How’s your project going?" Isn't this the first question

people ask? It's natural and necessary to have some idea how a project is proceeding. Some people want to know if it will be done on time. Others have fixed the delivery date and want to know how much functionality the system will have. Some people are looking for an early warning sign that changes need to be made. Everyone wants to peer into the future for reassurance that they'll get what they want.

Burn charts are a simple method to monitor the progress of the work. They provide an easy to comprehend, visual representation of project progress. They can be visually extrapolated to make predictions about delivery date for fixed scope, or scope for a fixed delivery date, as shown in figures la and lb. If this

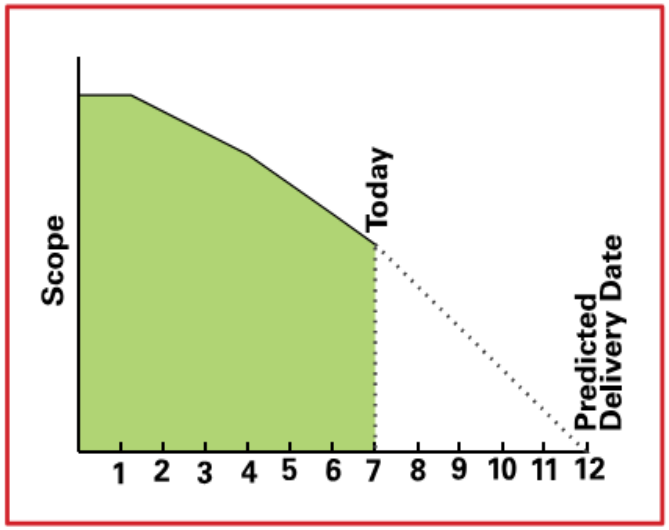

Figure 1a

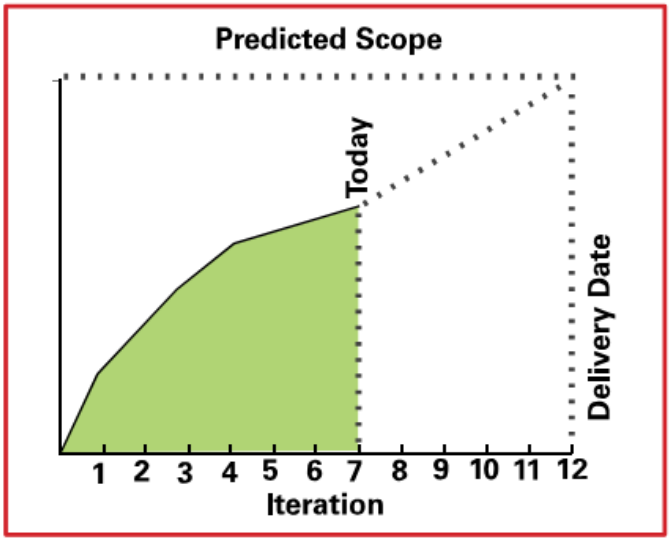

Figure 1b

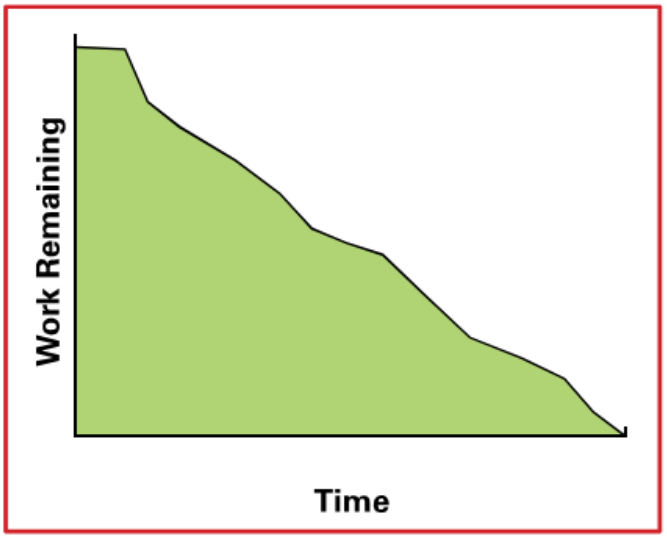

Figure 2

visual extrapolation is difficult because of the shape of the line, then the chart is telling us other things about how the project is going. This may be a diffi- culty in delivering working software or a difficulty in deciding what the soft- ware should be. We'll take a closer look at some of these scenarios a little later. First, we'll examine some of the variations of these charts.

Measuring Work Remaining

If you've got a shovel and you're spreading a pile of dirt, the size of the re- maining pile shows you how much work you've got left to do. As you take shovelfuls off the top of the pile and spread them around, the height of the pile goes down over time. If we plot the height of the pile over time, we'll see something like figure 2.
This is an example of a burn down chart. At each increment of time, we plot the work remaining. We can see the progress toward our final goal, which is to have no work remaining. In effect, we “burn down” the planned amount of work until there's none left. If the rate of progress is at all consistent, then we can easily predict when the work will be fin- ished. If we've got a fixed deadline, we'll see if we're going to meet that deadline.

Ways of Measuring Work

There are various ways of measuring work. For a physicist, work is measured in joules. If your manager asks how much work is Leif to do on your project, and you reply with a value in joules ... well, let's hope your manager has a sense of humor.

BY HOURS, DAYs, WEEKS

A more common way of measuring work is with units of time. I do eight hours of work for a day's pay. This is a simple and easily understood measure. It's just that it does not measure what you may think it does.

Suppose I start work on time but then go for coffee while waiting for my com- puter to boot up. In the cafeteria, I run into my boss, and we spend forty-five minutes talking about company policies. After that, I log onto my email to catch up. Some of the emails require detailed responses, and it takes a couple of hours to collect the information. Now, down to work. I start up my IDE, start up my local server, re-deploy the code I'm writing, and start the application. I try the scenario that my current user story addresses. Oops, time for lunch. Right after lunch I've got a two-hour meeting. OK, where did I leave off? Oh yeah, here it is. I start to code, but I notice people are leaving. The workday is over and I've done eight hours of “work,” but I haven't accomplished very much.

Of course, not every day is this bad, but time has an amazing tendency to slip away unnoticed. Not every hour corre- sponds to the same amount of work. It doesn't take much for hours—or days, weeks, months—to become a highly in- accurate way of measuring work.

People are not very accurate in making predictions, particularly about the future. If we're doing a repetitive task, such as spreading dirt from a big pile, then we might estimate how long it will take to spread the rest of the pile based on the amount we've done so far. Or, if we're doing a task that is essentially the same as one we've done before, we use that experience to estimate how long this instance will take.

With a complex task such as soft- ware development, we've got bigger problems. The work doesn't proceed at a steady pace. We're zipping along and suddenly run into an issue that stops us in our tracks. There's something we don't know, and we have to figure it out before we can continue. This makes a mess of our prediction for how long the task will take. With no apparent prog- ress while we are discovering, we can't know when the stuck period will end.
This doesn't mean that you can't succeed by estimating in hours and days. By estimating in time units and checking your actuals against your estimates, you will get better at estimating. At the indi- vidual level, there's some merit to that. But with a group of people, the chance of getting really good at estimating is lessened, the cost of doing so is in- creased, and the payoff is uncertain. The goal of a software development team is building valuable software. It's not clear that better estimating will increase either the quantity or quality of the software produced, but it is clear that the estima- tion ability will be disturbed every time there's a change in the team.

Adherents of Scrum recommend that the team estimate each task and then daily re-estimate the remaining amount of work on each task in order to calculate the total hours of work remaining. [1] This may provide more precision, but it creates a lot of overhead work and doesn't produce the software any faster. And, by taking up time and energy, it delays delivery.

By Story Points

I recommend a different approach divide the work into user stories small but functional slices o the application (see the StickyNotes for more information). Assign each user story a simple relative estimate.
For example, assign each of the simple stories one point. As- sign the stories that seem about twice as hard two points. Estimate your work in story points and track the number of story points left to do. You don't “get credit” for a story point until it is com- pletely done—coded, tested, acceptable to the customer, and ready for potential deployment. This gives you a reliable in- dication of work accomplished—what you really want to know—rather than effort expended.

Don't worry about improving your estimating accuracy. In a large project, there's probably enough random pertur- bation that you'll never achieve the ac- curacy you would like. Remember, our desired output is working software, not accurate estimates. The estimates just have to be consistent enough to use for planning purposes. While our estimates may be off by 50 percent or even 100 percent, if we expect to accomplish the same estimated amount next iteration as we did this iteration, we'll be on target.

Variable Scope

The simple burn down chart uses the zero point of the y-axis as the goal line. This works fine as long as the goal doesn't change.
If you don't keep the scope of work constant, then from time to time you're likely to get a burn down chart like that shown in figure 3. What happened here? Why did the amount of work remaining increase rather than decrease? Did more work get added?

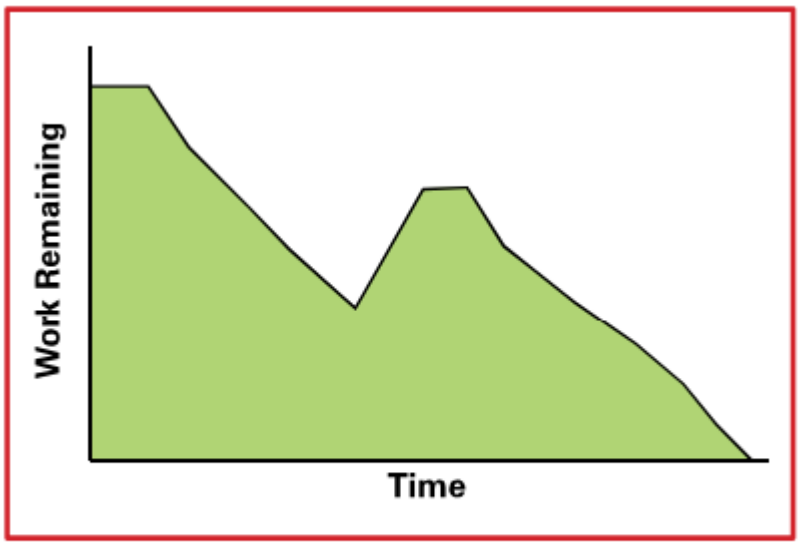

Figure 3

Within a single itera- tion, this shouldn't happen but some- times does. Over a longer period of time, it's generally expected that new work will be added.
Did unforeseen tasks come to light? If you're burning down hours and re-esti- mating the remaining work on each task, as frequently recommended in Scrum books and articles, then you might get such a thing.
Did a story move backward? Perhaps it was thought to be done, but a new scenario was discovered and the story was handed back to the developers for more development. Or, perhaps the tes- ters found a bug that had escaped devel- opment.
When the burn down chart curves upward like this, you no longer can pre- dict when the work will be done. If the scope changes, we can no longer trust the goal line. If the measure of work remaining proves unreliable, then we're losing the ability to predict when the current scope of work will be done. We can't extrapolate the burn down chart to the goal line.

Burn below Zero.

One way to separate changes in scope and progress is to use a burn down chart with a variable floor, as shown in figure 4.

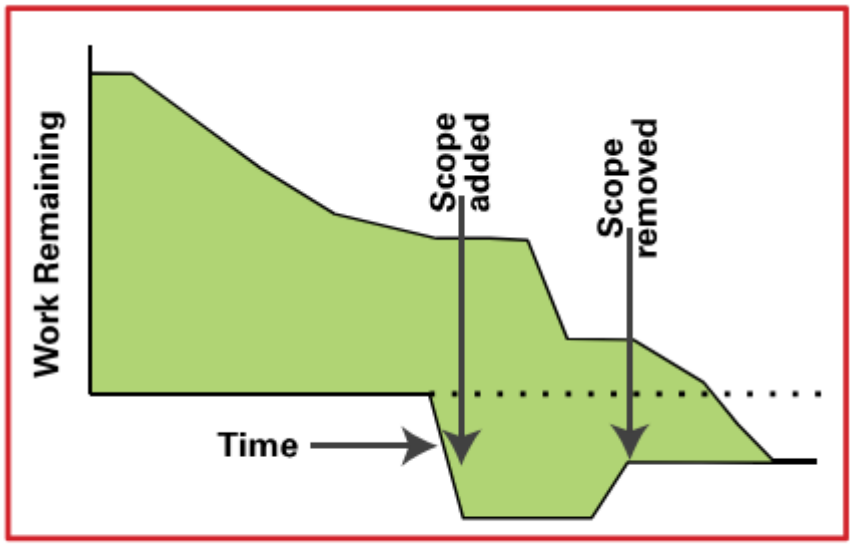

Figure 4

In other words, the work remaining is plotted as before, but the goal line may vary from the zero point. If scope is added, the goal line moves below the zero point. If scope is removed, the goal line moves up. We can see our continued progress, but the intersection with the goal changes. Or, to reach completion sooner, we may reduce scope and bring the goal closer (see the StickyNotes for more information).

This seems to me a little compli- cated to draw, especially if the scope changes very much. It also presupposes that you know the goal line at the very start, which I find difficult to do. I gener- ally find that the addition of new story points continues well into development, if not to the very end. I prefer not to use this chart, but people have used it effec- tively when the scope of work is changed during an iteration. I also prefer not to do that, but there are times when it's appropriate—when the team has under- estimated and is running out of things to do, the team has overestimated and clearly needs to cut scope, or an emergency occurs.

BURN UP

A good way to indicate variable scope is to use a burn up chart. [2] This is like a burn down chart turned upside down. Instead of tracking work remaining a burn up chart tracks work completed. The goal line can be moved, and the dif- ference between it and the work completed will give you an estimate of the work remaining. In the burn up chart in figure 5, you can see the addition of scope over time. Periodically, the goal takes a step upward. Each of the upticks in the goal line may represent a major feature's being defined as stories and estimated. Or, the upticks may indicate scope that was assumed but is now being made explicit. The scope of work also may be adjusted downward to meet a particular release date or because the least important functionality is being trimmed.

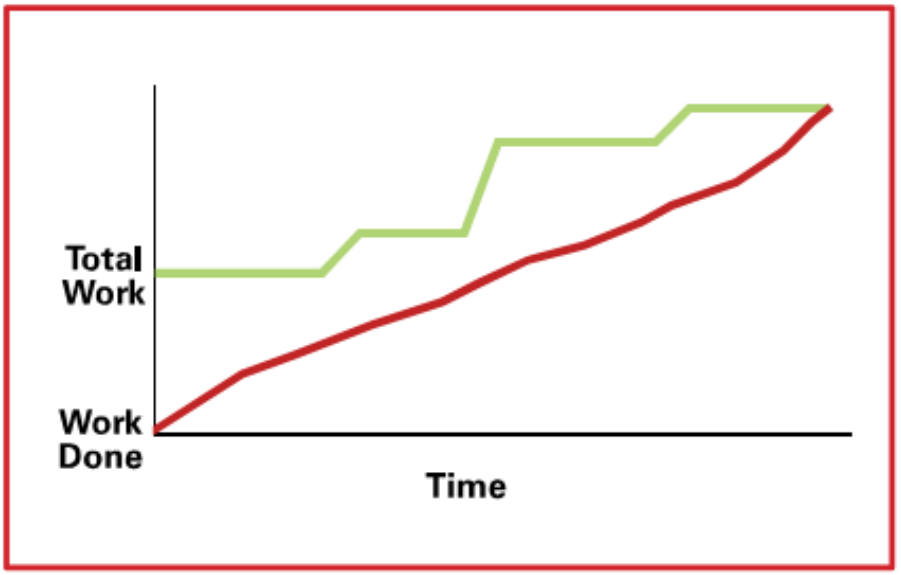

Figure 5

For a fixed scope of work, a burn up chart doesn't match the simplicity of a burn down chart. But for variable scope, as most projects are over any extended period of time, the burn up chart gives a clear indication of the progress so far and a visual prediction of the finish date. My preference when doing iterative software development is to use a burndown (burn down) chart for an iteration, which has a short time span and a fixed scope, and a burn up chart for release planning or other longer-term project planning, as shown in figures 6a and 6b.

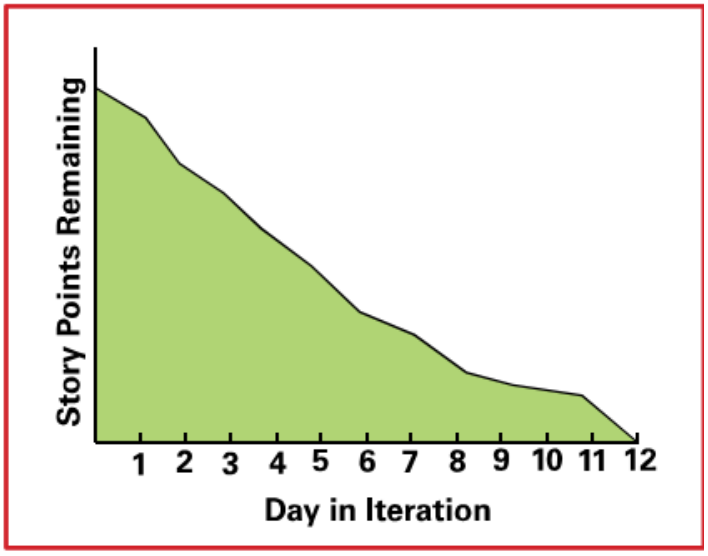

Figure 6a

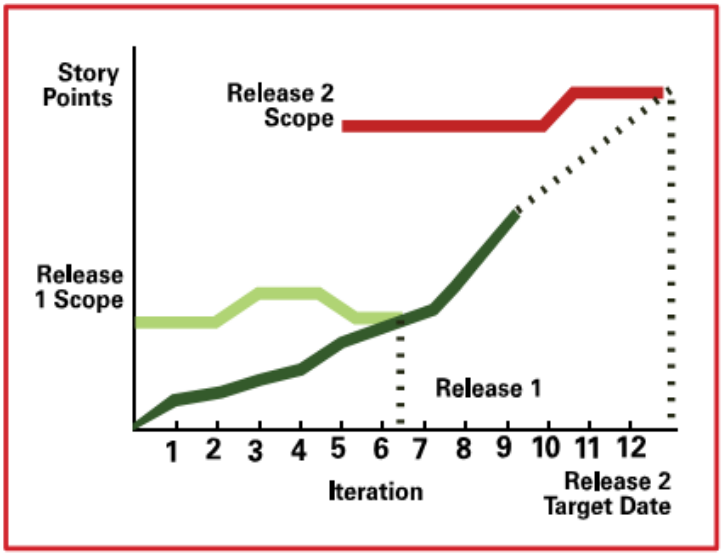

Figure 6b

Reading the Charts

A burn down or burn up chart does more than track progress. By examining the graph, we can make inferences about how the work is progressing. Figures 7a through 7e show some examples of burn down charts tracking progress in iterative development.

Too Much WORK

Figure 7a shows a burn down chart that doesn't reach the goal line. At the end of the iteration, there's work left un- finished. This may be an indicator of an unusual problem in this iteration, but if it happens frequently, I would interpret it as a sign of overcommitment.

Frequently, software development teams are overly optimistic about their capabilities. They know how to do the things that they see need to be done. Often they don't remember or consider the times they got stuck, either on a hard problem or waiting for some external dependency. They may ignore the time spent doing things other than

creating code. And when they come up a little short at the end, they may commit to even more work during the next iteration to catch up.

SANDBAGGING

Some software development teams go in the other direction. Not wanting to miss their commitments and disappoint their stakeholders, they are conservative in their commitments. Figure 7b shows a steady progress and then slacking off when reaching the goal seems assured. Or, this chart could mean that the later stories were estimated low, relative to the work they required. Or, it could mean that technical debt incurred in the earlier stories slowed down the development of later ones. It's often the case that a single graph of the iteration could be telling one of several stories. You may need to look at other information to clarify your understanding.
Another possibility: A team that never misses its commitment is a team that isn't pushing the boundaries. A high-performing team should always be testing where the line is between too much and too little work. This is much like a sailor who heads closer to the wind until the sail just starts to luff and then bears off to achieve maximum speed. The wind and waves are always changing, so this test needs to be done repeatedly and continually. So it is with software development. Try to push for just a little more, but be sensitive for signs that you're tempted to cut corners or leave something not quite done. Then, back off slightly.

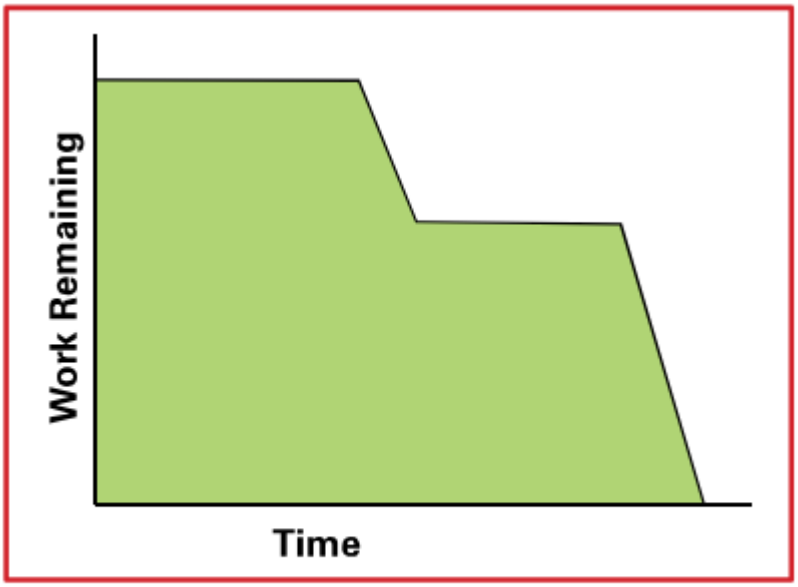

Figure 7c

THE RULE OF THREE

Gerald M. Weinberg, in his book, The Secrets of Lonsult- ing, states the Rule of Three:
If you can't think of three things that might go wrong with your plans, then there‘s something wrong with your thinking. This rule is applicable to much more than plan- ning. It particularly applies to reading burn down charts. The possibilities that Ioffer are common ones I've seen, but they are offered without
any knowledge of the context of your project. You‘ll need to discern what your burn down charts might be telling you. After you have an answer, try to think of at least two more. Then, armed with these pos- sibilities, try to check it out to see which, if any, reflect the reality around you.

STORIES TOO BIG

Figure 7c shows a picture where there's no indication of progress for a long time; then, suddenly, there's a big chunk accomplished all at once. The most likely cause for this is “big sto- ries.” Often, teams accustomed to a serial project lifecycle—with requirements definition, design, implementa- tion, and then testing—have a difficult time breaking work down into small but valuable stories. Instead, they do “mini-waterfalls” on larger chunks of work, and they break the work down by architectural layers instead of by functionality. As a result, they bundle a lot of work together into one unit. Doing work in large chunks makes real progress hard to see. Some people try to counter this by estimating progress on unfinished pieces of work. It's easy to fool yourself about the amount of

progress, though. There is a danger of creating the old situation of having done 90 percent of the work but the 10 percent remaining takes 90 percent of the time. It's much more reliable to judge progress by functionality that's easily tested to be complete or not. I like to see progress every day. Otherwise, I think I need to look for a bottleneck that's halting progress. This is just a small instance of developers “going dark” and no one else knowing what's happening.

Large stories also prevent the product owner from effectively steering the development of the product. Such a story may contain many little details that are “nice to haves” rather than “must-haves.” If they're all lumped together with must- have functionality, the product owner must accept or reject the story alto- gether. This will mean putting these nice to haves into the product ahead of must haves in other stories.

What really works, once you learn how to do it, is to create very thin slivers of functionality. I call these vertical slices because they slice through the architectural layers from the user interface all the way down to the database. (This is the typical description of layers for business systems, but other types of software will have different layers.) This has been called “a walking skeleton” [3] or “firing tracer bullets through the application.” [4] Each thin sliver doesn't do much, but it works all the way through. In addition to these slices of functionality, there may be stories that only modify existing functionality.

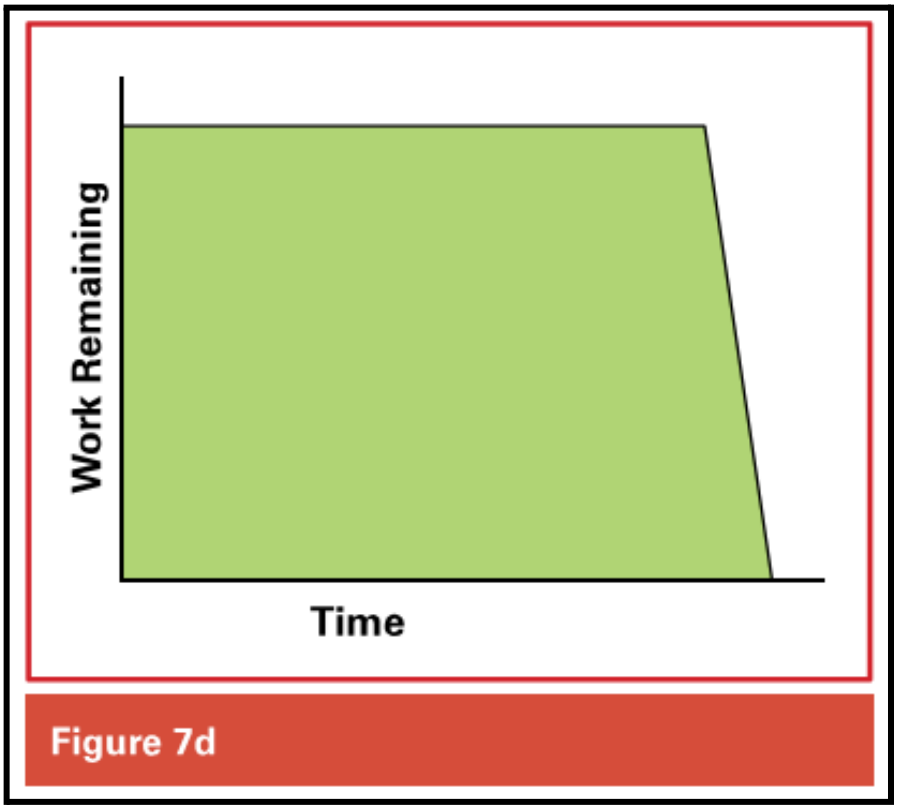

Figure 7d

”Unfinished work has a cost (the work done so far) but no value (we can’t deliver it to the customer). It's far better to finish a small amount of work than to start a large amount”.

BIG WORK IN PROGRESS

Figure 7d could be telling us that there's one huge story, the worst case of “stories too big.” Or, it could be telling us that all of the stories are being worked simultaneously. Either of these is an indi- cation that a lot of work is in progress at one time. Another alternative, that the developers were doing other work rather than the requested stories, would also produce a burn down chart like this.

What's the problem with this? One issue is that you're not sure of com- pleting the stories. What if you reach the end of the iteration and all of the stories are “90 percent done,” but there's no working business value to deliver to the product owner?

Unfinished work has a cost (the work done so far) but no value (we can't de- liver it to the customer). It's far better to finish a small amount of work than to start a large amount. By dividing the story into small, functional slices, we can see the progress being realized. If we then start work on all of those small sto- ries at once, we lose that advantage. It's better for the team to swarm over each story to get it done and then start on the next one. The goal should be to have as few stories in progress at a time as you can work on productively.

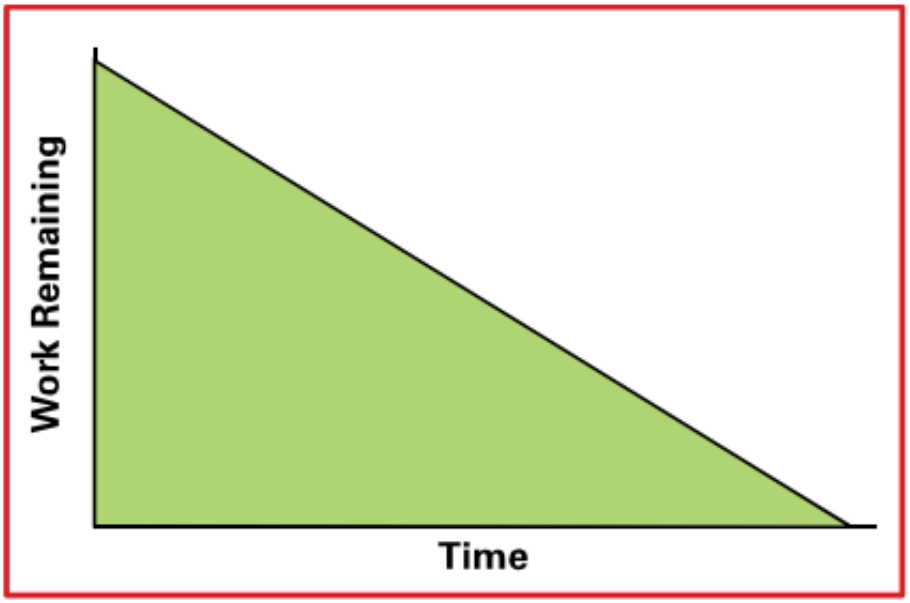

Figure 7e

COOK THE BOOKS

One of the most insidious burn down charts is the one where the progress line is arrow straight from start to finish, as shown in figure 7e. While this could happen occasion- ally, it most likely means that someone is cooking the books. He may be reporting the progress that he perceives people want to see rather than an honest mea- sure of how the project is proceeding. This is especially easy to do when tracking progress in task hours rather than story points. It also can be done by cutting corners on stories and claiming they're done when the code is still a mess internally.

Why would this happen? There may be a number of contributing factors:

- The person (or software) drawing the burn down chart may mark a line to be able to tell easily if de- velopment is lagging behind or getting ahead of the average ex- pected rate. Such a line can be in- terpreted by others as the “ideal” rate.
- The team may feel that it is being judged on the basis of the chart, so team members want it to look good.
- It may be that progress is being measured more by effort than by accomplishment. Systems that automatically calculate time re- maining on a task are prone to this.

When a “good looking” burn down chart becomes the focus, then the ability to use it to understand and manage de- velopment is lost.

It's Your Choice

Burn down and burn up charts offer considerable variety in the ways they can display progress so it can be understood at a glance. They're highly adaptable to your context—you can track work with whatever measurement makes sense in your environment. You can track it over short or long periods of time or both. Do you have a fixed scope? Then a burn down is a good fit. Do you have a well- understood scope with relatively small adjustments? Either the variable floor burn down chart or a burn up chart would be appropriate. Are you working in a continuous flow, continuing to add to the backlog as you develop? Then a burn up chart is probably the best choice.

The important point is that your burn chart should reflect an objective reality, not wishes and hopes. It should be built with data that is measured, not estimated. Post your chart prominently where you will look at it every day. In this way, it can tell you at a glance whether your wishes and hopes are likely to be realized. If it tells you something else, it will give you the clues you need to take some corrective action.