How and why to create Value Stream Maps for software engineering projects

Ted Rivera

March 04, 2010

Although Value Stream Mapping is well-understood for manufacturing applications, its utility is comparatively unknown in the software engineering community. This article explains how you can start a Value Stream Mapping process and reveals how Value Stream Maps can positively impact how your software is produced.

Value Stream Maps exist for two purposes: to help organizations identify and end wasteful activities. Finding problems and creating a more efficient process isn't easy; even the best organization can be made more efficient and effective. But bringing about substantive organizational change that actually eliminates waste is a tall order. It's comparatively easy to identify waste, but it's another matter entirely to stop waste from happening in the first place. Value Stream Maps can both sharpen an organization’s skills in identifying waste and help drive needed change. But first things first: What a Value Stream Map is and how one can be intelligently produced. The examples and concepts that follow are based on applying a Value Stream Map to a software engineering organization, but these concepts are applicable to a wide range of settings.

How to create a Value Stream Map

A simple example illustrates how a Value Stream Map can be created. A particular team faced travel restrictions, and they had been asked to work with another group outside of their organization to form an agreement as to how to collaborate on an important project. The agreement needed to be in place prior to commencing the work. Figure 1 shows the Value Stream Map they came up with to describe their situation.
Consider the various elements that make up this basic Value Stream Map:

1. An organizational process or activity is identified as potentially wasteful. This can be a simple or complex process or activity. Just choose a process or a problem that you believe is relevant or important.
2. This process or activity is broken down into its constituent tasks.
3. The relationships between the tasks are identified.
4. The average working time required to complete each task is estimated. Perfect precision is not needed. If sometimes it takes 3 hours to do something, another time 7 hours, and another time 4 hours, we might just estimate that on average it takes about 5 hours to get the work done for this task.
5. The average wait time between tasks in the process is identified. Once again, an estimate is appropriate since it is rarely the case that high levels of precision are possible or necessary.
6. Total working time, elapsed time, wasted time, and efficiency are calculated.
7. If any tasks are to be repeated, include the average number of repetitions (or cycles, as in the example above).
8. Optional: A "what if?" alternative can be created (a second such picture, sometimes referred to as a "future state" Value Stream Map). If one or more changes were instituted, what would the future state look like in terms of total working time, elapsed time, wasted time, and efficiency?

Do your very best to ensure that your Value Stream Map starts and ends with a real customer. Above all other aims, we are trying to do useful things for the people that buy and use our software. Value Stream Maps help us consider, "What is getting in the way of delivering value to them?" Also, Value Stream Maps should include all steps that contribute to delivering the customer value – even (or especially) when multiple teams are involved in the delivery. Often, waste collects at these organizational boundaries.

**Anti-patterns to look for when creating a Value Stream Map**

Beyond this basic example, let's consider some of the elements that could potentially go into creating a Value Stream Map. If our aim is to identify and end wasteful processes or activities, the
first step will be to identify one or more possible sources of waste. It can be hard to identify the most important organizational issues that need to be addressed. What should we look out for? For starters, consider the following anti-patterns that might exist in your organization, because they provide clues to help identify which Value Stream Maps may need to be created (adapted from a set of anti-patterns identified by Paul Gibson). In the background, you should hear this steady drumbeat for improvement: delegate, authorize, delegate, authorize.

Queues

On the surface, how frequently queues appear in an organization might not be obvious. But even an e-mail inbox can be a queue, and it often is. What if a team leader depends on e-mail as the notification that it is time to do a code review? What if an e-mail message is sent only to a test lead when a build is complete, rather than made public in some way? What if an employee has to wait indefinitely for a manager to approve a form for an ID to be produced? Queues are everywhere, and e-mail backlogs are only one possible anti-pattern to consider that can be categorized as queues.

Figure 2. Queues

You can assess the effect of any queue by looking at the minimum time items are in the queue, as well as the average time taken to process an item in the queue. If the queue occasionally goes to zero, then it is probably just handling variable workload. However, if the queue never goes to zero, the net effect is to delay every item passing through for that minimum time, multiplied by the average duration. Thus, for example, if you process one item a day and have a minimum queue of 20, the effect is to delay every item by 20 days.

Backlogs

Backlogs are very similar to queues, but the very name might help you spot them. Good examples of this are requirements backlogs. As a rule of thumb, use the idea that a requirements backlog should never be bigger than what you could complete in two releases of your software. If your product or project has a backlog of requirements that it would take 100 years to complete, that backlog is too big. What to do with all the rest of the backlog? Delete it. If it's important, your stakeholders will tell you, and you can do something about it.

Mary and Tom Poppendieck offer a more sophisticated and slightly less extreme approach to handling larger requirements backlogs. The following four steps are in their second book, Implementing Lean Software Development (see Resources):

1. Start by asking, "'How many things in this queue are we realistically never going to get around to?'" Cut all of the things you'll never get to out of the queue immediately. Be honest. Just hit the delete key.
2. So, how many items did that exercise get rid of? Half? Now take the remaining items and do a Pareto analysis on them. Rate each one on a scale of 1 to 5. The critical items will rate a 5.
The unimportant items will rate a 1. Now get rid of all except those that got 4s and 5s. Just hit Delete. Don't worry; if they turn out to be important, they'll come back to you.

3. Now take the items that are left and calculate how many days, months, or years of work they represent. Will you have other things added to the list that will be more important? With that in mind, do you have the capacity to do the remaining items on the list in the near future? If not, should you add additional capacity?

4. If your list is still unrealistically long, there is probably some purpose that it is serving beyond making effective decisions on what to do and what not to do. For example, a long list might deflect undue attention or absorb frivolous requests. Break the list into two lists, one which will serve the exterior purpose, and the other which you will keep short and work off of.

**Figure 3. Unnecessary review cycles**

Another key example of backlogs is the defect backlog. It is true that quality can never be sacrificed on disciplined software engineering projects. But if your defect backlog is enormous, is it even wise to necessarily fix it all? Might you potentially introduce lower quality by fixing many defects? With very large defect backlogs, in addition to wasting time managing these backlogs, it is possible that you are less likely to really see what is important. What if you did something radical like delete your defect backlog, with the understanding that going forward not a single defect could cross an iteration boundary? Would this be wise, possible, or workable? Regardless, backlogs need to be identified and either eliminated or greatly minimized. When backlogs exist, they must be necessary and manageable.

**Regular review and solution meetings**

Weekly status meetings, monthly management checkpoints, and quarterly operations reviews are all typically part of the software development workflow. Add to these the meetings held to prepare for other review meetings, and the list can appear seemingly endless. Take a step back and identify the various standing meetings in your organization's calendar. Which of these meetings are really needed? Which ones are bottlenecks? If approvals are needed (this in itself might be open to question), can the approval process be made asynchronous? Is it possible to shorten the cycle so that wait times drop? Remember that if a monthly review/decision meeting asks for rework and another review cycle, the net effect is to delay the decision by a month even if only a few hours of work are required.

Moreover, it is appropriate to consider the effectiveness of the meetings that are taking place if it's determined that the meetings are needed. Is there a clear purpose for them? Is a facilitator needed? Are actions and outcomes clear? Is progress consistently evident?

Brief daily stand-up meetings may be of help to establish a cadence that drives progress and identifies issues quickly on software engineering projects. When we introduce 15-minute daily stand-up meetings in a team, though, we ask them to do so only if they are willing to pay a price, namely: what two hours of existing meetings can be cancelled to justify the establishment of
1.25 hours of meetings each week that will be the lifeblood of the team? Generally speaking, the worst thing that an organization can do is to just add meetings without thinking about what can be deleted or substantially shortened. In addition, we insist that these daily stand-up meetings rigorously adhere to the 15-minute time limit.

**Callbacks**

Software support organizations often operate by means of a process that intentionally incorporates callbacks. The logic usually goes something like this:

- Anyone can answer the phone (as an aside, this is decidedly not the case). Pay someone with entry-level skills to handle such matters as license verification, confirm contact information and the like, and train them to answer the most basic questions and point clients to knowledge bases and other resources. Assign a priority or severity to the call. Queue up the trouble-tickets.
- Have another team with more skilled engineers handle these calls by priority or severity.

Quite apart from any frustration a caller might experience, is this the most efficient use of resources? Isn't it the case that more skilled engineers might be able to deliver value to your clients – problem resolution, in this case – much more quickly? If software developers and testers, for example, took support calls from time to time, wouldn't they be more attuned to the need to address common issues in the software?

The point of this anti-pattern is not restricted to trouble tickets. More broadly, where are calls coming into your organization, and how are they handled? The callbacks represent a kind of queuing mechanism.

**Flow chart shows multiple required approvals**

Because this article focuses primarily on the use of Value Stream Maps for software engineering teams, it is fair to note that if we hire professional software engineers, we ought to treat them like professionals. It is worth questioning every approval process that is in place. Certainly, appropriate checks and balances are valid, but two examples might bring out some of the potential problems with cumbersome approval processes.

**Figure 4. Multiple approvals**

Imagine an organization that approves hiring engineers for a team that is in another country. For the sake of simplicity, pretend that they are trying to hire engineers in Brazil, and the company's approvers are in the United States. Consider the steps that the team in Brazil might need to go through:

- Brazilian engineers interview various candidates and choose the best one.
• The manager of the Brazilian engineers must approve.
• The middle manager of the Brazilian engineers must also approve.
• The Brazilian country-level manager must approve.
• Human resources and finance in Brazilian must approve.
• Human resources and finance in the United States must approve.
• Technical management approval (at one or more levels) might also need to take place in the United States.

Not surprisingly, a process like this might take more than a year to complete. None of the approvers provide any real added value. Why not give the team in Brazil a clear budget and headcount target, and let them manage it?

Another example may prove helpful. Imagine a software product that is not sold solely as an individual product, but as part of a larger solution – some of which have many possible moving parts. What happens if you have to make an architectural decision? Consider the simplification that the illustration Figure 5 shows.

**Figure 5. Architectural decisions**

Imagine that you are an architect who works on **Product A**. The team that you are a part of wants Product A to work with Products B, C, D, E, F and G. It might be the case that Products B, C, D and E are built by another part of the organization, and Products F and G might be built by still another organization. It might also be the case that Product G is very important to your company's revenue stream.

You have two basic choices with respect to how architectural decisions could be made. Perhaps every quarter, meetings could take place where decisions are made about how everything should work together. The key problem with this kind of approach, though, is that the infrequency of these meetings mean that Product A might have to wait a long time to make a key decision about how to build that product. This might make it difficult to ship Product A with high quality on a short schedule.

In practice, we have found another option that works well. If we treat the architect on Product A as an intelligent engineer and encourage this engineer to make decisions based on their best judgement (so that the team is not held up) and then communicate with the whole team about architectural decisions off the critical path, this kind of optimistic decision-making means that about 95% of the time, the architect gets it right. What about the 5% when they get it wrong? It's not
the architect's fault; the team decided this approach worked best. We have found that the rework needed for this 5% when changes need to be made to the architecture is well worth the overall speed gained. It should also be noted that the 95% success rate might be higher than teams that waited for approval.

**Parallel activities with integration costs**

The reason that "continuous integration" is a central mantra for many lean and agile software engineering practices is because integrating large swatches of code is really hard to do. The longer that scaffolding code is in place, and the longer that assumptions are left unchallenged, the more severe integration challenges will be.

**Figure 6. Bottlenecks caused by parallel activities**

Using the flow chart in Figure 6 again, it would be bad enough if Product A was developed in such a way that builds were done infrequently and components or modules within the product were integrated only sporadically. But imagine if this was compounded by the fact that integration with the other products in the larger solution was being "deferred." Inevitable problems would be uncovered late in the process, and quality issues would be likely.

In addition to continuous integration being done for Product A, and hopefully for each of the products in the solution, frequent builds of the larger solution are necessary in this way, as well.

**Long test cycles**

Deferring testing until the end of your development process is always expensive. The greater the time between when a defect is injected and when it is introduced, the harder it is to correct. This is why practices such as meaningful unit test, test driven development, pair programming, code reviews and inspections, test automation, and many others, are so often emphasized in lean and agile circles help to eliminate waste.

If your Value Stream Map reveals long test "cycles" in the end game, it's very likely that you are guaranteeing wastefulness. What can be done to help ensure that functional, customer-like system tests are run throughout development iterations?

Even if your team does reserve a long or late period for system test, it is important in a Value Stream Map to distinguish between the time spent doing actual testing and the time fixing code. It is not uncommon, for example, for a large test organization to be able to complete all of their tests in a week or so, assuming that the code is genuinely stable. If you have a 13-week system
test cycle, could it be that much of this time is really for defect repair more than for testing and that a focus on earlier defect removal could significantly reduce both the system test cycle and the overall cycle?

Similarly, long test cycles might be symptomatic of overall sub-optimization of your development process. If they exist, could it be that analysis of the organization needs to take place to consider how to maximize overall flow, rather than maximize the organization's ability to generate untested code?

Linking the value stream to your customer at both ends

Often, teams will talk about creating Value Stream Maps that consider an internal organizational flow. Ideally, as often as is conceivably possible, a true external interested customer should be identified at the beginning and end of the Value Stream Map.

For example, it may be the case that your team is reviewing a Value Stream Map to understand how long it takes to implement a new requirement that comes from an external customer source. Very often, teams will consider the end of this Value Stream to be the time of product or project shipment. In fact, though, the end of the Value Stream Map should identify what it takes to get that product or project implemented in their production environment, where that same customer is deriving real value from their original requirement. If you are creating a Web-based application, it may be that your customer will be in production as soon as you post the changes to your code. In contrast, if you are producing shrink-wrapped code into an IT shop, deployment considerations could be of paramount concern. In this case, a whole series of activities that will take place after shipment, which are in fact part of the Value Stream, can be considered as possible areas of waste and are candidates for optimization by changing how activities are completed earlier in the cycle.

Reasons to create Value Stream Maps

There are three primary reasons to create Value Stream Maps: objectivity, clarity, and persuasion.

Emotion and opinion have their place. But questions about potential organizational processes and activities that need to change – which is what Value Stream Maps are targeting – are best addressed dispassionately. Let's consider the very basic sample Value Stream Map used at the beginning of this article. Assume that you go into the office of a decision-maker and say, "'It would be more cost-effective and more satisfying to our customer if we were to travel to their offices the next time that we need to sign an agreement.'" It is far too easy for a decision-maker to provide a default response: "'Travel is not approved at this time.'"

If, instead, a simple Value Stream Map is presented, this same point would be made, but the problem would be clear, and the potential savings and the possible benefit to the client would be far more obvious. What if you were to create a second Value Stream Map — a future state Value Stream Map — that sets this contrast clearly in view: "'With travel, it would cost us $1,200 to complete this agreement, and it would be done in two days. Without it, we will spend $13,500 (based on headcount or other costs), and it will take a month and a half.'" Might this not give your argument more of a fighting chance? Although the outcome might still be the same (that is, the
travel is not approved), the likelihood that cost savings might be realized if travel is approved can be considered more objectively. Rather than arguing or ruminating over the decision, you can provide justification for appropriate exceptions.

One final example might prove helpful: How a customer request handled. Study the hand-drawn diagram in Figure 7.

**Figure 7. Value Stream Map, implementing a customer request**

Examine the flow here for a moment. A number of points stand out in this example. First of all, it’s not pretty. If it were converted into a crystal clear graphic by using Microsoft® Visio® or some other tool, would it change the content? Not really. The easier it is to create Value Stream Maps, the more likely your organization will routinely use them. Consider also some of the features of this particular Value Stream Map and, very quickly, you can draw some important conclusions:

- The overall development approach being used by this particular team appears to be essentially a waterfall process.
- Although the graphic contains only wait times and not the work times associated with these tasks, some questions stand out about the work time, regardless. In producing the work times that are associated with these tasks, it might well be the case that the tasks themselves are hiding waste. Some of these tasks (such as "Develop / Document / Test") will need to be broken out further to ensure that waste is not embedded.
- There are a lot of approval cycles here. Are they all valuable?
- There are no loops here, which might be because the graphic is a simplification. That said, how often are some of these approval cycles repeated? They could be sources of additional waste.
- Positively, it’s good to see that this particular process (a feature request) both starts and ends with a customer. As we’ve considered, it’s too often the case that some teams think that
because they have shipped their software or finished their work in some other way, they have finished their jobs. The customer, though, still has to deploy their software in many cases, at times into complex environments.

Many more points could be made about this particular Value Stream Map, but fundamentally, the key issue is that everyone concerned can take a step back and think about the flow of a feature request through the organization. It should then be comparatively easy to suggest that the biggest area of waste should be addressed. It wouldn't be a lot more work to break the tasks out further to an appropriate level of detail and to calculate the overall working time and the overall wasted time, as in the first example. And then, very likely, it becomes clear what needs to be addressed first – and why.

**Summary**

Value Stream Maps help us bring about organizational improvement, progress in our processes and methods, and most importantly, better software. Value Stream Maps can help both identify and stop waste in an organization. By leveraging the lessons explained in this article, your organization can find greater efficiencies and deliver higher-quality products to your customers.