical. At the extreme case, you may have someone with deep domain knowledge (e.g., a stress analyst in aircraft design with a PhD in the mathematics required to do the equations as well as 10 years of experience). Even in less extreme cases, certain people have domain knowledge or familiarity with legacy code that requires their attention. While Scrum will tell you to remove this impediment (definitely a good idea) what do you do while you are doing this? Scrum provides little guidance.

Kanban will manage this by having those people with specialized knowledge work on those areas for which they represent the bottleneck in the process. Attending to flow will have others who can get a task done but who perhaps aren’t the best person for it, do tasks as necessary. For example, Tom may be best at a particular task, while Mary can do it. If Tom is jammed because of other demands, when a task that Tom can do well, but Mary can also do (albeit not as well), Mary will take the task. Scrum can manage this in an ad hoc manner, but Kanban’s defined work flow and attention to small queues will handle this in a smoother fashion.

If this method results in larger queues than desired, then features requiring special skills can be identified. These features can be limited from starting by having the appropriate number of Kanban cards for them. This avoids starting a feature that won’t be able to be completed due to some specialized skill not being available. This also clearly identifies the bottleneck so a business decision can be made as to whether to cross-train someone or not.

Keeping Stories From the Same Feature Together
When Scrum teams work on complex features, they often need to break them down into many smaller stories. Scrum does not prescribe the use of Minimal Marketable Features (although more and more Scrum teams are learning to use them). We have often seen Scrum teams do release planning by breaking their features down into stories and then congregate stories into releases. The difficulty here is that stories then need to be separately tracked as to which story they came from. Unfortunately, this is sometimes not done. We have heard Scrum coaches suggesting working on “the one big thing” where this OBT is presumably a feature or set of features. But, because it isn’t explicit, there is a certain amount of lack of clarity involved. Furthermore, if the release includes several features, many may find themselves in process during the sprints involved in a release. If something needs to get descoped, this larger WIP may cause problems.

Epics, unfortunately, don’t necessarily help out here. That’s because an epic does not contain the notion of ‘minimal’. It is mostly thought of as a either a big story (before being broken down) or a collection of smaller stories.

In Kanban, MMFs are typically what is put in the front of the pipe. An entire MMF flows through the pipeline in as efficient a manner as possible – because of the pull method and limiting WIP management involved. One breaks down the features into stories and tasks as they need to be and just before they are worked on.
**Breaking Up Features**
We have always been asked – “how can you keep the big picture in mind while working on little pieces?” The answer is you must start with the big picture. In other words, you don’t want to break a feature down into little pieces and then put them together. Scrum does not prescribe doing this, but it also doesn’t tell you not to. Kanban’s value stream view implies full features are being worked on.

**Having Test Keep Up with Developers**
A chronic problem many Scrum teams have is that testers can’t keep up with developers. While teams understand that they should define their tests first, testers are typically required to work write the tests and may not be able to keep up. The question is, how do we get this to happen? A common approach is to tell developers to work on helping the testers. But devs often don’t want to do this and it is not uncommon for new stories to be started while others are not yet tested. While Scrum should be able to help here, if the team hasn’t completed jelled or bought into Scrum it is just too easy for a developer to say – “well, I thought it was best if I did this.”

Kanban has teams create an agreement as to how many tasks are allowed in the different stages of development at any time. Devs are not free to do what they want individually. Rather, as a team they agree to a process and abide by the team decision. The limited number of Kanban cards force devs to not continue to start new stories once a certain number of testing tasks are outstanding (because their will be no more kanban cards left). Devs can argue over whether these limits are a good thing. This is actually a good thing – the discussion. This now has developers discussing amongst themselves what the best way to do development is. One can see how fast and what quality at which the software is being developed. Instead of each developer having his own opinion, the team can quantifiably validate the best set of limits.

**Summary**
Scrum and Kanban can be viewed as a collection of practices. Doing so, however, will miss the fundamental difference between the two. Scrum is a team-based approach geared towards having the team figure out how to best develop software in isolation from management. While the team presents progress to management on a daily basis, little is done to explain to management the manner in which the team works. Kanban teams, however, create great visibility into their approach. This enables management to make decisions based on business need while seeing how such decisions will affect the team.

Kanban’s explicit model of development also enables team members to work together to investigate the best way to do their work. By trying things and seeing how they work, the team’s model (explicitly defined by the value stream, kanban cards and other items) can be validated and improved.

**Try This**
These exercises are best done as a conversation with someone in your organization. After each exercise, ask each other if there are any actions either of you can take to improve your situation.
• Think of when a team you’ve been involved with had too many things in process.
  o How did you manage when an item was started?
  o How were these rules formulated and communicated?
• How much of your team’s learning curve involved the management of the amount of work in process?
• How did you resolve differences of opinions on the development team about how developers and testers should work together?
  o Would explicit rules have helped?
  o Did you ever get agreement by the team members that a team process would be decided on?
  o If you did, was it possible to create such a process?

Recommended Reading