CHAPTER 9

Kanban, Flow, and Constantly Improving

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

—David Anderson, Kanban

Kanban is a method for process improvement used by agile teams. Teams that use Kanban start by understanding the way that they currently build software, and make improvements to it over time. Like Scrum and XP, Kanban requires a mindset. Specifically, it requires the lean thinking mindset. We've already learned that Lean is the name for a mindset, with values and principles. Teams that use Kanban start by applying the Lean mindset to the way they work. This provides a solid foundation which, combined with the Kanban method, gives them the means to improve their process. When teams use Kanban to improve, they focus on eliminating the waste from their process (including muda, mura, and muri—the wastes of unevenness, overburdening, and futility that we learned about in Chapter 8).

Kanban is a manufacturing term, adapted for software development by David Anderson. Here's how he describes its relationship to Lean in his 2010 book, *Kanban*: "The Kanban Method introduces a complex adaptive system that is intended to catalyze a Lean outcome within an organization." There are teams that apply Lean and lean thinking to software development without using Kanban, but Kanban is by far the most common—and, to many agile practitioners, the most effective—way to bring lean thinking into your organization.

Kanban has a different focus from agile methodologies like Scrum and XP. Scrum primarily focuses on project management: the scope of the work that will be done, when that work will be delivered, and whether the outcome of that work meets the needs of the users and stakeholders. The focus of XP is software development. The XP values and practices are built around creating an environment conducive to development,

and developing programmer habits that help them design and build code that's simple and easy to change.

Kanban is about *helping a team improve the way that they build software*. A team that uses Kanban has a clear picture of what actions they use to build software, how they interact with the rest of the company, where they run into waste caused by inefficiency and unevenness, and how to improve over time by removing the root cause of that waste. When a team improves the way that they build software, it's traditionally called *process improvement*. Kanban is a good example of applying agile ideas (like the last responsible moment) to create a process improvement method that is straightforward and easy for teams to adopt.

Kanban has practices that give you a way to stabilize and improve your system for building software. The latest set of core practices can be found on the Kanban Yahoo! group:

> First follow the foundational principles:
>
> - Start with what you do now
> - Agree to pursue incremental, evolutionary change
> - Initially, respect current roles, responsibilities & job titles
>
> Then adopt the core practices:
>
> - Visualize
> - Limit WIP
> - Manage Flow
> - Make Process Policies Explicit
> - Implement Feedback Loops
> - Improve Collaboratively, Evolve Experimentally (using models/scientific method)
>
> It is not expected that implementations adopt all six practices initially. Partial implementations are referred to as "shallow" with the expectation that they gradually increase in depth as more practices are adopted and implemented with greater capability.

In this chapter, you'll learn about Kanban, its principles and how they relate to Lean, and its practices. You'll learn how its emphasis on flow and queuing theory can help your team put lean thinking in practice. And you'll learn about how Kanban can help your team to create a culture of continuous improvement.

Narrative: a team working on a mobile phone camera app at a company that was bought by a large Internet conglomerate

Catherine – a developer

Timothy – another developer

Dan – their boss

Act II: Playing Catch-Up

Catherine and Timothy were getting sick of Dan. They understood that there were deadlines, and that the work they were doing was important. And they even recognized the kind of pressure that he was under to deliver. But it seemed like every little project, no matter how small, ended up in "Intensive Care Unit mode," which was what Dan called it when he started to micromanage them.

Their current project was no different. They were working on a new feature for their phone camera app that turns their friends' faces into old timey "Wanted" posters. It was supposed to be a simple project that integrated their camera app with their parent company's social networking system. And as usual, they found a bunch of bugs during testing that would cause them to miss the deadline.

"I'm pretty sure we wouldn't have any trouble getting this done on time if he would just let us do our jobs," said Timothy.

"I know what you mean," said Catherine. "It's like he's meddling in every little thing I do. Speaking of meddling, it's after seven and starting to get dark out. We're late for his evening status meeting."

It was "crunch time," which is what Dan called it when there was a deadline they weren't going to make. That meant it was always crunch time.

Catherine and Timothy walked over to Dan's office and sat down. Their other teammates were already there, and nobody looked happy. Dan was already in full-on micromanagement mode.

"Look, we're in intensive care on at least three projects. Tim, Cathy, we'll start with yours."

Catherine said, "We're making progress—"

Dan cut her off. "You're not making progress. This project is falling apart. I've given you enough time to do it your way, now we need to do it the *right* way."

Timothy replied, "But there was that bug that had to be fixed."

"It seems like there's always that bug. You guys just can't seem to get these projects done on time. I'm going to have to jump in here, because you're not giving this project enough urgency."

"Hold on," said Catherine. "It's not a coincidence that we always end up here. There's a lot of waste that happens along the way."

"What do you mean, waste?" asked Dan. "That sounds really negative. We need to stay positive if we want to be successful here." (Dan always talked about staying positive, even when he was berating people who worked for him.)

"Well, like the way it takes weeks to get everyone to agree on any change to a user interface. About halfway through that discussion you tell us to get started on it, and it seems like we spend more time changing that code than we spent writing it in the first place."

Timothy said, "Right. Or the way we're always surprised that the QA team finds bugs. Somehow it happens every time, but we never give ourselves enough time to fix them.

Dan started looking angry. "Look, that's just the way software projects work. Stop pointing fingers and blaming people. It's my fault, or the QA team's fault."

Catherine was fed up. "Look, Dan, that's enough."

Everyone looked at her. She almost never raised her voice like that.

"We're not blaming anyone. There are problems that seem to happen over and over again. We have these meetings twice a day, and we sound like a broken record. We need to talk about problems that happen over and over again. They're always the same problems, and we act surprised every time."

Dan seemed a little taken aback that she was yelling back at him. He stood up, stared her down for a minute, then sat back down in his chair again. "You know what? All of those things are great to think about next time. Right now, we're going to just have to put a little more time and urgency into it. This is crunch time."

The Principles of Kanban

Let's take a closer look at Kanban's foundational principles:

- Start with what you do now
- Agree to pursue incremental, evolutionary change

- Initially, respect current roles, responsibilities, and job titles

That first principle—starting with what you do now—has a different focus from everything you've read so far in this book.

We've spent a lot of time comparing agile methodologies to traditional waterfall projects. Scrum, for example, gives you a complete system for managing and delivering your projects. If you want to adopt Scrum, you need to create new roles (Scrum Master and Product Owner), and you need to give your team new activities (sprint planning, Daily Scrum meetings, task boards). This is necessary for adopting Scrum because it's a system for managing your project and delivering software.

Kanban is *not* a system for managing projects. It's a method for improving your process: the steps that your team takes to build and deliver software. Before you can improve anything, you need a starting point, and the starting point for Kanban is what you do today.

Find a Starting Point and Evolve Experimentally from There

The thing about habitual problems is that they're habitual.

When your team does something on a project that will eventually lead to bugs and missed deadlines, it doesn't feel like a mistake at the time. You can do all the root-cause analysis that you want after the fact; faced with the same choice, the team will probably still make the same decision. That's how people are.

For example, let's say that a programming team always finds themselves delivering software to their users, only to repeatedly have awkward meetings where users can't find the features they think they were promised. Now, it's certainly possible that these developers are incredibly forgetful, and that they always forget one or two of the features that they discussed with the users. But it's more likely that they have a recurring problem with how they gather their requirements or communicate them to the users.

The goal of process improvement is to find recurring problems, figure out what those problems have in common, and come up with tools to fix them.

The key here is the second part of that sentence: figuring out what those problems have in common. If you just assume, for example, that a developer simply can't remember all of the things the users asked for, or that the users constantly change their minds, then you've effectively decided that the problems are unfixable. But if you assume that there's a real root cause that's happening over and over again, then you stand a chance of finding and fixing it.

That's where Kanban starts: taking a look at how you work today, and treating it as a set of changeable, repeatable steps. Kanban teams call steps or rules that they always follow **policies**. This essentially boils down to a team recognizing their habits, seeing

what steps they take every time they build software, and writing all of those things down.

Writing down the rules that a team follows can sometimes be tricky, because it's easy to fall into the trap of judging a team—or an individual team member—on the results of a project: if the project is successful, everyone on the team must be good at their jobs; if the project fails, they must be incompetent. This is unfair, because it assumes that everything in the project is within the control of the team. Lean thinking helps get past this by telling us to see the whole, which in this case means *seeing that there's a bigger system in place*.

That's worth repeating: *every team has a system for building software*. This system may be chaotic. It may change frequently. It may exist mainly inside the heads of the team members, who never actually discussed a larger system that they follow. For teams that follow a methodology like Scrum, the system is codified and understood by everyone. But many teams follow a system that exists mainly as "tribal" knowledge: we always start a project by talking to these particular customer representatives, or building that sort of schedule, or creating story cards, or having programmers jump in and immediately start to code after a quick meeting with a manager.

This is the system that Kanban starts with. The team already has a way to run their project. Kanban just asks them to understand that system. That's what it means to start with what you do now. The goal of Kanban is to make small improvements to that system. That's what it means to pursue incremental, evolutionary change—and why Kanban has the practice **improve collaboratively, evolve experimentally**. In lean thinking, part of seeing the whole is taking measurements, and measurements are at the core of experimentation and the scientific method. A Kanban team will start with their system for building software, and take measurements to get an objective understanding of it. Then they'll make specific changes *as a team*—later in this chapter, you'll learn exactly how those changes work—and check their measurements to see if those changes have the desired effect.

The Lean value of amplifying learning is also an important part of evolving the system that your team uses to build software. Throughout this book you've learned about feedback loops. When you collaborate to measure the system and evolve experimentally, those feedback loops become a very important tool for gathering information and feeding it back into the system; the Kanban practice of **implementing feedback loops** should make sense to you, and should help you to see how Kanban and Lean are closely linked.

Amplifying learning also factors into the Kanban principle of initially respecting current roles and responsibilities. For example, say that a team always starts each project with a meeting between a project manager, a business analyst, and a programmer. They may not have written down a rule for what goes on in that meeting, but you probably have a good idea of what goes on in it just from reading those job titles.

That's one reason why Kanban respects current roles, responsibilities, and job titles—because they're an important part of the system.

A common theme between all of these principles is that Kanban only works for a team when they take the time to understand their own system for building software. If there was one right way to build software, everyone would just use it. But we started this book by saying back in Chapter 2 that there is no silver bullet—there's no single set of "best" practices that will guarantee that a team builds software right every time. Even the same team, using the same practices, can have success with one project but fail miserably in the next one. This is why Kanban starts with understanding the current system for running the project: once you see the whole system, Kanban gives you practices to improve it.

Wait a minute. So Kanban doesn't tell me how to run my project?

No, it doesn't.[1] Kanban asks you to start by understanding how you currently run projects. That could be Scrum, XP, better-than-not-doing-it Scrum, an effective waterfall process, an ineffective waterfall process, or even a haphazard, "fly-by-the-seat-of-your-pants" way of doing things. Once you figure out how your team currently builds software, Kanban gives you practices to help improve it.

So if you already have a way of building software, why do you need Kanban?

Most teams manage to deliver *something*. How do you know if the team is wasting a lot of time or effort? Are they doing work that delivers a lot of value? Or are they being asked to work in a way that habitually causes problems or makes it hard to deliver valuable software?

When we have a system in place—no matter what that system looks like—it doesn't occur to most of us to question it. Even if you're using Scrum or XP, you could be wasting a lot of time and effort without even knowing it. Habitual problems are very difficult to spot. Everyone can be following the rules and doing the right things. But just like behavior can emerge from a system, waste can emerge from the way many different people work together.

We saw an example of this in Chapter 8, where a team finds themselves with a very large lead time, even though everyone is constantly working and nobody is slacking off or intentionally waiting to do work. But even though everyone is constantly working, to the person who requested the software, it looks like there are very big delays—

1 Kanban isn't a project management method, but that certainly doesn't mean that Kanban isn't for project managers! In fact, David Anderson posted a series of blog posts that explain exactly how project managers fit into Kanban, and offers a "Project Management with Kanban" class through the Lean Kanban University. We recommend reading his posts to learn more about this.

and nobody on the team even noticed, because they feel like they're doing everything that they can do to get the work done as quickly as possible.

This is the kind of problem that Kanban solves.

Stories Go into the System; Code Comes Out

> Systems thinking looks at organizations as systems; it analyzes how the parts of an organization interrelate and how the organization as a whole performs over time.
>
> —Tom and Mary Poppendieck, *Lean Software Development*

The first step in improving a system is recognizing that it exists. This is the idea behind the Lean principle *see the whole*. When you see the whole, you stop thinking of the team as making a series of individual, disconnected decisions, and start to think of them as following a system. In Lean, this is called **systems thinking**.

Every system takes inputs and turns them into outputs. What does a Scrum team look like from a systems thinking perspective? You can think of Scrum as a system that takes project backlog items as its input and produces code as its output. Many Scrum teams have a project backlog that consists entirely of stories; these teams can almost think of themselves as "machines" that turn stories into code.

Clearly these are teams, not machines, and we certainly don't want to fall into the habit of thinking of people as machines or cogs. But there's value in thinking about the work that you do as part of a greater system. If you apply systems thinking to your Scrum team, it makes it much easier to see when you're doing work that doesn't directly (or even indirectly) help turn stories into code. And by recognizing that all of Scrum is a system, you can understand how it works better and make improvements to it. This is the kind of thinking that leads to improvements such as Jeff Sutherland's changes to the questions in the Daily Scrum that you learned about in Chapter 5. That's a good example of systems thinking applied to Scrum that leads to incremental, evolutionary improvements.

Kanban asks you to start by understanding the system that you and your team use. And even if you don't follow a methodology with a name, you can still apply systems thinking to your own team to figure out how you work.

Every software team follows a system, whether they know it or not

It's easy to see how a team that follows Scrum is using a system. But what if your team doesn't have anything like that? Maybe you just dive in and start working on software. It certainly feels like you don't run your projects the same way every time. So you don't really have a system, right?

The funny thing about people—especially on teams—is that we *always* follow rules. They may not be written down, and we often make them up if they don't exist. But humans intuit rules all the time, and once a rule gets into our heads we have a tough

time shaking it. And even if you don't think you follow rules, when a new person comes onto your team, you definitely recognize when that person breaks an unwritten rule.

And Lean even gives us a tool to take unwritten rules and turn them into a system: a value stream map. When you take an MMF (that's the minimal marketable feature we learned about in Chapter 8—like a story, requirement, or user request, for example) and draw out the value stream map that it followed on its path to becoming code, you've written down a description of a path through your system.

When you work on a team that follows many unwritten rules, it's likely that one MMF follows a very different path from another. But because humans naturally intuit and follow rules, it's also pretty likely that you can map out a small number of value streams that cover the majority of the MMFs that your team turns into code. If you can do this, then you can build a very accurate description of the system that you and your team follow. The first step in that system is deciding which value stream the MMF will flow down.

It's worthwhile to have a system that works the same way for everyone, even if there are many different possible paths that an MMF can take. Once you understand how the system works—in other words, once you see the whole—you can start to make real decisions about which paths are wasteful, and make incremental, evolutionary changes to the way you and your team build the code.

Kanban is not a software methodology or a project management system

One of the most common pitfalls that people run into when learning about Kanban is to attempt to treat it as a methodology for building software. It isn't. Kanban is a method for process improvement. It helps you understand the methodology that you use and find ways to make it better.

Take a minute and just flip forward a few pages in this chapter. You'll see pictures that look like task boards. *These are not task boards.* They're called kanban boards. The way that you know they're not task boards is that *they don't have tasks on them*. They have **work items**. A work item is a single, self-contained unit of work that can be tracked through the entire system. It's typically larger than an MMF, requirement, user story, or other individual scope item.

One difference between a task board and a kanban board is that while tasks flow across a task board, *work items are not tasks.* The tasks are what the people do to move the work items through the system. In other words, the tasks are the "cogs" of the machine that push the work item through. This is why you can use systems thinking to understand your software workflow without dehumanizing people by thinking of them as cogs in a machine. It's the tasks that are the cogs; the people are still unique individuals with their own personalities, wants, and motivations.

The columns on the kanban board may seem similar to steps in a value stream; however, many Kanban practitioners distinguish value stream maps from kanban boards. They will map the state of work items in the workflow separately from the value stream, something that they call **workflow mapping**. The difference is that value stream mapping is a Lean thinking tool to help you understand the system that you work in; workflow mapping is how the Kanban method determines the actual steps that each work item goes through. (You'll learn more about how to build a kanban board later on in the chapter.)

Here's an example of how task boards do one thing, while kanban boards do another. Scrum is focused on helping teams self-organize and meet their collective commitments. By the time the task board is in play, the team has selected backlog items—work items—to include in the sprint, and broken them down into tasks. As the tasks flow across the task board, the work items start to move from the "To Do" column to the "Done" column. To the team, it feels like they're making progress.

A typical kanban board only shows those larger work items, not the individual tasks. And while the task board only "sees" the work items when they're To Do, In Progress, or Done, the kanban board will have a bigger picture. Where do the work items come from? How does the Product Owner know what work items to put in the project backlog, and how to prioritize them? After the team completes the work item, is it tracked by a production team to make sure that it's been deployed properly? The work item has a larger lifecycle that extends beyond the team that's building it. The kanban board will have columns for the steps that a work item goes through before and after the Scrum team gets their hands on it.

What's more, the kanban board can help show problems that never appear on the task board. Many Scrum teams are very good at building the software that they're asked to build, but still find themselves with disappointed users. Maybe the work items they're building aren't the right ones that satisfy the customers' needs. Or maybe there's a very long lag before or after the team works on the work items, maybe due to a lengthy review or deployment process. Even though these problems are completely out of the team's control, they'll often be blamed for them. The kanban board will help with this.

So while Kanban is not a system for project management, it has a very important relationship with the project management used by the team. Kanban is intended to improve and change the process in use on the project and that this can and will affect how the project is managed. Typically, Kanban is used to improve predictability of flow, and this will affect planning and scheduling on the project. Extensive use of

Kanban and its metrics is likely to have a significant knock-on effect on the method of project management.[2]

In the next part of this chapter, you'll learn about the practices of Kanban, how to build a kanban board, and how to use to make incremental, evolutionary improvements to the *whole* system.

Improving Your Process with Kanban

We already know that Kanban is an agile method that's focused on process improvement, and is based on Lean values and lean thinking. Kanban was formulated by David Anderson, who first started experimenting with the ideas of Lean while working at Microsoft and Corbis. And like much of Lean, the name "kanban" originates with ideas developed at car manufacturers in Japan. But what's so agile about Kanban? How is it different than traditional process improvement?

Software teams have been doing process improvement for about as long as people have been building software. In an ideal world, process improvement works very well: the team gets senior sponsorship for the effort, takes measurements, identifies problems, implements improvements, and then starts over again to identify more improvements. Eventually, the improvements help the entire organization first let them make their process repeatable, then managed, and finally bring it under statistical control. There have been many companies that have reported extensive success doing this.

If you're a developer who has lived through a typical process improvement effort, you're probably ready to put this book down in frustration after reading that.

The term "process improvement" often conjures up Dilbert-like images of endless committees and binders full of process documentation, because typical process improvement is very different from the ideal. In a typical process improvement effort, a large company decides that their programmers are not producing software efficiently enough (or that they need a process certification for contracting or marketing purposes), so they hire a consulting company to spend a lot of time (and money) drawing many flowcharts of the existing and desired development processes, and training the teams to use the new ones. Then the teams spend about 10 minutes trying out one of the new processes, find that it's unnatural, awkward, and difficult to work with, and stop using it. But because senior managers sponsored the whole process improvement effort, politically the teams have to look like they're using it, so they fill out whatever paperwork the new process requires (like scope documents and statements of work) and produce the compulsory artifacts (like code review meeting

2 Thanks to David Anderson for helping us with the wording of this paragraph.

minutes and test reports that are mostly empty and boilerplate). For every successful process improvement effort—and yes, there are a few—there are many neutral or failed efforts whose main by-product is a deep dislike of the term "process improvement."

There's one big difference between Kanban and traditional process improvement. In traditional process improvement, the decisions are typically sponsored by senior managers, made by a committee (like a software engineering process group), and handed down to the teams through their bosses. In Kanban, the improvement is *left in the hands of the team*, and this is one reason that agile teams have found success with Kanban. The team members themselves find the problems with their workflow, suggest their own improvements, measure the results, and hold themselves accountable to their own standards.

So how does Kanban help a team improve their own process?

Visualize the Workflow

The first step in improving a process is understanding how the team currently works, and that's what the Kanban practice **visualize** is about. This sounds simple, but it's much more challenging than it sounds—and it's where many traditional process improvement efforts go wrong.

Imagine that you're a programmer, and your boss comes to you and asks, "How do you build software?" Your job is to write down how you do your work. So you start up Visio (or Omnigraffle, or another diagramming application) and start building a flowchart that shows all of the things that you do every day. Then you realize that while everyone always talks about holding code reviews (or testing your code before you commit, etc.), you don't actually do it every time. But you think that it would be a good idea, and you definitely do it *sometimes*, so you add it to your diagram. This is human nature. It's easy to justify the addition to yourself—if it's a good idea, then writing it down may help make sure that you do it all the time.

This will thoroughly screw up a process improvement effort.

One reason is that it masks a real problem. If the step that you've added to your diagram is a good idea, the fact that it now appears on a diagram makes it seem like you're already doing it. Nobody will think to ask, "Why aren't we doing it?" Why would they? You're doing it already! What if there's a reason that you're not doing it every time—say, code reviews always get cancelled because only senior team members are allowed to do code reviews and they're always busy? You'll never discover that and try to fix the underlying problem, because everyone will look at the diagram, see that code reviews are always happening, and focus their improvement effort elsewhere.

In Kanban, visualizing means writing down exactly what the team does, warts and all, without embellishing it. This is part of lean thinking: a Kanban team takes the Lean principle of *see the whole* very seriously. When the team has the right mindset, it *just feels wrong* to tinker with the workflow while you're trying to visualize it, because that would interfere with seeing the whole. The value of *decide as late as possible* is also important here: you don't have all of the information about how you build software yet, so there's a later responsible moment to make decisions about how you'll change it.

Like other agile methodologies, doing the practices of Kanban helps you get into the Lean mindset and adopt lean thinking. The better you accurately, objectively visualize the workflow, the better you embed the values of *see the whole* and *decide as late as possible* into your own thinking.

So how do teams visualize the workflow?

Use a kanban board to visualize the workflow

A **kanban board** is a tool that teams use to visualize their workflow. (The K in the methodology name *Kanban* is typically uppercase; the *k* in *kanban board* is usually lowercase.) A kanban board looks a lot like a Scrum task board: it typically consists of columns drawn on a whiteboard, with sticky notes stuck in each column. (It's more common to find sticky notes stuck to kanban boards than it is to find index cards.)

There are three very important differences between a task board and a kanban board. You already learned about the first difference: that kanban boards only have stories, and do not show tasks. Another difference is that columns in kanban boards usually vary from team to team. Finally, kanban boards can set limits on the amount of work in a column. We'll talk about those limits later on; for now, let's concentrate on the columns themselves, and how different teams using Kanban will often have different columns in their kanban boards. One team's board might have familiar To Do, In Progress, and Done columns. But another team's board could have entirely different columns.

When a team wants to adopt Kanban, the first thing that they do is visualize the workflow by creating a kanban board. For example, one of the first kanban boards in David Anderson's book, *Kanban*, has these columns: Input Queue, Analysis (In Prog), Analysis (Done), Dev Ready, Development (In Prog), Development (Done), Build Ready, Test, and Release Ready. This board would be used by a team that follows a process where each feature goes through analysis, development, build, and test. So they might start off with a kanban board like the one shown in Figure 9-1, with sticky notes in the columns representing the work items flowing through the system.

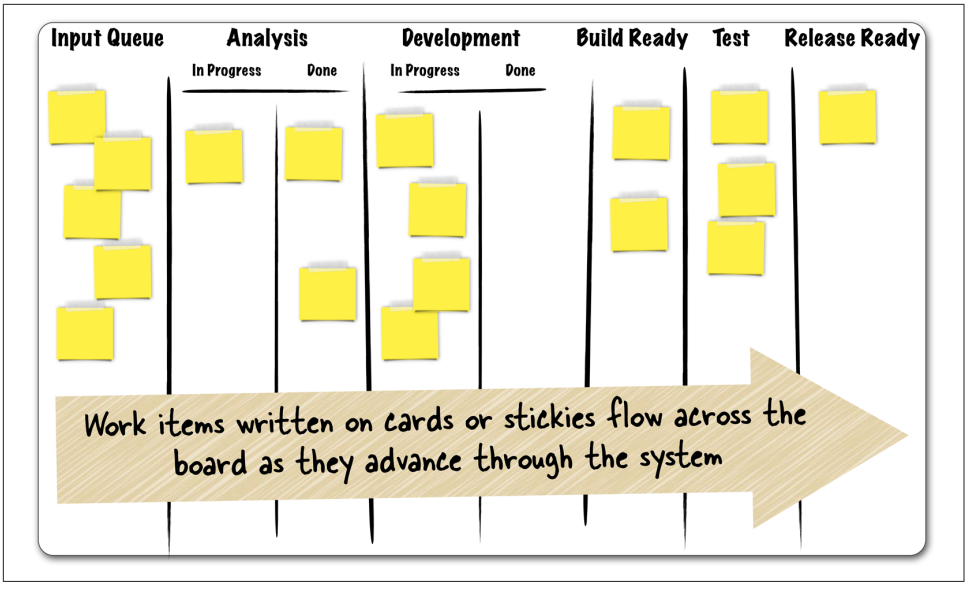

Figure 9-1. An example of how work items written on sticky notes flow across a kanban board. David Anderson used a board with these columns in his book, Kanban, but kanban boards have different columns depending on how the people on the team actually do their work.

The team would then use the kanban board in a way that's similar to how a Scrum team uses a task board. The Kanban team holds a meeting (usually daily) called "walking the board," in which they discuss the state of each item on the board. The board should already reflect the current state of the system: every item that completes a current step should already have been advanced to the next column by having its sticky note pulled up and moved into the next column. But if that hasn't been done yet, the team will make sure that the board is up to date during the meeting.

It is important to understand that a kanban board visualizes the underlying workflow and process in use. Any examples given here or in other texts (such as David Anderson's *Kanban*) are merely examples from real contexts. In general, you should never copy another kanban board; rather, you should develop your own by studying your own workflow and visualizing it. Copying an existing process definition out of context would be the antithesis of the Kanban Method's evolutionary approach to change. If the method requires you to start with what you do now, then you should not start by copying something someone else is doing.[3]

3 Thanks to David Anderson for helping us out with the wording here.

Let's go back to our example from Chapter 8, in which a team was trying to cope with very long lead times and disappointed customers. If you flip back to the initial description of the team, it's basically a summary of the initial workflow that the project manager described to the boss. Here's a quick recap:

1. Team gets a feature request from a user
2. Project manager schedules features for the next release
3. Team builds the feature
4. Team tests the feature
5. Project manager verifies that the tests pass
6. The feature is done and included in the next release

The paragraphs in Chapter 8 are one way to communicate this workflow, and this numbered list is another way, but a visualization is a much more effective tool to do that. Figure 9-2 shows what the project manager's "happy path" version of the workflow looks like on a kanban board.

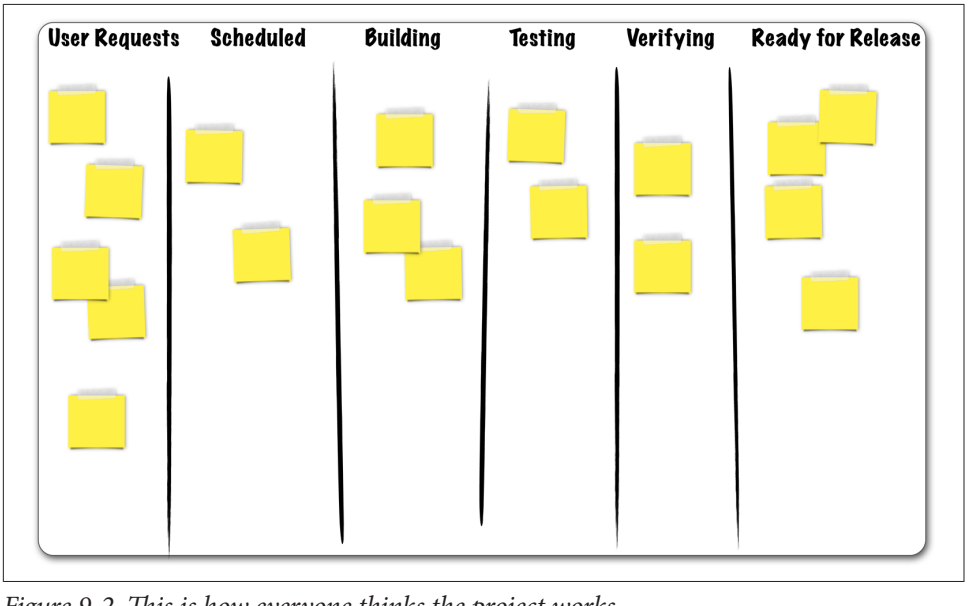

Figure 9-2. This is how everyone thinks the project works.

But that's not what's happening in real life. In Chapter 8, the team used Five Whys to learn more about their workflow. Afterward, it looked more like this:

1. Team gets a feature request from a user
2. Project manager schedules features for the next three-month release

3. Team builds the feature
4. Team tests the feature
5. Project manager verifies that the tests pass
6. *Project manager schedules a demo with senior management*
7. *If senior management wants the team to make changes to the feature, the project manager does an impact analysis on the changes, and the feature moves back to step 1—if not, it moves on to step 8*
8. The feature is done and included in the next release

Now we know that there's an extra step, where senior managers can optionally make changes to features and bump them to future releases after the team thought they were done.

We'll modify the kanban board to represent this better understanding by adding a column called "Manager Review" for those features that are waiting for the demo with the senior manager.

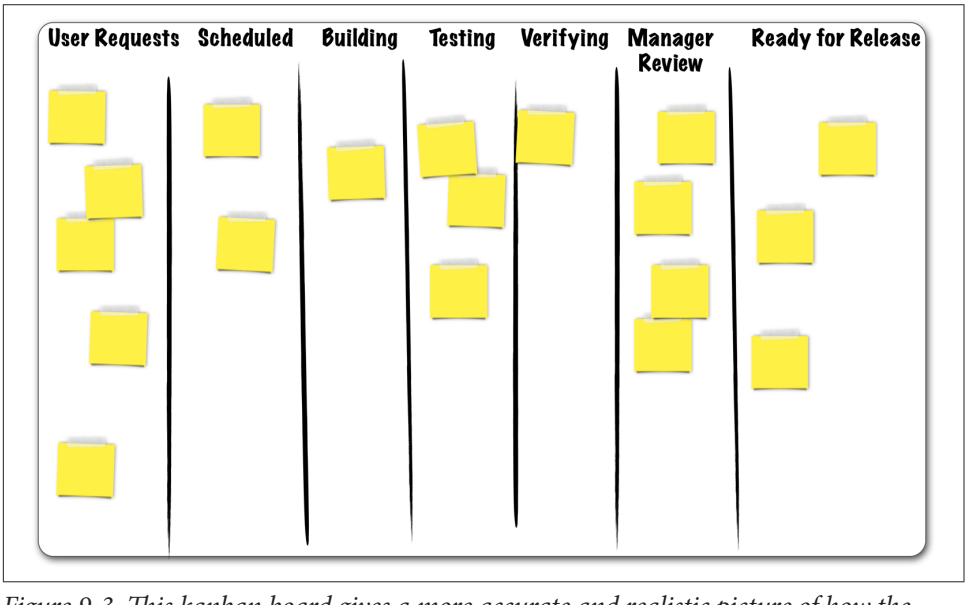

Figure 9-3. This kanban board gives a more accurate and realistic picture of how the project is run.

Now we have a more accurate visualization of the team's workflow. If we keep the kanban board going over the course of a release, it becomes very obvious where the problem is. The work items pile up in the "Manager Review" column, and keep piling up until the end of the release, as shown in Figure 9-4.

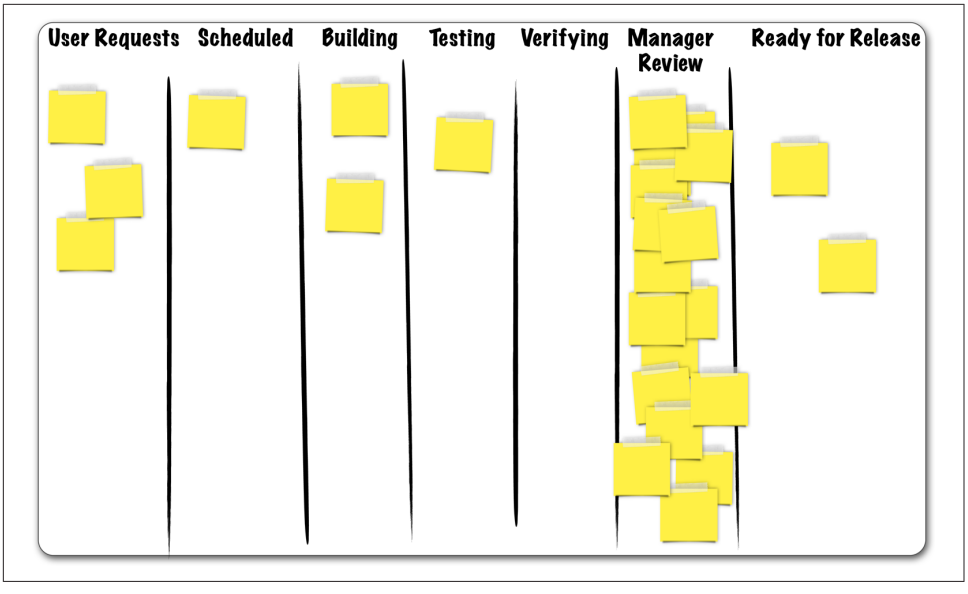

Figure 9-4. When you use a kanban board to visualize the workflow, problems caused by unevenness (mura) become easier to spot.

But what about the work items that were bumped to a future release to make room for the managers' changes? We especially care about those work items, because they're the ones that were causing some users to switch to the competitors. Sometimes work for those work items has already started, and needs to keep going even when they're bumped. When work items are bumped after the manager review, they end up going right back to the beginning of the process. Let's make sure these are represented on the kanban board—we'll add a column at the beginning of the board called "Bumped by Managers" and move those stickies back there (we put a small dot on each of the bumped stickies to make them easier to spot as they flow across the board a second time).

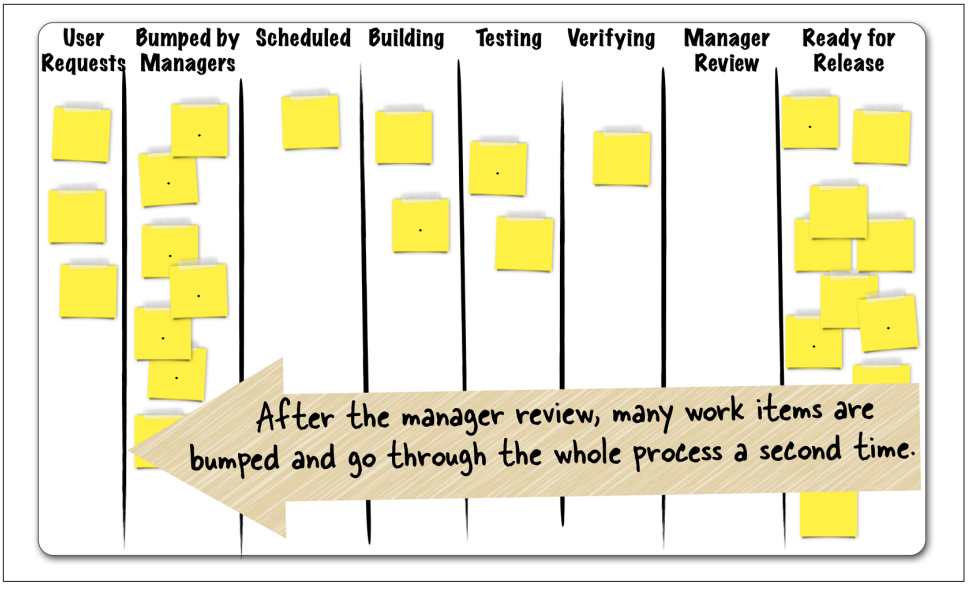

Figure 9-5. The kanban board makes the waste more obvious when you can see stickies go through the workflow more than once.

This is a pretty good visualization of the process that this team is following. Now we can see exactly what's gone wrong with this project, and why the lead time keeps getting worse. Some people on the team probably had a pretty good idea that this was going on, but now it's made clear to anyone who looks at the board. And more importantly, it becomes objective and explicit evidence that the way the senior managers review the features is a major cause of the lead time problem.

Limit Work in Progress

> We would never run the servers in our computer rooms at full utilization—why haven't we learned that lesson in software development?
>
> —Mary and Tom Poppendieck, *Lean Software Development: An Agile Toolkit*

A team can only do so much work at a time. We learned this with both Scrum and XP, and it's an important part of lean thinking as well. When a team agrees to do more work than they can actually accomplish by the time they'd agreed to deliver it, bad things happen. They either leave some of the work out of the delivery, do a poor job of building the product, or work at an unsustainable pace that will cost dearly in future releases. And sometimes it's not obvious that a team has taken on more work than they can handle: each individual deadline may seem reasonable for the work being done, but if each person is expected to multitask on multiple projects or tasks at the same time, the team slowly becomes overburdened, and the extra effort required for task switching can eat up as much as half of their productivity.

Visualizing the workflow helps the team see this overburdening problem clearly, and that's the first step toward fixing the problem. Unevenness and overburdening—which we learned about in Chatper 8—become clear on the kanban board when stickies always pile up in one column. Luckily, queuing theory doesn't just alert us to the problem; it also gives us a way to fix it. Once unevenness in the workflow has been identified, we can use it to control the amount of work that flows through the whole system by placing a strict limit on the amount of work that is allowed to pile up behind it. This is what's behind the Kanban practice **limit work in progress**.

Limiting work in progress (WIP) means setting a limit on the number of work items that can be in a particular stage in the project's workflow. A lot of people tend to focus on moving work items through the workflow as fast as they can. And for a single work item, that workflow is linear: if you're a developer and you're done with the design for a feature, and your workflow says that the next thing you do with it is build it and then send it on to a tester, the next thing you're going to do is build the code for it—and it's easy to become fixated on getting that feature built and sent on to the tester as quickly as possible because it was the last thing you were working on.

But what if the test team is already working on more features than they can test right now? It doesn't make sense to jump into development for this feature if it will just end up waiting around because nobody's ready to test it. That would cause overburdening for the test team. So what can be done about it?

For this, Kanban goes back to lean thinking—specifically, the principle of options thinking that you learned about in Chapter 8. One reason that a Kanban team uses a kanban board is because it shows all of your options. If you're that developer who just finished the design for a feature, it's easy to think that you now have a commitment to work on the code next. But working on the code for that particular feature is an option, not a commitment. When you look at the whole kanban board, you'll see many stickies that you can work on next. Maybe there are other stickies in earlier columns for other features that need to be designed, or ones in later columns for features with bugs that the testers found that need to be fixed. In fact, most of the time you have many options that you can choose from. Which do you choose first?

Setting a WIP limit for a step in your workflow means limiting the number of features that are allowed to move into that step. This helps limit the team's options to make that decision easier in a way that will prevent overburdening and keep the features flowing through the workflow as efficiently as possible. When you finish designing that feature, for example, and you see that the workflow is already at its limit for writing code, then you'll look for other options and work on them instead—and the test team won't get overburdened. (Think about this for a minute: can you see how this will reduce the average lead time for a feature? If so, then you're starting to get the hang of systems thinking!)

Let's go back to the team that used Five Whys to find the root cause of their lead time problem. Once we created a kanban board for them, the overburdening became clear: stickies started piling up in the "Manager Review" column. So to limit the work in progress for this team's workflow, we just need to find a way to place a strict limit on the number of features that pile up before the managers hold a review session.

In Kanban, once you recognize a workflow problem like this, the way that you deal with it is to **set a WIP limit** (or work in progress limit). To do this, the team needs to meet with the boss and the other senior managers and convince them to agree to a new policy. The new policy would set a WIP limit that says that only a certain number of features are allowed to pile up in the "Manager Review" column on the kanban board. The kanban board and lead time measurements give the team and their project manager a lot of objective evidence to help convince the managers to agree to this.

When we set a WIP limit on a column, it's no longer a sinkhole that collects a pile of work items. Instead, it becomes a **queue**, or a stage in your workflow where work items are managed in a smooth and orderly way.

There is no hard-and-fast rule that says how large the WIP limit should be; instead, teams will take an evolutionary approach to setting WIP limits. Kanban teams typically choose a WIP limit that makes sense and that everyone can agree on, and then use measurements to adjust it experimentally. For our team, let's say that each release typically has 30 features, and that the senior managers feel comfortable that they can meet three times over the course of the release, so we'll choose a WIP limit of 10. We'll do that by adding a number 10 to the "Manager Review" column on the kanban board, as shown in Figure 9-6.

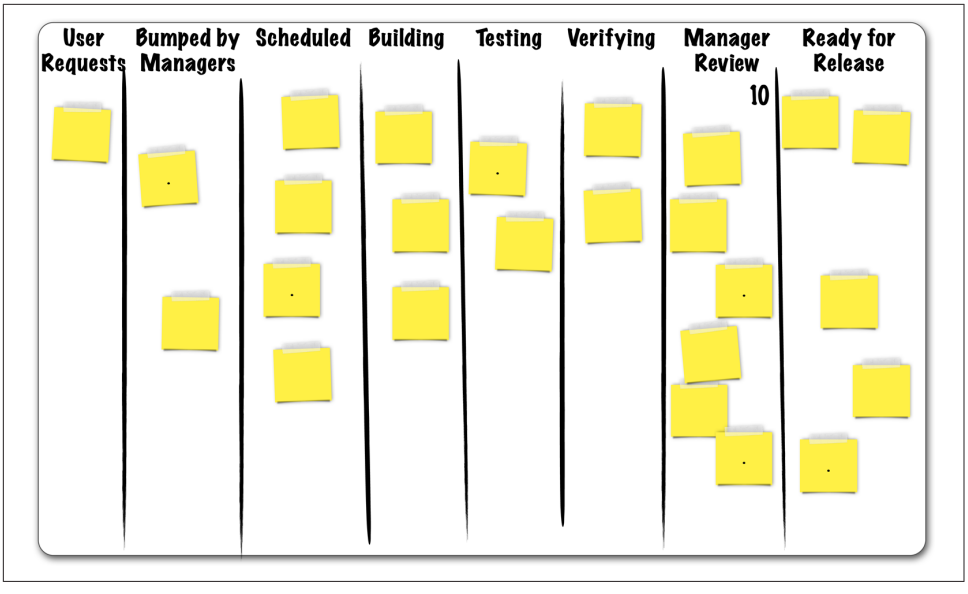

Figure 9-6. The number 10 in the Manager Review column is its WIP limit.

What happens when the tenth work item flows into the Manager Review column, causing it to hit its WIP limit? After that limit is reached, the team no longer pushes any more work items into that column. There's probably other work that they can do. If all of the development is done, they can help the QA team start testing the software. Maybe there's technical debt that they can start fixing—if there aren't stickies somewhere on the board for these items, there should be. The one thing that they *don't* do is push more features into the "Manager Review" column. That's the agreement that they've made with the senior managers: as soon as this column hits its WIP limit, the managers will hold the review meeting, and work will pile up behind it until that happens.

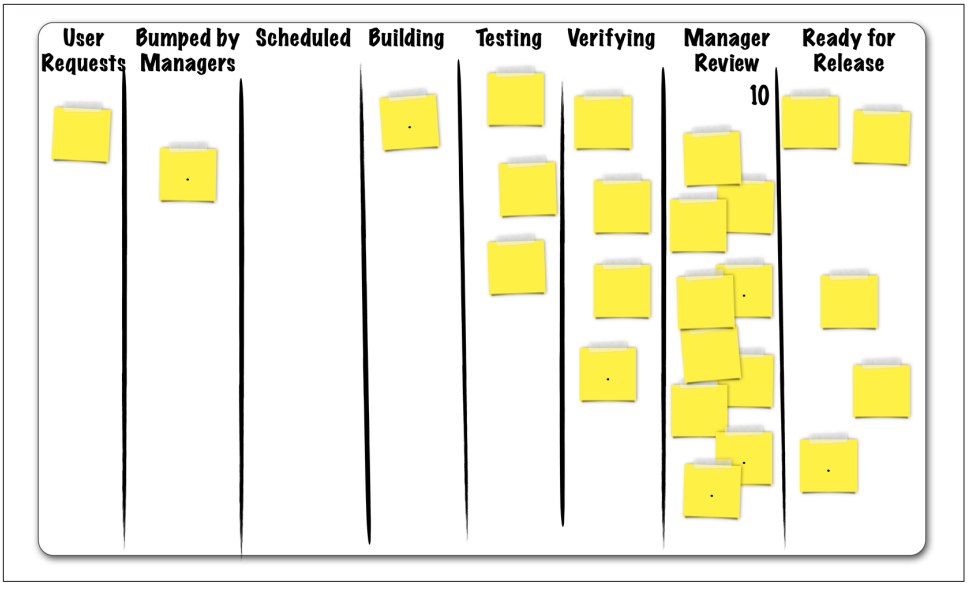

Figure 9-7. When a column reaches its WIP limit, no more stickies can be moved into it, even if the work for those stickies is done. The team shifts their focus to work on work items in other columns that have not yet reached a WIP limit.

Once the managers honor their agreement, hold the review meeting, and clear the logjam, the work can progress. The WIP limit causes the managers to give their feedback much earlier, and the team can immediately make the adjustments they need. If it looks like features are going to get bumped, the managers know at the beginning of project instead of the end. Now it's less disruptive to the team because they're not pushing so much work that's already been done into a "Bumped" backlog. The work they're doing becomes valuable immediately, rather than getting stale sitting in the "Bumped" backlog until it's no longer as relevant to the company (or until it makes customers so mad that they switch to a competitor).

The reason that this is effective is that *the cycle of manager review and team adjustment is a feedback loop*. When this feedback loop is too long—for example, when it's the length of the whole release—the information that's fed back into the project becomes disruptive, not helpful, because it causes the team to react by shelving a bunch of work that they already did. It causes waste.

Adding a WIP limit to fix the overburdening causes the average time a work item spends in the system to get much shorter. Now the managers hold their review multiple times over the course of the project, which allows the team to make their adjustments earlier. And that lets the managers use that information to change the priority of the features early enough in the project so that valuable work isn't wasted.

Even better, the team and the managers now have control over the length of the feedback loop. For example, if the managers find that this information is valuable, they can agree to shrink the WIP limit to six features, which will cause them to meet more frequently, and give earlier feedback to the team.

Why not make the WIP limit 1 and meet all the time? Aren't shorter feedback loops better?

Not if the team spends more time holding review meetings and dealing with feedback than they do actually getting work done. Like any tool, WIP limits and feedback loops can be abused. When a system has a feedback loop that's too short, it ends up in a state that's called **thrashing**. This is what happens when too much information is fed back into the system, and there isn't enough time to respond before the next batch of information is fed back.

Visualizing the workflow with a kanban board helps the team see the feedback loops, and experiment with WIP limits to find an optimal feedback loop length that provides frequent feedback that the team can respond to, but which allows them enough time to respond to that feedback before the next batch comes in.

Teams should also avoid sending the same items through the feedback loop many times, because this can clog the system. Once again, the kanban board makes it clear when this happens. For example, on our team's board, when managers have feedback and need it to be redone, they'll send it back to the backlog. This causes a team member to pull the sticky off of the board and move it back to an earlier column. The team can keep track of stickies that were moved backward on the board by adding a dot or other mark on them. This is a clear indicator that the feedback loop is going to be repeated for this feature. When the feature comes back through the workflow, it will end up in the "Manager Review" column again and take away one of the WIP limit slots. That has the effect of clogging up the feedback loop.

The team can avoid this problem by removing the extra loop from the workflow and making it more linear for most of the features. What if the team can convince the managers to agree to one feature review, and only choose some features to be bumped back into the backlog? If the managers can trust the team to accept their feedback for most of the features and not require an additional review for them before the feature is released, the team can add an extra development and testing step after the review.

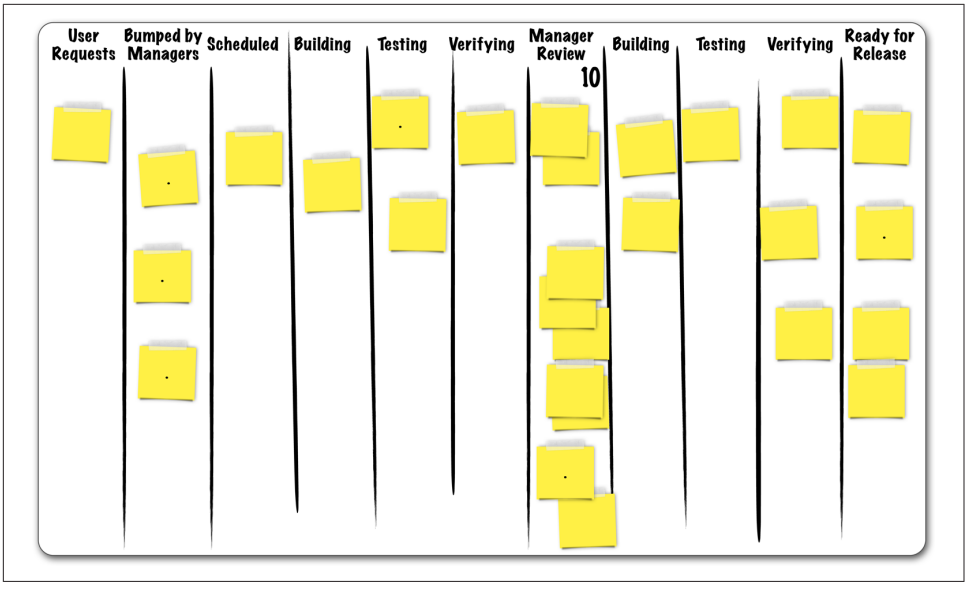

Figure 9-8. Adding extra columns to the kanban board—and preventing many stickies from being bumped back to an earlier column—gives the team more control over the process.

This workflow looks longer on the kanban board. But because we used objective evidence of lead time measurements and experimentation with WIP limits to evolve the workflow like this, we can be confident that it's actually faster. And we can keep measuring lead time and visualizing the workflow to prove to the team and the managers that even though there are more steps, it's faster for each feature to get through the workflow and included into the release. But while those measurements are useful for continuing to improve the project and to convince the managers that the project is improving, you probably won't need them in order to convince the team. They'll be convinced by their own results, because after unevenness and overburdening are removed, the project *feels faster*.

Key Points

- Kanban is a method for process improvement, or a way to help teams improve the way they build software and work together as a team, that's based on the Lean mindset.
- Kanban teams *start with what you do now* by seeing the whole as it exists today, and pursue incremental, evolutionary change to improve the system gradually over time.

- The Kanban practice *improve collaboratively, evolve experimentally* means taking measurements, making gradual improvements, and then confirming that they worked using those measurements.
- Every team has a system for building software (whether or not they recognize it), and the Lean idea of *systems thinking* means helping the team understand that system.
- A **kanban board** is a board way for Kanban teams to visualize their workflow.
- The kanban board has columns that represent each stage in the workflow of a work item, with stickies in the columns to represent each work item in the system that flow across the board as they advance through the system.
- Kanban boards use work items, not tasks, because Kanban is not a system for project management.
- When Kanban teams **limit work in progress** (WIP), they add a number to a column on the kanban board that represents that maximum number of work items allowed in that stage in the workflow.
- Kanban teams work with their users, managers, and other stakeholders to make sure that everyone agrees that when a column is at its WIP limit, the team will focus their work elsewhere in the project and not advance any more work items into that workflow stage.

Measure and Manage Flow

As teams continue to deliver work, they identify workflow problems and adjust their WIP limits so that the feedback loops provide enough information without causing thrashing. The **flow** of the system is the rate at which work items move through it. When the team finds an optimal pace for delivery combined with a comfortable amount of feedback, they've *maximized the flow*. Cutting down on the unevenness and overburdening, and letting your teams finish each task and move on to the next one, increases the flow of work through your project. When unevenness in the system causes work to pile up, it interrupts the work and decreases the flow. A Kanban team uses the **manage flow** practice by measuring the flow and taking active steps to improve it for the team.

You already know what it feels like when work is flowing. You feel like you're getting a lot accomplished, and that you aren't wasting time or stuck waiting for someone else

to do something for you. The personal feeling of being on a team with a lot of flow is that every day, all day, you have the feeling that you're doing something valuable. That's what everyone is striving for.

You also know what it feels like when work doesn't flow. It feels like you're mired in muck, and that you're barely making progress. It feels like you're always waiting for someone to finish building something that you need, or to make a decision that affects your work, or to approve some ticket, or somehow always find some other way to block your work—even when you know that nobody is intentionally trying to block it. It feels uncoordinated and disjointed, and you spend a lot of time explaining to other people why you're waiting. It's not that you're underallocated; your project team is probably 100% allocated, or even overallocated. But while your project plan may say that you're 90% done, it feels like there's still 90% of the project left to go.

And your users know what it feels like when work doesn't flow, because their lead times keep going up. It seems like the team takes longer and longer to respond to their requests, and even simple features seem to take forever to build.

The whole point of Kanban is to increase flow, and to get the whole team involved in increasing that flow. When the flow increases, frustration with unevenness and long lead times decreases.

Use CFDs and WIP Area Charts to Measure and Manage Flow

The kanban board is an important tool for managing flow specifically because it visualizes the source of the problem, and lets you limit work in progress where it will be most effective. When you look for the work that piles up and add a WIP limit to smooth it out, you're taking steps to increase the flow. The WIP limit works because you're helping the team focus their effort on the specific part of the project that's blocking work from flowing. In fact, *that's all that a WIP limit does*—it changes which work items the team is currently working on, and causes them to work on the ones that will even out the flow and clear up unevenness and workflow problems before they start to form.

But how do you know that you're actually increasing flow when you add WIP limits? Once again, we can go back to lean thinking, which tells us that we should take measurements—and an effective tool for measuring flow is a **cumulative flow diagram**, or CFD. A CFD is similar to a WIP area chart, with one important difference: instead of flowing off of the diagram, the work items *accumulate* in the final stripe or bar. And while the WIP area charts that you saw in Chapter 8 have stripes or bars that correspond to states in a value stream map, the CFDs (and WIP area charts) in this chapter have stripes that correspond to the columns on a kanban board.

The CFDs in this chapter have additional lines on them that show the average **arrival rate** (the number of work items added to the workflow every day) and average **inven-**

tory (the total number of work items in the workflow). The CFD can also show the average **lead time** (or the amount of time each work item stays in the system, just like we talked about with work items in Chapter 8). Not all CFDs have these lines; however, they are very helpful in understanding the flow of work items through the system.

The key to managing flow with a CFD is to look for patterns that indicate a problem. The kanban board can show you where the unevenness, loops, and other workflow problems are today, and helps you manage your flow on a day-to-day basis by adding WIP limits. The CFD lets you look at *the way your entire process is performing over time*, so you can take steps to find and fix the root cause of any long-term problems.

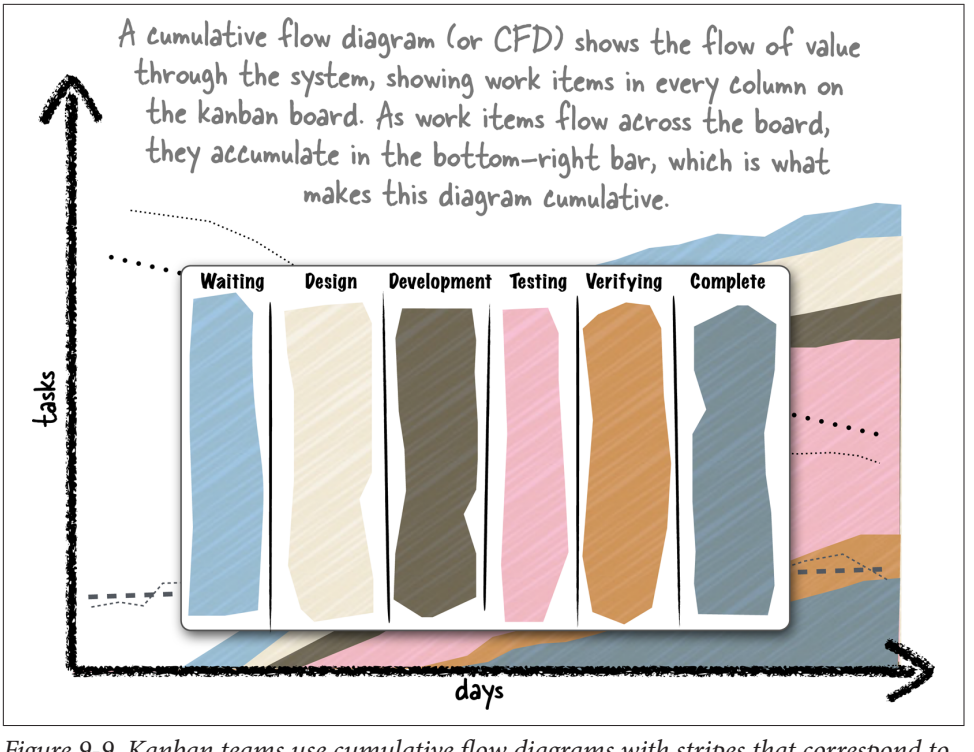

Figure 9-9. Kanban teams use cumulative flow diagrams with stripes that correspond to the columns on the kanban board. A cumulative flow diagram (CFD) is like a WIP area chart, except that instead of flowing off of the end of the chart, work items accumulate so all of the stripes or bars continue to go up over time.

How to build a cumulative flow diagram and use it to calculate the average lead time

To build a CFD, start with a WIP area chart. But instead of gathering data from a value stream map, you'll gather the data from the number of work items in each column on the kanban board.

Next, you'll need two additional pieces of data that you'll add to the diagram every day: the arrival rate and the inventory. To find the arrival rate for each day, count the number of work items that were added to the first column. To find the inventory for each day, count the total number of work items in every column. Add a dot to the CFD each day for the arrival rate and inventory, and connect them to create two line charts overlaid on top of the WIP area chart.

Most teams that use CFDs don't draw them incrementally on a wall; they use Excel or another spreadsheet program that supports charting. One reason—aside from ease of managing data—is that the spreadsheet can automatically add a linear trendline to the arrival rate and inventory line charts. These trendlines are very useful, because they can tell you whether or not your system is stable. If they're flat and horizontal, the system is stable. If one of them is tilted, then that value is changing over time. You'll need to *add WIP limits to stabilize your system*, and you'll be able to tell that the system is stable once those lines flatten out.

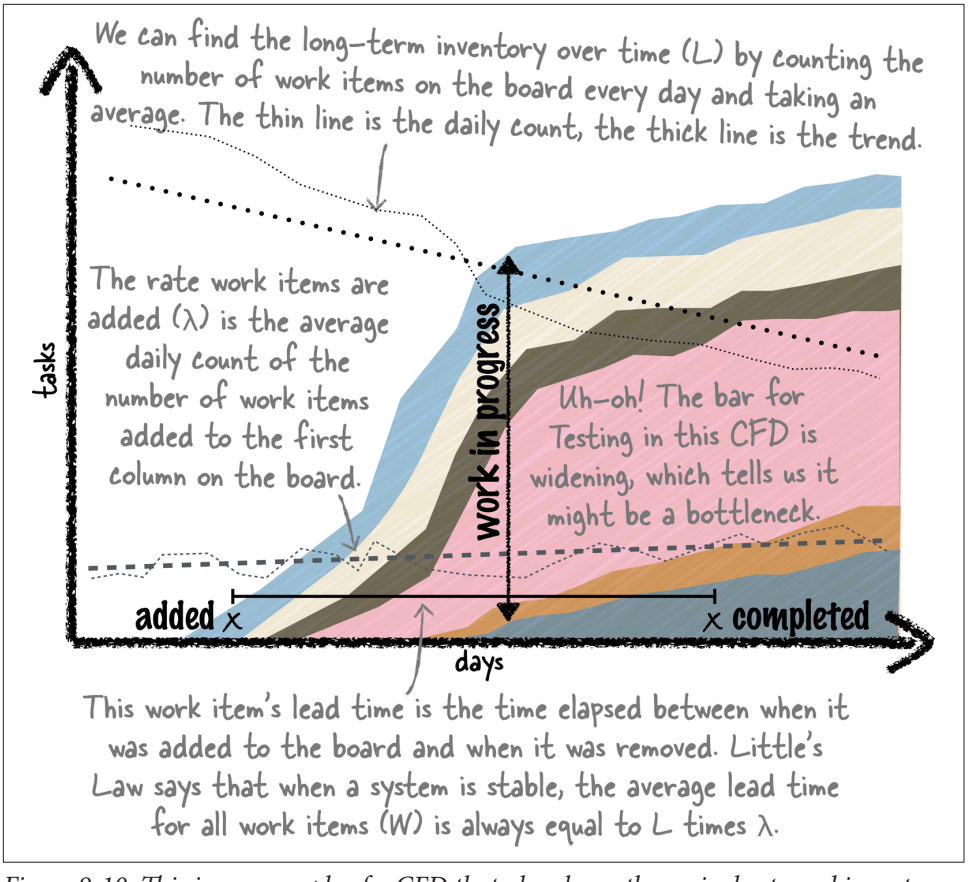

Figure 9-10. This is an example of a CFD that also shows the arrival rate and inventory. The total size of the work in progress at any given time can be found by measuring the difference between the top of the chart and the top of the "done" stripe. The horizontal solid black line shows the lead time for a specific work item in the system. We can't calculate the average lead time yet because the system is not stable—the long-term inventory and arrival rates have tilted trendlines, which means they aren't constant.

If you look at the hand-written notes in Figure 9-10, you'll see that we assigned letters to these values: we used L for the average long-term inventory, λ (that's a Greek lambda) for the average arrival rate (or the number of work items added every day), and W for the average lead time (or the average amount of time a user is *w*aiting for the team to finish a work item request).

Take a look at the inventory line in the CFD—that's the thick dotted line near the top. It's trending downward, which tells us that the total inventory is going down over time. It shows that many work items flowed out of the system and were never replaced. But if you look on the bottom, the arrival rate is increasing.

If we keep tracking this project, what will happen? Will the inventory fill up again? Will there be another release that empties work items out of the system? If more features arrive than leave, then over the long term the inventory will trend upward—and the team will feel it. They'll slowly have more and more work to do, and will feel like they have less time to do it. It's that "mired in muck" feeling that happens when the system isn't flowing.

Luckily, we know how to fix that problem: add WIP limits. The team can use experimentation and feedback loops to find WIP limit values that work for their system, and if they get them right eventually the rate that work items arrive will balance out with the rate that the team can finish them. The long-term inventory trend will be flat, and so will the long-term arrival rate. And once that happens, the system is **stable**.

And when a system is *stable*, there's a simple relationship between these values called **Little's Law**—a theorem that's part of queueing theory—named for its discoverer, John Little, who first proposed it in the 1950s, and who is considered by many to be the father of marketing science And even though there's a Greek letter involved, you don't need to know a lot of math to use it:

$$L = W \times \lambda$$

In English, this means that if you have a stable workflow, the average inventory is *always* equal to the average arrival rate multiplied by the average lead time. That's a mathematical law: it's been proven and if a system is stable, it is always true. And the reverse is true, too:

$$W = L \div \lambda$$

If you know the average inventory and the average arrival rate, *you can calculate the average lead time*. In fact, calculating the average inventory and arrival rate is pretty straightforward: just write down the total number of work items on your kanban board every day, and write down the number that were added to the first column that day. We show those on our CFDs using thin dashed or dotted lines. If your system is stable, then after some time you can find the average daily inventory and the daily arrival rate, which we show as thick straight lines. Divide the average inventory by the average arrival rate and you get the lead time.

Stop and think about that for a minute. Lead time is one of the best ways that you have to measure your users' frustration level: deliver quickly and your users are pleased; take a long time to deliver, and your users grow increasingly frustrated. And long lead times are good indicators of quality problems, as David Anderson pointed out in his book, *Kanban*:

> Longer lead times seem to be associated with significantly poorer quality. In fact, an approximately six-and-a-half times increase in average lead time resulted in a greater than 30-fold increase in initial defects. Longer average lead times result from greater amounts of work in progress. Hence, the management leverage point for improving quality is to reduce the quantity of work in progress.

Your lead time is determined entirely by the rate that work items arrive into your system, and the rate that they flow through it—and WIP limits give you control over the flow rate.

So what does this mean? It means that when your system is stable, you can cut your customers' lead time down *simply by not starting work on new features*.

Use a CFD to experiment with WIP limits and manage the flow

One of the core ideas of Kanban is that once you visualize the workflow, you can measure the flow, make your system stable, and actually take control of your project's lead time by managing the rate that you start work on new work items.

This may seem a little abstract. It's helpful to think about how you'd apply a CFD to a very simple system that most of us are familiar with: a doctor's office. Let's say that you had to visit a particular doctor several times to get a series of tests and discuss the results. You notice that if you have an appointment with the doctor in the morning after the office first opens, you don't have to wait very long. But if your appointment is later in the day, you have to sit in the waiting room for a long time—and the later the appointment, the longer the wait. Clearly this system is not stable. How would you use a CFD to stabilize it?

The first step would be to visualize the workflow. Let's say that every patient starts out by taking a seat in the waiting room. Eventually, a nurse calls the patient back to one of the exam rooms, where she gets weighed and has her blood pressure and temperature taken. Then she waits for the doctor to see her. In this office, there are five exam rooms and two doctors, and they're almost always occupied. More importantly, that means there can never be more than five patients in the exam rooms, or two patients seeing doctors. Those are WIP limits, imposed by the real-life constraints of the system.

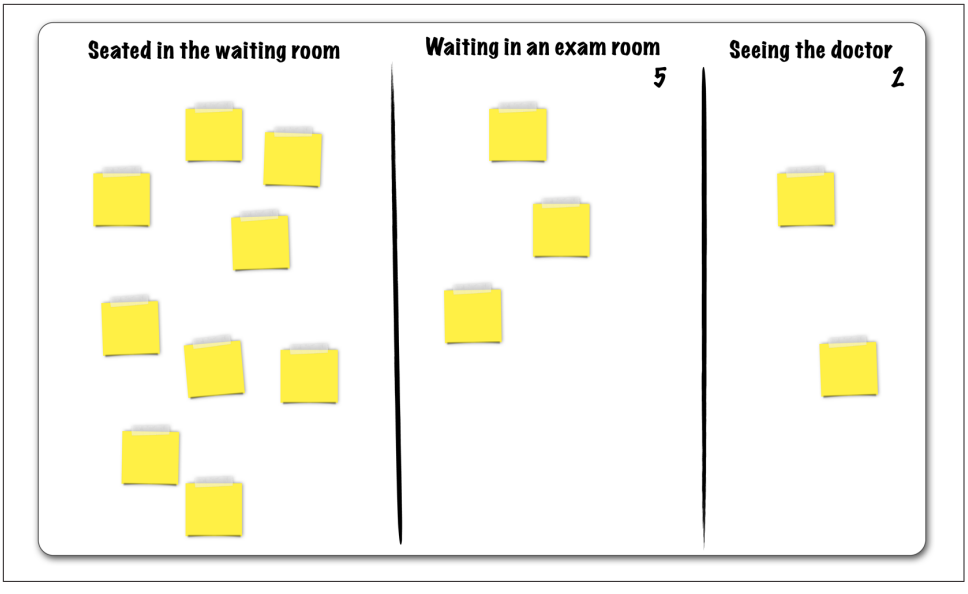

Figure 9-11. A kanban board for a doctor's office. The first column shows the patients who are currently in the waiting room, the second column shows the patients in the exam rooms, and the third shows the patients currently being seen by a doctor. There are five exam rooms and two doctors—these are natural WIP limits, so they're visualized on the board as well.

These doctors don't like the fact that their patients later in the day always complain of long wait times. What's worse, they feel more and more rushed to get the patients out of the office at the end of the day, and they're worried that they may not always be making the best medical decisions because they're under pressure to move patients through the office as fast as they can. Can Kanban help these doctors reduce their patients' waiting time and provide better care?

Let's find out. We'll start by building a CFD for a typical day. We'll have the office staff count the number of patients who walk into the office every 15 minutes. This gives us an arrival rate—literally the number of patients who arrived in the office. And we can count the inventory for each 15-minute interval by counting the total number of patients in the waiting room and the five exam rooms. Every time someone arrives, they'll add a sticky to the first column on the kanban board. When a patient moves from the waiting room to an exam room, they'll move the sticky to the second column. And when the doctor sees the patient, they'll move the sticky to the third column. Once the doctor is done with the patient, the sticky comes off the board. The office staff can record the counts of the stickies in the columns for every 15-minute interval.

Now they have all of the data that they need in order to build a CFD.

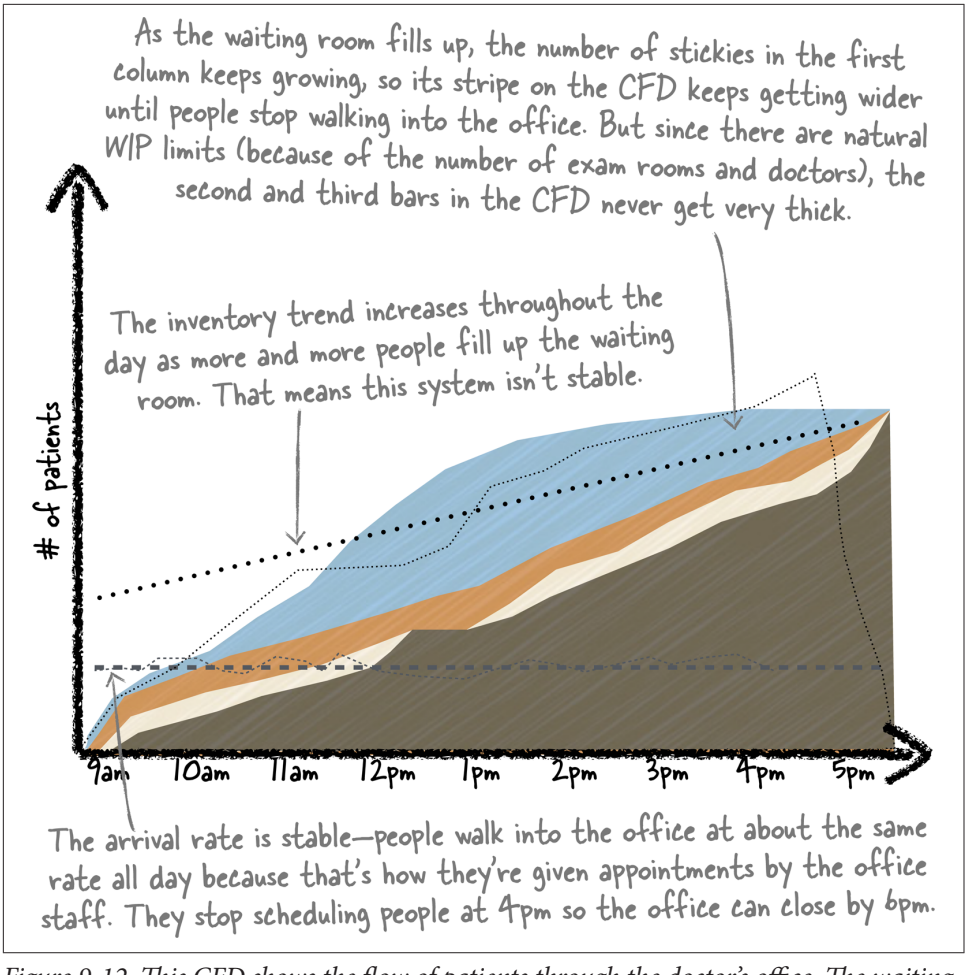

Figure 9-12. This CFD shows the flow of patients through the doctor's office. The waiting room gets more and more full throughout the day. Can you guess why the bar for the third column disappears between 12:15 p.m. and 1 p.m.?

This system isn't stable yet. The arrival rate is stable, because the office staff books patient appointments so that they arrive at a constant rate. They don't want to stay late, so they stop booking appointments at 4 p.m. People do run late for their appointments, but the trend for the arrival rate is flat so this must be pretty constant throughout the day.

The trend for the inventory, however, is not flat. It's tilted upward, because the inventory keeps growing and growing. This makes sense; the number of patients in the waiting room also grows throughout the day. So how can the office staff use this new information to improve patient care and reduce waiting times?

The first thing they need to do is stabilize the system—and the tool we have for that is setting a WIP limit. They'll use the Kanban practice of evolving collaboratively and improving experimentally by deciding on a WIP limit together, and using that as a starting point. After looking at the data, everyone decides to add a WIP limit of six to the waiting room. But there's a tough decision that needs to be made: the doctors have to agree that if there are already six patients in the waiting room, then the office staff must start calling low-priority patients scheduled for the next hour and ask to reschedule their appointments (but they will find a way to handle more serious cases without compromising patient care). They'll also ask patients in the waiting room if anyone would be willing to volunteer to reschedule their appointment—and they promise to give that patient priority for the rescheduled appointment. This is a *new policy* that they need to make explicit. They'll do that by adding a WIP limit to the kanban board, and also posting a big notice to patients on a piece of paper at the front desk letting them know that this is the policy going forward.

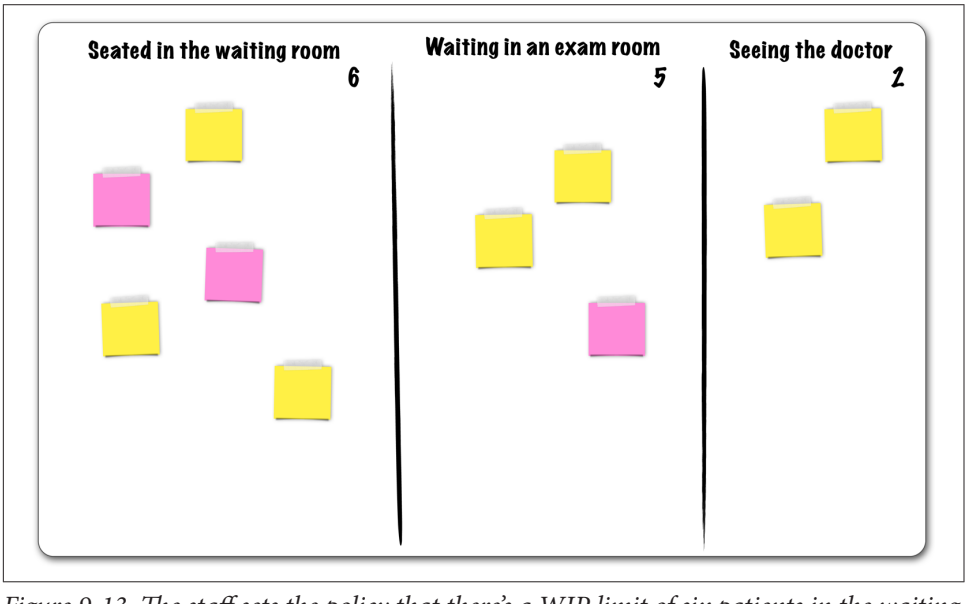

Figure 9-13. The staff sets the policy that there's a WIP limit of six patients in the waiting room. They enforce this policy by calling up patients and rescheduling them as soon as the waiting room hits its WIP limit. They use pink stickies to indicate patients with more serious problems who can't be rescheduled.

It takes a little practice, but after a few days the office staff gets used to the new policy. They discover that they need to take the patient's complaint into account. They decide to do this by defining the **class of service**: they'll use a pink sticky to indicate patients who have a more severe condition and cannot be rescheduled, and keep the yellow

stickies to indicate patients who have more minor problems. This allows them to provide more prompt service for patients who need it the most.

And it works! The office staff discovers that once they impose the WIP limit, they no longer have to stop scheduling appointments at 4 p.m. in order to be done by 6 p.m. —they can schedule patients as late as 5:40 p.m. as long as they never schedule patients with more severe problems later than 4:40 p.m. (and they write down this policy as well). Obviously, if someone has a very severe problem and walks into the office, the doctor will see that patient (or send them to the emergency room)—but that's a rare exception, and because the office staff are smart and responsible, they can handle situations like that on a case-by-case basis. The patients seem much happier because they're not waiting as long to see the doctor.

Little's Law Lets You Control the Flow Through a System

The office staff took the Kanban practice of improving experimentally very seriously. And through their experiments, they discovered something interesting: once they found a good WIP limit, they *could control how long patients had to wait*. If they scheduled more appointments every hour, there would be four or five patients in the waiting room, and patients would have to wait longer. If they scheduled fewer appointments every hour, there would only be two or three patients waiting to see the doctor, and they would wait less time. This gave them a feeling that for the first time, they were really in control of a system that had caused them so many headaches in the past.

So what's going on here?

What the office staff discovered is that in a stable system, there's a relationship between inventory, lead time, and arrival rate. For example, if the office staff schedules 11 patients to arrive every hour (so the arrival rate λ is 11/hour), and the office has average inventory of seven patients over the course of the day (so the inventory L is 7), then Little's Law tells us the average time patients have to wait:

$$W = L \div \lambda = 7 \text{ patients} \div (11 \text{ per hour}) = .63 \text{ hours} = 37 \text{ minutes}$$

But what if after some experimentation, they discover that scheduling 10 patients to arrive every hour causes the average inventory to drop to 4 patients? There are going to be peak times during the day when all of the exam rooms are filled, but most of the time there's one patient in the waiting room, one patient in an exam room waiting to see a doctor, and two patients in exam rooms talking to the doctors:

$$W = 4 \text{ patients} \div (10 \text{ patients per hour}) = .4 \text{ hours} = 24 \text{ minutes}$$

By using a kanban board and a CFD and experimenting with WIP limits, the office staff discovered that they could reduce the patient waiting time by almost 15 minutes just by scheduling one fewer patient per hour.

This works because Little's Law tells us that lead time in a stable system is affected by exactly two things, inventory and arrival rate—and WIP limits let you control one of those things. When you add WIP limits to your kanban board, you can reduce unevenness, which keeps inventory from piling up. That gives you a straightforward way to reduce lead time: by reducing the arrival rate (for example, by keeping work in a backlog until the team has time to deal with it, just like Scrum teams do—or by reducing the number of patients scheduled per hour).

This is why the office staff can use Little's Law to calculate the average lead time. But even if they never calculate it, *they're still affected by it*. The reason is that Little's Law *always* applies to every stable system, whether or not the team is aware of it. This affects your projects because it isn't just theory. It's a proven mathematical law that applies to any system with a stable long-term inventory.

So now that we've seen a simple example, let's have a look at an example that's closer to real life. Let's say that every three weeks, your entire team gets mired with supporting production releases.

Unfortunately, things aren't going so well for this team. At first everything was working just fine. But over time, problems started to crop up. Even though all of the support work eventually gets done, the team doesn't feel like there's enough time to do it. Instead, everyone feels like this month is always more stressful than last month—and everyone knows that if they don't get the support work done as quickly as possible, next month will be even worse.

This is a familiar feeling for a lot of teams. It feels like the project is slowly sinking in quicksand, and if it keeps up, eventually it will get so bad that the people on the team don't feel like they have enough time to think about their work. We learned in Chapter 7 what kind of damage an environment like that can do to the code: developers will end up building poorly designed software, and will create a codebase that has a lot of technical debt and is difficult to work with.

What can we do about this problem? It won't necessarily be obvious just from looking at the kanban board every day, because most of the time the board looks more or less healthy. There are some extra stickies on the board while the team is doing work to support the release, but that's expected. Eventually those stickies flow off the board, and it seems to return to normal. But it doesn't feel normal to the team (or, worse, it does feel normal—but it doesn't feel good!).

Can the tools that we've learned about help us find and fix the team's problem? Let's have a look.

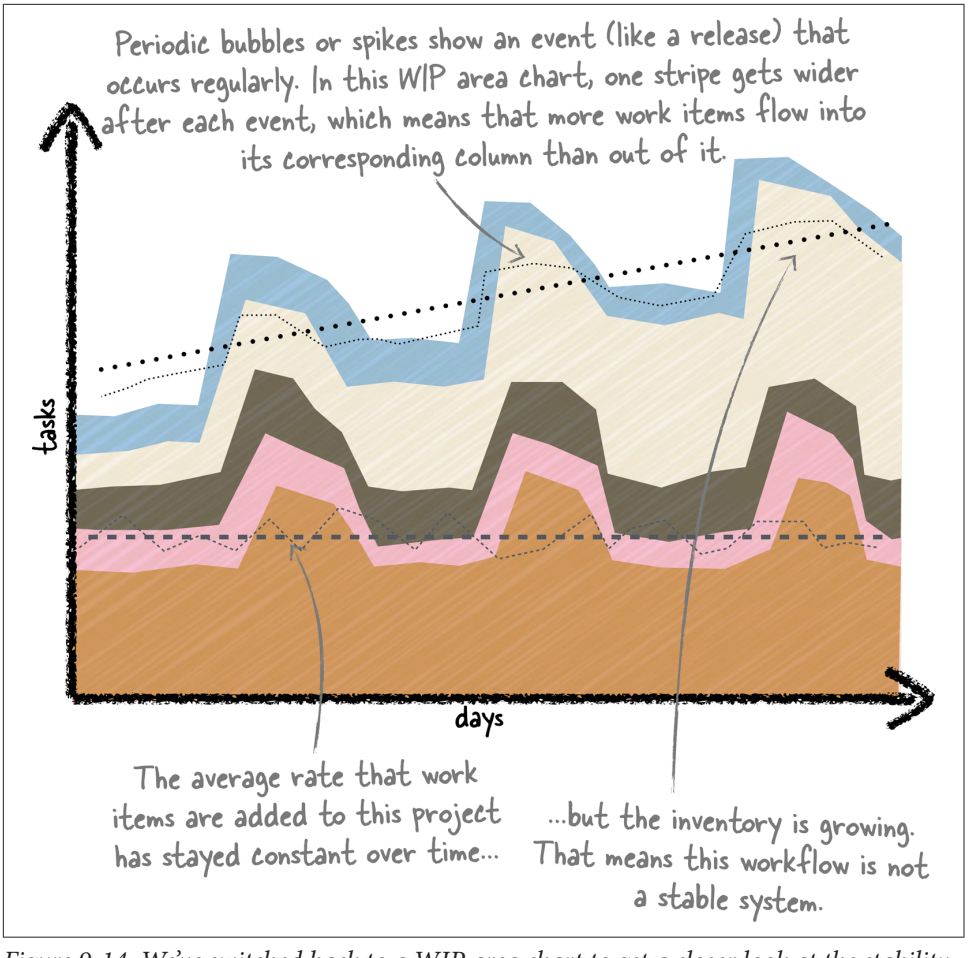

Figure 9-14. We've switched back to a WIP area chart to get a closer look at the stability of the system. This diagram shows that the arrival rate is constant but that the average inventory increases over time, which means the system isn't stable.

The additional work for each production release causes a visible spike in the CFD. Remember, each stripe corresponds to one of the columns on the board, and the vertical thickness of the stripe represents the number of stickies in its column for that day. In this CFD, one of the stripes gets thicker and thicker after each release, and that causes the total height of the diagram to slowly go up over time.

The WIP area chart makes the cause of the problem clearer: work is *accumulating* in that column because more work is flowing into it than out over time. If the team doesn't make a change, each new release will push them further behind and that stripe will keep getting thicker after every release. And we can see that the long-term

inventory is going up, which means our system isn't stable—work isn't flowing. No wonder the team feels like they're sinking.

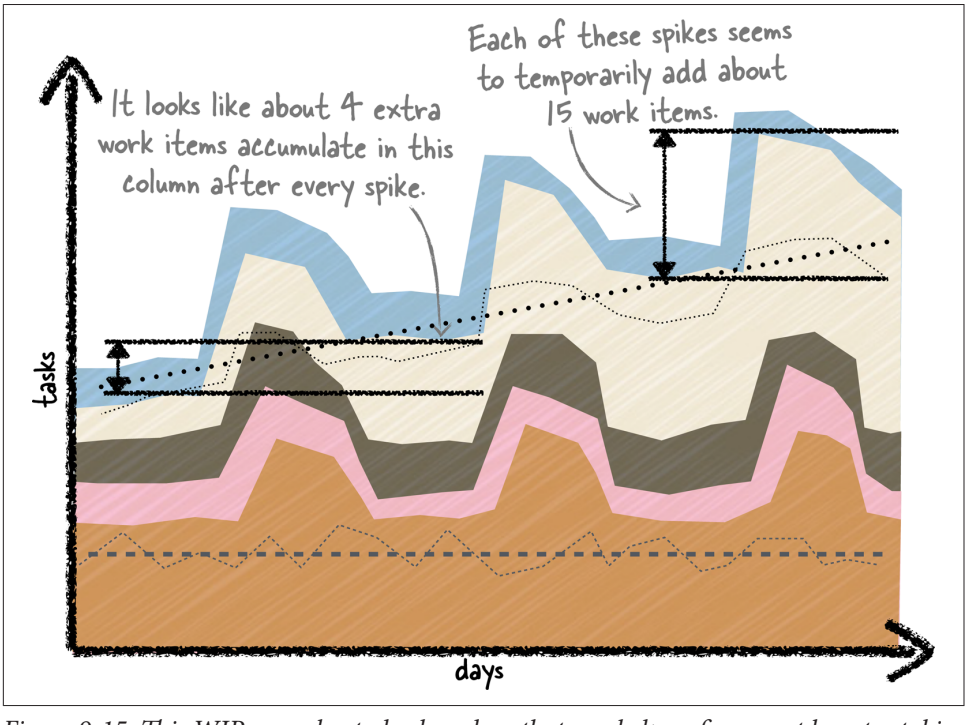

Figure 9-15. This WIP area chart also has clues that can help us figure out how to stabilize the system. The average inventory increases after periodic spikes in the work. If we can figure out how much extra inventory accumulates, it will help us choose a good starting point for experimenting with a WIP limit.

The team may not realize that work is getting "stuck" in one column on the kanban board. The WIP area chart makes this a lot easier to spot, because the corresponding stripe for that column gets thicker over time. Luckily, we already have a tool to smooth out this unevenness: adding WIP limits. And measuring the number of work items that seem to be getting "stuck" in the column can help us find a good starting point.

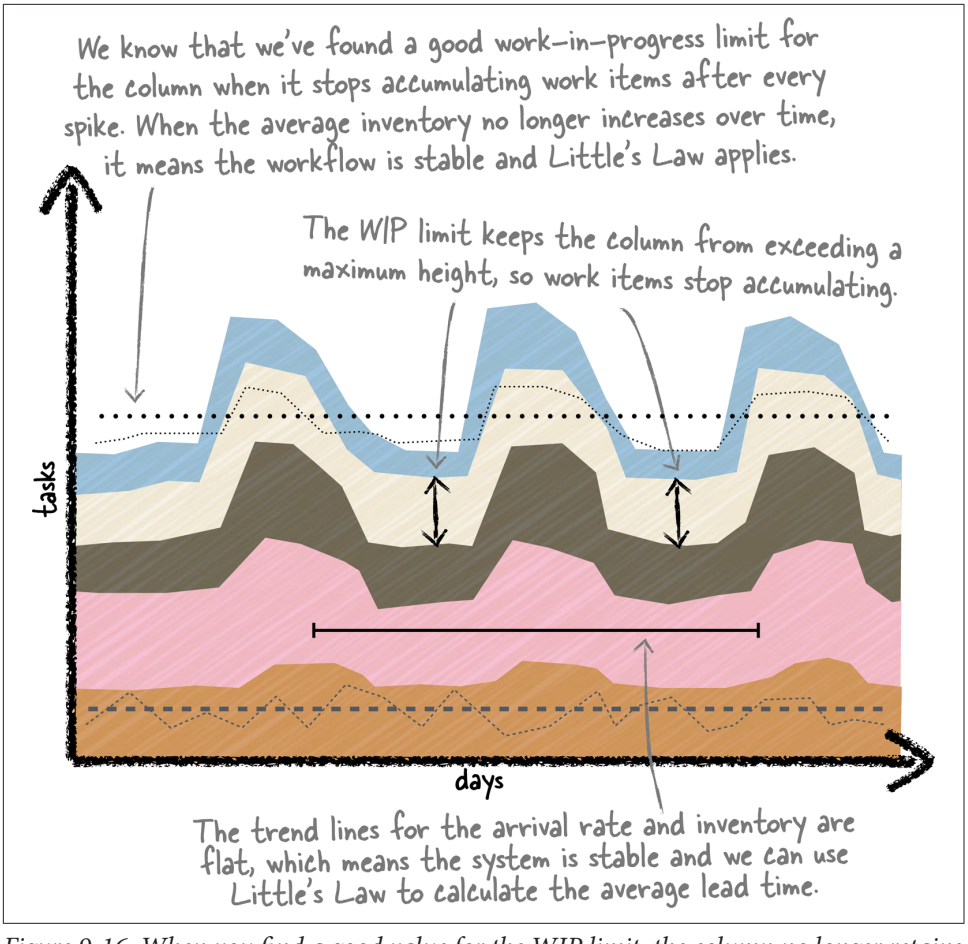

Figure 9-16. When you find a good value for the WIP limit, the column no longer retains work, and the stripes on the WIP area chart don't accumulate work over time. How would this look on a CFD instead? Do you think a CFD would do a better job of visualizing this particular problem than a WIP area chart?

The spikes are still there, but there's *no longer an increase in the total number of tasks*. By adding a queue, we stopped extra work from flowing into the system, which increases the overall flow for the whole project. The work is more evenly distributed across all of the columns. When the overburdened column hits its WIP limit, the team shifts their focus to work items in earlier columns. You can see this on the WIP area chart: the bumps on the lower stripes are smaller than they were before the WIP limit.

Kanban teams don't just set a WIP limit and stop, however. They implement a feedback loop: as the project continues, they keep adjusting the WIP limit based on new

information. If there's still accumulation, the team can experiment with different WIP limits until we find a value that gives us the most flow. Kanban teams take that experimentation seriously: they don't just randomly set these WIP limits; they run experiments by forming a hypothesis about how WIP limits will affect the system, and carefully prove them out by taking measurements. This is how the team can improve collaboratively and evolve experimentally.

As the team collaborates to evolve their system, they'll feel that increased flow in their day-to-day work. The support tasks won't cause them to feel like they're putting off important development work because they're treating the support work as features that need to be developed. By including work items for the support work on the board, the team has effectively promoted support tasks to first-class citizens of the project, and can focus on them and do a better job with them (instead of trying to cram them in and rush through them—and, in the process, create more work for themselves).

There's an added long-term benefit as well. Many of the support issues may be caused by technical debt that the team was forced to add in order to keep up with the backlog; now that they have time to do the job right, they might discover that they have fewer support issues in the future. In the meantime, the team can enjoy increased focus and a more energized work environment, which will let them build better software.

If the team still has to do all of that production support work, then what happened to the other work they were doing? Won't it accumulate somewhere else?

No. The reason that extra work isn't accumulating anymore is because *it isn't being added to the project in the first place*. This team was feeling stress because they were expected to put all of their effort into building software—but at the same time, they were also expected to stop every few weeks and concentrate on support without affecting development work. This was a choice made by their boss. Or, more accurately, it was a choice that the boss refused to make. His magical thinking let him pretend that the team could handle a full workload of development work while still taking on extra support work every few weeks. The queue will force the boss to choose which work the team will do.

But wait a minute—what if the support work takes up all of the slots in the queue? How will any development work get done?

If the support work seems to be "clogging" up the queue, that means the boss made a choice to prioritize support work over development work. It's the team's top priority, whether or not the boss explicitly acknowledges it; putting work items for support on the kanban board is a way for the team to give support the attention that top-priority items deserve. It's no longer the team's fault anymore if they don't do any new development. Of course, many developers prefer to do development than support; they

might not like being turned into a full-time support team. But this is better than being responsible for all support work, while still being held responsible for all of the development work too.

And now the boss has much more accurate information about the progress that the team is making. This is one of the principles of agile software development from Chapter 3: *working software is the primary measure of progress*. Before, the overloaded team was seemingly able to produce software and do support; it wasn't immediately obvious that the extra load was causing them to build worse software that was more difficult to maintain, and potentially causing some of those support issues. Now that the team is delivering less software, the boss has a more accurate measure of progress. That makes it much more difficult for him to use magical thinking to pretend that the team can do much more work than is humanly possible. If he wants more software built, he either needs to prioritize it over support work, or hire additional people. But it's a lot less easy for him to simply blame the team.

Managing Flow with WIP Limits Naturally Creates Slack

Developers need slack, or "wiggle room," in the schedule. They need it to make sure that they have time to do a good job. We saw in the XP chapters that when a developer feels like he doesn't have enough time to think about the work, he cuts corners and adds technical debt. He has a mindset where he wants to get his current task done as quickly as possible, because there's always more work to do, and the project is always behind.

This kind of "always behind, always rushed" atmosphere leaves little or no room for creativity, and stifles innovation at every turn. It also kills quality practices, because teams that feel like they're running behind will often cut out any activity that doesn't directly produce code. This is why XP teams include Slack in their primary practices.

Teams using Kanban also value slack, and understand the impact that slack has on each team member's ability to do their best work. This is one of the main reasons that they limit work in progress. And when teams use Kanban to improve the process, instead of following a strict timebox they will adopt a **delivery cadence**. To do this, they commit to delivering software on a regular schedule (for example, a team might commit to releasing software every six weeks)—but they don't commit to a specific set of work items that will be included with the release. Instead, they trust their system to deliver work items. If they've removed overburdening and unevenness, then they will naturally get a set of completed work items to include in each delivery.

So what, specifically, causes a programmer or other team member to feel like there isn't enough time? That feeling is caused by a vivid awareness of the amount of work left to do, and that his or her current work is blocking the rest of the project. When someone feels like the current work is just an impediment preventing more work from getting done, he'll naturally try to get through it as quickly as possible.

This is why many people have a surprising reaction when the team gets their managers to agree to a WIP limit: *a feeling of relief*.

Before the WIP limit, extra work always seemed to sneak into the sprint, and the team had to rely on slack to cover that extra work. They'd always go into the sprint assuming that they'd only have, say, 70% of the work figured out, because impatient bosses and users would find a way to cram in the other 30% (or, worse, 40% or more!) with last-minute changes and emergency requests.

After the WIP limit—and, just as importantly, the agreement to stick to the WIP limit—these extra requests will still come in, but now the team doesn't have to try to absorb the impact of the unplanned work. Instead, the new work goes into a queue that's been created by adding a WIP limit somewhere in the workflow. There's less pressure, because they know that there won't be an unlimited amount of work that piles up. They've managed their flow (possibly using a CFD), so they know that the queue's WIP limit is set to a level that will give them enough time.

This is why many Kanban teams end up setting WIP limits on every column on the kanban board.

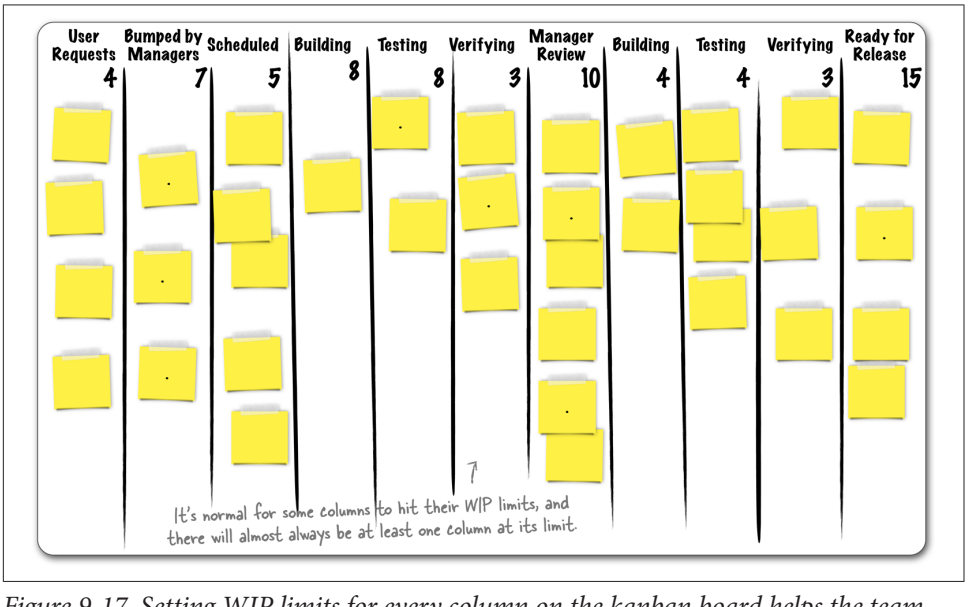

Figure 9-17. Setting WIP limits for every column on the kanban board helps the team maximize flow throughout the entire project.

This helps the team control the flow through each step in development. There's even a WIP limit on the "Ready for Release" column. If there is too much *"done"* done work that's piled up and is ready for release, they can find other work to do to prepare for

the next release—and now they have better information to adjust their delivery cadence in the future by reducing the time between releases.

The team can't always get everyone to agree to WIP limits everywhere in the first step; this is why Kanban teams follow their cycle of improving collaboratively and evolving experimentally. After seeing that WIP limits help the team build software more quickly and reduce lead time after the first round of improvement, managers are more likely to agree to additional WIP limits in later rounds.

Controlling the flow across the entire project helps everyone on the team relax and focus on the work that they're doing now, without having to worry that tasks are piling up behind them. They can trust the WIP limits to keep the chaos out. And they know that if chaos starts to leak in, it will show up when they measure the workflow, and they can adjust the WIP limits and cadence to keep work focused and flowing.

Make Process Policies Explicit So Everyone Is on the Same Page

What would happen if you asked everyone on an effective Scrum team to write a detailed description of how they build software? There's a good chance that almost all of their descriptions will match up pretty well. That's because everyone on the team is familiar with the rules of Scrum, and they've all been working consistently with those rules. Every single person on an effective Scrum team has a common understanding of how the whole team builds software, and the rules for a Scrum project are simple enough that everyone on the team can understand them.

On the other hand, what would happen if you asked the same thing of a typical, ineffective waterfall team? There's a good chance that they have a fractured perspective (just like we learned about in Chapter 2). Everyone would probably write down how they do their own work on a day-to-day basis: developers would write about coding, testers about testing, business analysts about requirements gathering, etc. The project manager *may* have a larger perspective, because she needs to understand what everyone does in order to build a project plan, so she *might* actually write a description that includes the work done by everyone on the team. But she might also just describe the steps that she follows to plan and track the project.

Sometimes a complex process is important and useful; other times, it's wasteful bureaucracy. Say a team is in the habit of sending around many emails any time a specification is updated, and the team can't (or won't) make any changes until enough people seem to be on board. That's a process, and even if it's not written down it's part of the team's "tribal" knowledge. How do you know which of those categories this particular spec updating process falls into? The answer is in the Kanban practice **make process policies explicit**—in other words, write down a description of how the team works, and show it to everyone who's affected by it. It may be that a complex process is required. The project manager, for example, might point out that this spec change control process is required for regulatory compliance. But more often, simply

making an unwritten policy explicit can cause people to shake their heads and agree that it's stupid.

Kanban teams don't need to write long documents or create huge Wikis to establish explicit policies. Policies can be as simple as WIP limits at the tops of the columns. Teams can also write down their policies by adding "definitions of done" or "exit criteria" bullets to the bottom of each column on a kanban board, so that the team members know exactly when to advance the work items through the workflow. This is especially effective when these policies were established collaboratively and evolved experimentally by the whole team, because it means that everyone understands why they're there.

Complex processes and unwritten rules tend to emerge over time, and they seem to be especially common on teams that have a fractured perspective. A complex change control process might come about because a business analyst has trouble keeping up with many last-minute changes, and is blamed loudly in front of the entire team any time the software doesn't meet a specific user's needs. He might impose a complex process to keep control of the scope, and to make sure that there's a paper trail so that he can CYA if someone decides later on to change his or her mind. It's hard to blame this business analyst for defensively creating bureaucracy; it may be the only way he can exert control over a spec that he's held accountable for.

Setting WIP limits is a policy choice, and that policy only works because everyone honors the agreement not to push more work on to a queue if it's full and has already reached its limit. Writing down the agreement—especially if it's in a visible place, like a kanban board—helps to make sure that everyone still agrees to it after it's written down. The team can point to that written policy any time an overeager manager or user tries to push additional work onto the queue. This gives the team solid ground when they need to pull another item off of the queue to make room for an urgent request. That discussion is much easier for the team if the manager or user has already agreed to a policy that was made explicit.

Emergent Behavior with Kanban

> The more you tighten your grip, Tarkin, the more star systems will slip through your fingers.
>
> —Princess Leia

Think back to the doctor's office example from earlier in this chapter. If you had asked a team of command-and-control project managers to help the office staff improve their process, what would have happened? The first step in command-and-control project management is often estimation, so they might make the doctors come up with an estimate for how long they would spend with each patient. That would require a lot of up-front analysis by the doctors, nurses, and office staff, but it would allow the project managers to take control of the system and create a complete

schedule for all of the resources—the exam rooms, doctors, and nurses—and micromanage every aspect of the practice.

This would be a stifling way to work; more importantly, it would be pretty ineffective. Doctors are trained in medicine, not estimation. It's possible that the project managers might come up with an ideal schedule; it's more likely that they'll create an unrealistic or ineffective one because the doctors will give them poor estimates. This is why Kanban looks at the entire system *in aggregate*. Instead of trying to micromanage every little activity, a team using Kanban uses systems thinking to understand, measure, and incrementally improve the system as a whole. They accept the fact that individual work items will vary, but the system as a whole acts predictably when unevenness, overburdening, and futility—muda, mura, and muri—are reduced.

When a team uses Kanban to gradually improve their process for building software, a funny thing often happens: the rest of the company starts to change. Consider the example of our team trying to reduce lead time. Before the team tried Kanban, the managers and users blamed the team for not responding to user requests quickly enough; the team reacted by creating project plans that "proved" that nothing was wrong. This led to conflict and bad feelings.

Kanban helped fix the underlying problem. The team used the Five Whys technique to find the root cause of the lead time problem, and introduced WIP limits to help fix it.

But how, exactly, did that fix the problem? Let's get back to the first two Kanban practices that we talked about in this chapter—**improve collaboratively**, **evolve experimentally**, and **implement feedback loops**—to understand what's going on. The first thing that it did was to *literally* change the way that everyone—including the managers—looked at the process. Visualizing the workflow with a kanban board helped to un-fracture their perspectives. Managers could see that work was piling up, and that helped convince them to agree to WIP limits, and to hold their reviews more often. This is an example of how setting a WIP limit causes behavior to change.

Consider, for example, a team that constantly gets requests from several different managers, and has to balance the requests between them. Each manager considers his or her request to be the most important one; if the team works on a request from one manager, another will feel snubbed. Everyone feels a lot of pressure because they're faced with an impossible choice.

If this team uses Kanban to visualize the workflow, all of the managers' requests for features will end up as work items in the first column. If there are more requests than the team can handle, then stickies will pile up in that column, and everyone can see it. Now the team knows what to do: get all of the managers to agree to a WIP limit on that column.

Let's say, just for this example, that they're able to get this agreement and establish that WIP limit. The managers will keep adding new feature requests as usual, and until that WIP limit is hit nothing changes. But as soon as that first manager hits the WIP limit, something interesting happens: she can't just add the new sticky to the board. Instead, she has to choose another one to remove. And if she doesn't want to remove one of her own, then she has to talk to the other managers.

That's worth repeating: when the manager ran into the WIP limit, she didn't blame the team. She *recognized it as a limitation of the system*, and worked with the other managers to find a solution. This is an important part of systems thinking. When everyone who works with the system recognizes it, they work within that system to solve their problems. And because everyone understands this system, the unevenness and overburdening are no longer just the team's problem. They're everyone's problem, including the managers.

This is how *new behavior emerges outside of the team*. Instead of just blaming the team for overburdening that isn't necessarily their own fault, everyone becomes very concerned with stickies queued up in that first column, because those are the ones that will be worked on by the team. They may have prioritization meetings, or do "horse trading" among themselves, or find some other way to decide on what tasks go into the queue. But there's one thing that does not happen anymore: the team no longer has to take the blame, because they're no longer expected to do more work than they are physically capable of.

In other words, before Kanban, the team had to deal with managers who had magical thinking, and expected to be able to push an unlimited amount of work on to the team. They were always disappointed with the results, and felt that they were promised things that were never delivered. With the WIP limit and an explicit policy, that magical thinking was eliminated. The managers changed their behavior, not because they were asked to change, but because they were working within a system that naturally encouraged them to behave differently. As a result, the team had the slack they needed to relax and do the work right. All of the effort that they'd been using to negotiate with managers individually about their requests could now be focused entirely on their work. That's a much more effective—and pleasant—way to run a team.

Key Points

- One goal of a Kanban team is to maximize the **flow**, or the rate at which work items move through the system.
- The Kanban practice **measure and manage flow** means taking measurements of the flow and making adjustments to the process to reach maximum flow.

- A **cumulative flow diagram** (or CFD) is a WIP area chart that also shows the number of work items added to the workflow every day (**arrival rate**), the total number of work items in the workflow (**inventory**), and the average amount of time each work item stays in the system (**lead time**).
- When the arrival rate and inventory do not change over time, the system is **stable**; Kanban teams set WIP limits to stabilize the system.
- When a system is stable, **Little's Law** applies, which means that the average lead time is always equal to the long-term arrival rate times the long-term inventory.
- If your team can stabilize the workflow with WIP limits, you can reduce the lead time for your users by getting your stakeholders to agree not to add new work items, which reduces the arrival rate.
- Kanban teams often **make process policies explicit** by adding definitions of done or exit criteria to the bottom of each column on the kanban board.
- When Kanban teams gradually improve the system by adding WIP limits, managing flow, and making process policies explicit, improved behavior often emerges in the rest of the company.

Frequently Asked Questions

You talk about WIP limits and explicit policies like they can completely change how a team operates. This sounds far-fetched. Is that really true?

Amazingly, yes. To understand why, it helps to go back to the origins of Lean and Kanban.

WIP limits are a simple but very effective tool for smoothing out flow, and this effect has been known for a long time. Kanban is based on a system for signaling work that originated at Toyota in the 1950s (like many of the ideas and tools of Lean). The name "Kanban" comes from the Japanese word that means "signal card" (literally "signboard" or "billboard"). In the manufacturing plant, a station on an assembly line would be given a certain number of kanbans—physical cards with a part number or name printed on it—for each type of part used at that station. When they ran low and needed more of that part, they would leave the kanban in an empty parts cart. The