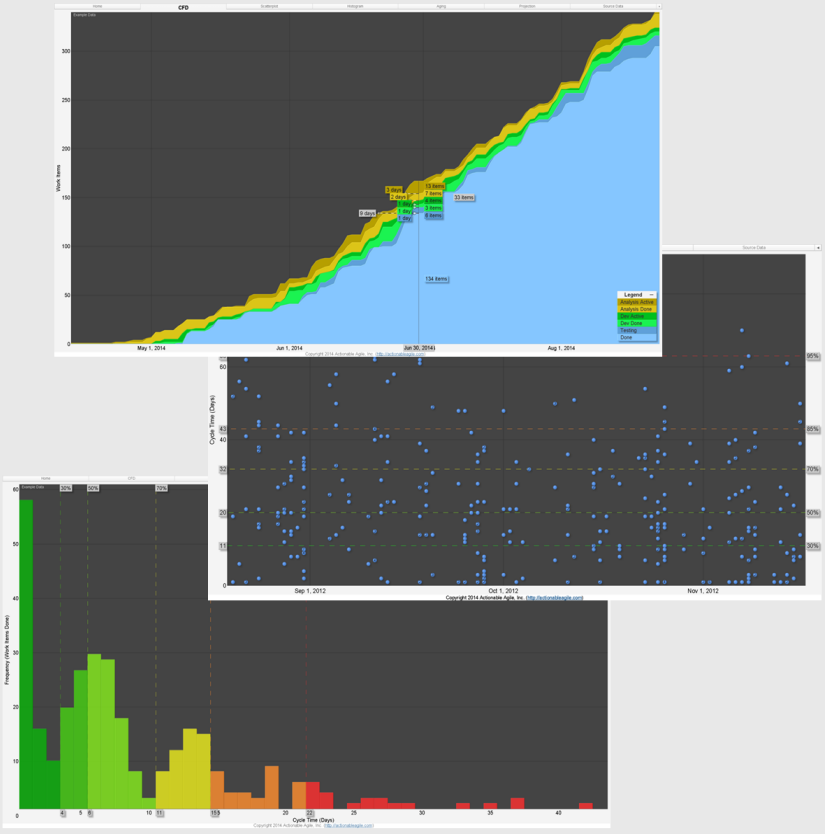
ActionableAgile™Press

Actionable Agile Metrics for Predictability

An Introduction



Daniel S. Vacanti

Actionable Agile Metrics for Predictability

An Introduction

Daniel S. Vacanti

This book is for sale at
http://leanpub.com/actionableagilemetrics

This version was published on 2016-05-16

ISBN 978-0-9864363-2-1

Tweet This Book!

Please help Daniel S. Vacanti by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ActionableAgile.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#ActionableAgile

To Ann, Skye, Sicily, and Manhattan: the only measures of value in my life.

Contents

TABLE OF CONTENTS **1**

Chapter 13 - Pull Policies **3**

TABLE OF CONTENTS

Preface
PART ONE - FLOW FOR PREDICTABILITY
Chapter 1 - Flow, Flow Metrics, and Predictability
Chapter 2 - The Basic Metrics of Flow
Chapter 3 - Introduction to Little's Law
PART TWO - CUMULATIVE FLOW DIAGRAMS FOR PREDICTABILITY
Chapter 4 - Introduction to Cumulative Flow Diagrams
Chapter 5 - Flow Metrics and CFDs
Chapter 6 - Interpreting CFDs
Chapter 7 - Conservation of Flow Part I: Arrivals and Departures
Chapter 8 - Conservation of Flow Part II: Commitments
Chapter 9 - Flow Debt
PART THREE - CYCLE TIME SCATTERPLOTS FOR PREDICTABILITY
Chapter 10 - Introduction to Cycle Time Scatterplots
Chapter 10a - Cycle Time Histograms
Chapter 11 - Interpreting Cycle Time Scatterplots
Chapter 12 - Service Level Agreements
PART FOUR - PUTTING IT ALL TOGETHER FOR PREDICTABILITY
Chapter 13 - Pull Policies
Chapter 14 - Introduction to Forecasting
Chapter 15 - Some Thoughts on the Monte Carlo Method
Chapter 16 - Getting Started with Flow Metrics and Analytics

PART FIVE - A CASE STUDY FOR PREDICTABILITY
Chapter 17 - Actionable Agile Metrics at Siemens Health Services: A Case Study
Acknowledgements
Bibliography
About the Author

Chapter 13 - Pull Policies

Most airports around the world allow access to the flight departure area if a person can prove that he is a passenger who is indeed flying that day. This proof usually takes the form of a valid boarding pass and a valid government-issued ID.

The United States is no exception to this rule. In the U.S., the Transportation Security Administration (TSA) is responsible performing passenger checks. TSA agents are stationed right before security and passengers wishing to get to the departures area must first check-in with these agents.

Many small airports in the U.S. staff only one TSA agent to perform traveler validation. At those small airports during busy periods, quite a long queue will form in front of the sole agent. Little's Law tells us that as more and more people join the queue, those people can expect to wait for longer and longer amounts of time to get through the checkpoint (on average). In this scenario, if you are a regular passenger, do you see the problem with predictability?

It gets worse.

In an attempt to streamline the process for what are considered low-risk passengers, the TSA has introduced something called "TSA Pre-check" (TSA Pre). Passengers who are certified as TSA Pre do not have to go through the whole security rigmarole of taking off shoes, taking off belts, taking off jackets, and removing laptops. That is great if you are TSA Pre. The problem is that you still have to go through the upfront TSA passenger validation outlined previously. However, the TSA has attempted to

solve this problem by establishing a different lane for TSA Pre passengers to queue in to get their credentials checked. So now there are two lanes for two different types of passenger: a first lane called TSA Pre (as I have just mentioned) and a second lane that I am going to call "punter". In the small airports, unfortunately, there is still usually only one upfront, credential-checking agent to serve both of these lines. The TSA's policy is that whenever there is a person standing in the TSA Pre line, that the agent should stop pulling from the punter queue and pull from the TSA Pre queue. See a problem with overall predictability yet?

It gets worse.

In addition to a separate TSA Pre lane there is usually a separate "priority lane" for passengers who have qualified for elite status on an airline. These passengers still have to go through the same security checks as the other punters, but they do not have to wait in a long line to get the upfront ID check. To be clear, this is technically not a TSA thing, it is usually an airport/airline thing. However, at those small airports, it is the single TSA agent's usual policy to look at the TSA Pre line first. If there is no one there, she will look at the priority lane next and pull people from there. Only if there is no one in the TSA Pre or priority queue will the agent start to pull again from the punter line. See a problem yet?

It gets worse.

As I just mentioned, everyone who wants to get air side at an airport must go through this upfront ID check. Everyone. This includes any and all airline staff: pilots, flight attendants, etc. Crew members can usually choose whatever line they want to get their credentials checked (TSA Pre, Priority, or punter). Further, once they are in those lines, the crew are allowed to go straight to the front of their chosen queue regardless of how many

people are ahead of them. At those small airports, the sole TSA agent first looks to see if there are any airline crew in line. If none, then they look to see if there are any TSA Pre passengers. If none, then they look to see if there are any priority passengers. If none, then they finally pull from the punter line. See a problem yet?

If you are in the punter line, guess what you are doing while that lone TSA agent pulls passengers from those higher priority queues? You got it: waiting. What do you think this is doing to the predictability of the punter queue? In other words, how many assumptions of Little's Law have been violated in this airport scenario? Is Little's Law even applicable here?

Class of Service

This airport screening example is a classic implementation of a concept known as Class of Service (CoS):

A Class of Service is a policy or set of policies around the order in which work items are pulled through a given process once those items are committed to (i.e., once those items are counted as Work In Progress).

That is to say, when a resource in a process frees up, CoS are the policies around how that resource determines what in-progress item to work on next. There are three subtleties to this definition that need to be addressed up front.

First, a Class of Service is different than a work item type (I spoke about how to segment WIP into different types in Chapter 2). This point can be very confusing because many a Kanban "expert" uses these two terms interchangeably. They are not. At least, not necessarily.

We can choose to segment our work items into any number of types and there is no prescription as to what categories we use for those types. Some previous examples I have given for work item types are user stories, defects, small enhancements, and the like. You could also segment work items into types using the source or destination of the work. For example, we could call a unit of work a finance work item type, or we could say it is an external website work item type. Or we could call a unit of work a regulatory work item type or a technical debt work item type. The possibilities are endless. And, yes, one of the ways you could choose to segment types is by Class of Service—but you do not have to. I have always thought a better way to apply CoS is to make it a dimension of an existing type. For example, a work item of type user story has an expedited CoS, a work item of type regulatory requirement has a fixed date CoS. But that is just personal preference. Just know that work item types and CoS are different. Do not let the existing literature out there on this stuff confuse you.

To be clear, you can have any number of types of Class of Service as well. The most talked about ones happen to be Expedite, Fixed Date, Standard, and Intangible. But those are only four examples of limitless kinds of Class of Service. Any time that you put a policy in place (explicit or not!) around the order in which you pull something through a process, then you have introduced a Class of Service.

The second subtlety of the above definition is that CoS does not attach until a work item has been pulled into the process. I cannot stress this point enough. There is absolutely no point in having a discussion about whether work item A is an Expedite (for example) and whether work item B is a Fixed Date while both items A and B are still in the backlog. The reason for this is, as I have

mentioned so many times before, is that while those items are still in the backlog there is no confidence that either will ever be worked on. Additionally, it is entirely possible that once committed to, our SLA would predict that we need not give any preferential pull order to that item. For example, let's assume that it is February 1 when we pull a new item into our process. Let's further say that this new item has a due date of February 28, and that the SLA for our process is 11 days. In this case, our SLA would predict that this item will complete well before its due date so there would be no point in giving it any preferential treatment. Given both of these scenarios, why waste time determining the order of pull before an item is in the system? That is why the decision of what CoS to use happens only at the time of an item's first pull transaction.

Which brings me to the last subtlety about CoS. The order in which items are pulled once committed to is very different from the decision criteria around what item to work on next at input queue replenishment time. Again, this is a very subtle but very important distinction. The criteria for what items we pull next off the backlog are very different from the criteria around the order in which we pull those items once in progress. If this concept is still ambiguous to you, then hopefully I will have cleared it up by the end of this discussion.

The Impact of Class of Service on Predictability

In the last chapter, I mentioned that most teams do not understand how the improper implementation of pull policy—whether explicit or not—negatively impacts their system's predictability. They do not understand these

negative impacts because CoS has either never been properly or fully explained to them. I would like to quantify these negative impacts by examining a pull policy scenario that I have set up for you.

In this particular example, we are going to be operating a process that looks like Figure 13.1:

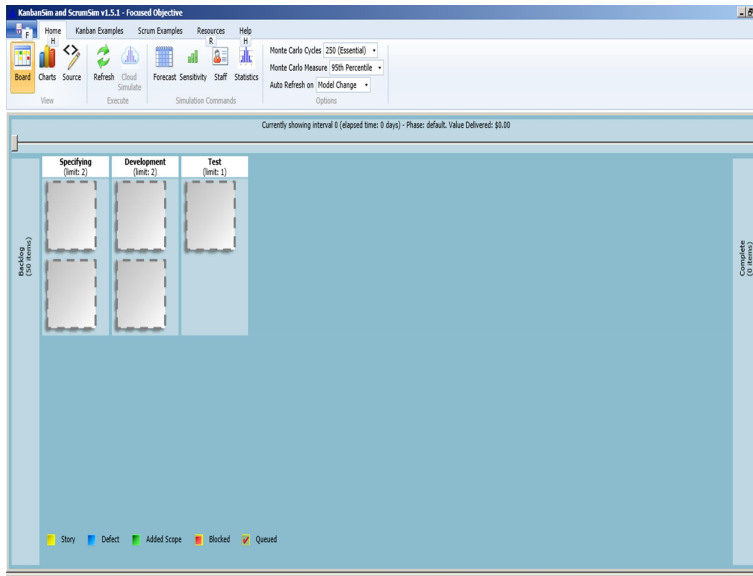


Figure 13.1: The WIP Limited Process in our Simulation

You will notice on this board that the Specifying column has a Work In Progress limit of two, the Development column has a Work In Progress limit of two, and the Test column has a Work In Progress limit of one. Let's further suppose that for this process we will be working through a backlog of 50 items. In this experiment, we are going to size all of our items such that each one takes exactly 10 days to go through each column. That is, every item in the backlog that flows through this board will take exactly 10 days in Specifying, 10 days in

Development and 10 days in Test. We are also going to introduce two Classes of Service: Standard and Expedite. I will explain the pull order rules for each of these as we go through the simulation. Lastly, you should know that in this experiment there will be no blocking events or added scope. We will start the simulation with 50 items in the backlog and we will finish the simulation with 50 items in Done. All items will be allowed to flow through unmolested.

Or will they?

You will notice from the design of the board in Figure 13.1 that, at the end of the 20th day, two items will have completed in the Dev column but there will only be space to pull one of those items into the Test column. As you are about to see, the simple decision around which of those two to pull will have a dramatic effect on the predictability of your system.

For the first run we are going to assign only a Standard CoS for work items on the board. Further, we are going to define a strict "First-In, First-Out" (FIFO) pull order policy for those Standard class items. That is, the decision around what item should be pulled next will be based solely on which item entered the board first.

Before I show you the results, I would like you to try to guess what the expected Cycle Time for our items will be. (Note: for these simulations I am going to consider the "expected value" for the Cycle Times to be the 85th percentile.) If you are ready with your guess then read on.

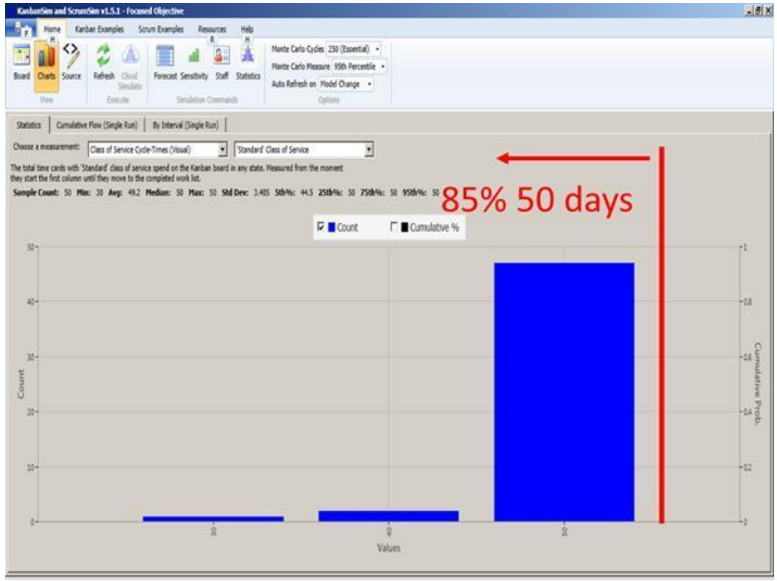


Figure 13.2: Strict FIFO Pull Order with No Expedites

Figure 13.2 shows a Histogram of the Cycle Time results. You can see that after running this simulation, the 85th percentile for our Cycle Times is 50 days. In other words, 85% of our items finished in 50 days or less. Also, as you look at the distribution of Cycle Times in the Histogram above, you will see that we have a fairly predictable system—there is not much variability going on here. But let's see what happens when we begin to tweak some things.

In this next round, we are going to replace our strict FIFO pull order policy with a policy that says that we will choose which item to pull next completely at random. One way it may help you to think about this is when two items are finished in the Development column, we are essentially going to flip a coin to see which one we should pull next into the Test column.

Any guess now as to what this new policy is going

to do to our expected Cycle Time? To variability? To predictability?

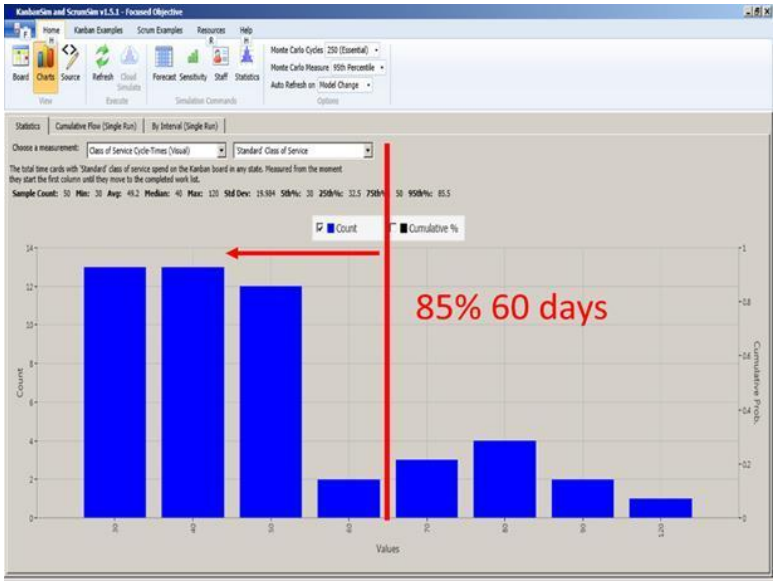


Figure 13.3: Random Pull Order with no Expedites

In this case (Figure 13.3), the simple switch from FIFO queuing to random queuing has increased our 85th percentile Cycle Time from 50 days to 60 days—that is an increase of 20%! Did you expect that such a minor policy change would have such a big Cycle Time impact? You can also see that the corresponding distribution (shown in Figure 13.3) is much more spread out reflecting the increased variability of our random decision making.

Things get interesting when we start to add in some expedites. Let's look at that next.

We are now going to go back to the pull policy where our Standard class items are going to be pulled through in a strict FIFO queuing order. The twist we are going introduce, though, is that we are now going to include

an Expedite Class of Service for some of the items on our board. In this round we are going to choose exactly one item on the board at a time to have an Expedite Class of Service. When one expedited item finishes, another one will be immediately introduced. These Expedites will be allowed to violate WIP limits in every column. Further, whenever both an Expedite and Standard class item finish simultaneously, then the Expedite item will always be given preference over the Standard item when it comes to deciding which one to pull next.

Standard questions apply before proceeding: any thoughts on Cycle Time impact? Variability? Predictability?

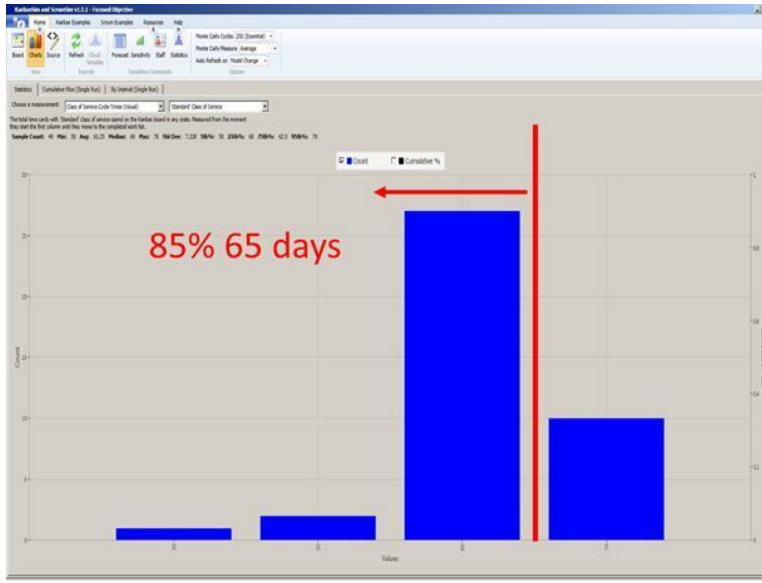


Figure 13.4: FIFO Pull Order with Always One Expedite on the Board

Any surprises here (Figure 13.4)? Compared to the previous case (the random pulling case), Expected Cycle Time has increased five days from 60 days to 65 days.

You can see the Histogram (Figure 13.4) has become much more compact, but there is still a wider spread than when compared to our baseline case (the strict FIFO/no expedites case), and, as I just mentioned, overall Cycle Times are longer. Did you expect this to be the worst case yet from a Cycle Time perspective? You can see that this is only marginally worse than the random queuing round—but it is still worse. That is an interesting point that bears a little more emphasis. In this context, introducing an Expedite CoS is worse for predictability than simply pulling items at random. Hopefully you are getting a feel for just how disruptive expedites can be (if you were not convinced already).

But we are not done yet. There is still one permutation left to consider.

In this final experiment, we are going to change our pull policies for Standard class items back to random from FIFO. We are going to keep the rule of always having one Expedite item on the board. The pull policies for the Expedites remain the same: they can violate WIP limits and will always get pulled in preference to Standard class items.

Now what do you think will happen?

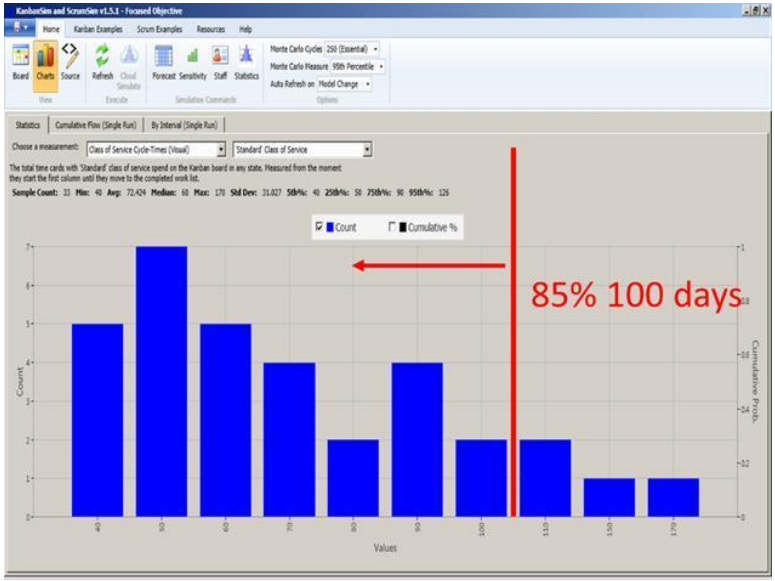


Figure 13.5: Random Pull Order with Always One Expedite on the Board

Expected Cycle Time in this scenario (Figure 13.5) has jumped to a simulation-worst 100 days! The spread of our data shown by the Histogram (Figure 13.5) is also worrying: Cycle Times range anywhere from 40 days to 170 days. If that is not variability, then I do not know what is. Remember, in the ideal system of the first case, the range of Cycle Times were 30 to 50 days.

Let's look at all these results side by side (Figure 13.6):

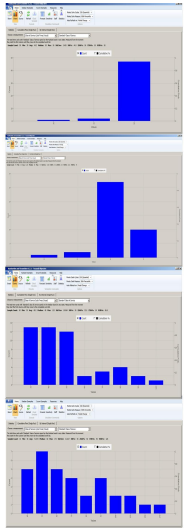


Figure 13.6: CoS Results Side by Side

I would like you to reflect on this result for a minute. Minor tweaks to process policies had a dramatic impact on simulation outcomes. Again, note that these policies were all things that were completely under our control! All of the variability in these scenarios was of our own doing. Worse still, my guess is that you have probably never even given any thought to some of these policies. Do you pay attention to how you decide what order to pull items through your process? Do you try to control or limit the number of Expedites on your board? Do you have any clue what a lack of these considerations is doing to your process predictability?

Obviously in the previous example I have controlled for story size. That is generally not possible (nor even required nor suggested) in the real world. Differences in story size are additional variability that is going to affect the predictability of the process and make these Histograms look even worse. That being the case, why

would we not try to mimic FIFO as closely as possible? Why would we not try to control pull policies that we can control?

The short answer is that we should. The longer answer is that in many contexts FIFO queuing may be impractical (leaving the business value dimension of pull decisions aside for a minute).

There are a couple of reasons for the impracticality of FIFO queuing. Think about a restaurant, for example. Patrons of restaurants do not flow through in a strict FIFO ordering. To illustrate, let's say a group is seated first at Table A. Then a different group is seated second at Table B. The group at Table B does not have to wait until the first group at Table A has finished before the second group is allowed to leave. That would just be silly. The groups are, however, usually seated in a First In First Served (FIFS) order. A (mostly) FIFS scheme is much more practical in the knowledge work context as well and usually is the best strategy from a predictability perspective.

Extending the restaurant example, let's say that a group of four people arrives to an establishment that is currently full and they need to wait for a table to open up in order to be seated. Let's further say that a group of two people arrives after the group of four and this second group needs to wait as well. If the first table to open up seats only two people, then it is reasonable that the group of two—who arrived second—would be seated first. This scenario happens all the time in knowledge work. Maybe a resource frees up and is ready to pull an item. But he does not have the expertise to work on the item that has been waiting the longest (which should be his first choice). From a practical perspective, it would be reasonable for him to pull the item that has been waiting the second longest (assuming, again, that

he has the right skills to work on that second one). But remember, even though this may be the best practical decision, it may not be the best predictable decision. In this scenario, what are some longer term improvements you could make for better predictability?

The point to all of this is that the further you stray from FIFO queuing, the less predictable you are. That is not to say that there are not practical reasons why you should forfeit FIFO. And by the way, arbitrary business value reasons and fictional Cost of Delay calculations do not fall into this practical category. But more on that a little later.

The most common objection I get when I explain why teams should adopt FIFO (or FIFS, or mostly FIFS) and dump expedites is that, “Expedites happen all the time in our context and we can’t not work on them”. They might go on to say that these expedites are unpredictable in size and number. Not only am I sympathetic to this argument, I acknowledge that this is the case for most teams at most companies.

Slack

So what is a team to do? Why, look at FedEx, of course.

Federal Express (FedEx) is an American shipping company that allows clients to send packages all over the world. For this example, though, let’s limit the scope of our discussion to just the continental United States. Suffice it to say that FedEx knows a thing or two about flow and predictability, and the company is worth studying.

When a prospective customer wishes to ship a package via FedEx that customer has several service options to choose from. She can choose to it send overnight,

2nd day air, and standard ground—to just name a few. All of these service options are going to result in different CoS that FedEx uses in order to make sure that packages get to their destinations within the agreed SLA. Think about this for a second. In the U.S. there are thousands of locations that FedEx will need to pick up packages from. On any given day, it is impossible for FedEx to proactively and deterministically know the exact number of packages, their respective requested CoS, their full dimensions, weight, etc. that will show up at any one of their locations. They could have one shop that is swamped with overnight requests while another location remains relatively quiet. The magnitude of this problem is almost beyond comprehension.

The incredible thing is, while I have not used FedEx a lot, I can tell you that every time I have needed to send a package overnight it has arrived at its location on time. How does FedEx do it?

There are a lot of strategies that FedEx employs, but the one that is probably most important is that at any given time FedEx has empty planes in the air. Yes, I said empty planes. That way, if a location gets overwhelmed, or if packages get left behind because a regularly scheduled plane was full then an empty plane is redirected (just-in-time it should be said) to the problem spot. At any given time FedEx has “spares in the air”!

A lot of people will tell you that Lean is all about waste elimination. But imagine if the FedEx CFO was hyper-focused on waste elimination for process improvement. Would that CFO ever allow empty planes to be in the air at any given time for any reason? Of course not. Flying empty planes means paying pilots’ salaries, it means burning jet fuel, it means additional maintenance and upkeep. Luckily for FedEx, they understand that Lean is not just about waste elimination, it is about

the effective, efficient, and predictable delivery of customer value. FedEx understands all too well the variability introduced by offering different CoS. They know that, in the face of that variability, if they want to deliver on their SLAs they must have spares in the air. They have to build slack into the system. Pretty much the only way to predictably deliver in the face of variability introduced by different CoS is to build slack into the system. There is just no way around it.

So let's get back to the "we have expedites that we cannot predict and that we have to work on" argument. Armed with this information about variability and slack, what do you think would happen if you went to your management and said, "if we want to predictably deliver on all of the expedites in our process (to say nothing of all of our other work), we need to have some team members that sit around, do nothing, and wait for the expedites to occur." You better have your resume updated because after you get laughed out of the room you will be looking for a new job.

"Ok, so you cannot have idle developers," so-called CoS experts will tell you, "then what you need to do is put a strict limit on the number of expedites that can be in your process at any given time." They will further advise you that the limit on expedites needs to be as small as possible—potentially as low as a WIP limit of one. Problem solved.

Not at all.

This advice ignores two further fundamental problems of CoS. For the first I will need another example. In my regular Kanban trainings I am a big fan of using Russell Healy's getKanban board game. I like the game not because it shows people how to do Kanban properly, but because it does a great job of highlighting many of the errors in the advice given by so many Kanban experts.

One of those errors is the advised use of an expedite lane on a Kanban Board (or CoS in general). Now in this game, there is a lane dedicated for expedited items, and, further, there is an explicit WIP limit of one for that lane. This is the exact implementation of the strategy that I just explained. So what is the problem? At the end of the game, I take the teams through an analysis of the data that they generated while they played the simulation (using all the techniques that have been outlined in the previous chapters). The data usually shows them that their standard items flow through the system in about ten or eleven days at the 85th percentile. And the spread in the Cycle Time data of standard items is usually between three and 15 days. The data for the expedited items' Cycle Time show that those items always take three days or less. You can see that the policies those teams used to attack the expedites made them eminently predictable. You will also note that those policies also contributed to the variability in the standard items, but that is not what is important here. What is important here is what happens when we project this to the real world. Imagine now that you are a product owner and you see that your requested item is given a standard CoS. That means that the team will request eleven days to complete it. But if your requested item is given an expedited CoS, then that item gets done in three days. What do you think is going to happen in the real world? That is right: everything becomes an expedite! Good luck trying to keep to the WIP of the expedited lane limited to one.

But that is not the only problem. Let's say that you work at an enlightened company and that they do agree that there will only be one expedited item in progress at any given time. It turns out even that is not enough! In the simulation example above, we limited our expedited

items to one but that still caused a sharp increase in Cycle Time variability. Why? Because there was always one expedited item in progress. If you are going to have an expedited lane, and you limit that lane's WIP to one, but there is always one item in it, then, I am sorry to say, you do not have an expedited process. You have a standard process that you are calling an expedited process, and you have a substandard process which is everything else.

For all practical purposes, introducing CoS is one of the worst things you can do to predictability.

But, you might argue, the real reason to introduce CoS is to maximize business value (for the purposes of this conversation, I am going to lump cost of delay and managing risk in with optimizing for business value). I might be persuaded by this argument if I believed that it was possible to accurately predetermine business value. If you could do that, then you really do not need to be reading this book because your life is easy. Obviously, if you have a priori knowledge of business value then you would just pull items in a way that maximizes that value. However, most companies I work with have no clue about upfront business value. And it is not due to inexperience, incompetence, or lack of trying. The reason most companies do not know about an item's business value upfront is because that value—in most cases—is impossible to predict. As value is only determined by our customers, an item's true value can only be known once put in the hands of the customer. Sure, most companies will require a business case before a project is started and this business case acts a proxy for business value. But, as you know, most business cases are anywhere

from pure works of fiction to out and out lies. Basing pull decisions on disingenuous arguments is suspect at best.

Let's put it another way. As I just mentioned, true business value can be determined only after delivery to the customer. Choices about what to work on and when, then, are really just you placing bets on what you think the customer will find valuable. By introducing CoS and by giving preference to some items in the process over other items means that you are gambling that the customer will find those preferred items more valuable. The problem is that when you lose that bet—and I guarantee you almost always will—you will have not only lost the bet on the expedited item, but you will have also lost the bet for every other item in progress that you skipped over.

Honestly, I am only mostly that cynical. I do believe that the business value of an item should be considered, but I believe it should only be considered at input queue replenishment time. After an item is placed in process then I believe the best long term strategy is to pull that item—and all other items—through the process as predictably as possible. After all, part of the business value equation is how long it will take to get an item done. If you cannot answer the question "how long?" then how much confidence can you really have in your business value calculation?

What about obvious high value expedites? Things like production being down that require all hands on deck? Or a new regulatory requirement that could result in massive fines for noncompliance? Obviously, those things will—and should—take precedence. But, just like the FedEx example, you should study the rate of occurrence for those items and adjust your process design accordingly. That will potentially mean lowering overall

process WIP. That will probably mean making sure free resources look to help out with other items in process before pulling in new items. And so on.

To come full circle on our discussion about Little's Law that was started in Chapter 3, I hope it is obvious for you to see how CoS represents a clear violation of the fourth assumption of Little's Law (and potentially the first and the third as well). The central thesis of this book is that every violation of a Little's Law assumption represents a reduction in overall process predictability. CoS represents an institutionalized violation of those assumptions. How could you ever expect to be predictable when using CoS as your standard process policy?

Conclusion

It is obvious that to solve the problem outlined at the beginning of this chapter, the TSA could simply hire more agents. At the very least you would want to have a minimum of one agent per queue. This intervention would potentially solve the problem—or it would go a long way to alleviating it. Note that in this case, however, CoS would be eliminated. If each queue had its own server, then there would be no need for CoS. Wouldn't it be great if all our problems could be solved by just adding more people? The reality is that most companies do not have the money to keep hiring. That being the case, we want to make sure that we are using the resources we do have as efficiently as possible. That means choosing pull policies that maximize our resources' effectiveness and eliminating policies that make it harder for those resources to do their jobs predictably.

Although it probably sounds like it, I am not saying that CoS is inherently evil or that all CoS implementa-

tions are incorrect. I am, however, coming at this from the perspective of predictability. With that consideration, what I am saying is that you need to consider all aspects of CoS before implementing those policies. By definition, CoS will introduce variability and unpredictability into your process. The unpredictability manifests itself—among other things—as Flow Debt (Chapter 9). The truth is that the only part of your process that is more predictable with CoS is the highest priority class. Overall, CoS will cause your process to actually take a predictability hit (see Figures 13.4 and 13.5). Are you really that confident that the upfront value decisions that you are making with CoS are worth more than all the negative implications?

The arguments swirling around out there about why to use CoS are very seductive. The people making those arguments are very persuasive. I am hoping I have at least given you something to think about before assuming you should start with CoS as a default.

For me, the better strategy is to consider an item's forecasted value at queue replenishment time. Then, once in process, pull that item through while paying attention to all the concepts outlined in this and the previous chapters.

You have to know what you are doing before you do it. Build your process. Operate it using the policies for predictability that I have outlined thus far. Measure it. And then make a determination if CoS can help. Chances are you will never need CoS.

Chances are you will never need Class of Service once you have a predictable process.

But what else do we need to consider ourselves pre-

dictable? I implied earlier that there are essentially to dimensions to being predictable:

1. Making sure your process behaves in a way it is expected to; and,
2. Making accurate predictions about the future.

Up until now we have mostly talked about point #1. It is time that we turn our attention to point #2.

Key Learnings and Takeaways

- Class of Service is the policy or set of policies around the order in which work items are pulled through a given process once those items are committed to (i.e., counted as Work In Progress).
- Class of Service only attaches at the point of commitment.
- Class of Service is different from queue replenishment.
- Assigning a work item a Class of Service is different from assigning a work item a type.
- Class of Service represents an institutionalized violation of some assumptions of Little's Law. This violation takes the form of Flow Debt which ultimately makes your process less predictable.
- The only way to predictably deliver using Class of Service is to build slack into the system.
- Instead of designing Class of Service into your process up front, consider other things you can do to eliminate or mitigate the need for them.
- Only introduce CoS after you have operated your process for a while and are confident that CoS is necessary. Still consider policies for CoS that mitigate their inevitable negative impact on flow.