Managing Capacity and Workload

7

Or How to Finally Acknowledge that You Can't Fight the Universe

Managers and Workload

From my own perspective, Agile managers shouldn't be the ones caring the most about workload and team capacity, especially in the short term. If we use Scrum terminology, that is the product owner's role. Managers should give some guidance about operational and strategic priorities, maybe in the form of resources to use for each project or relative importance. But when managers start to arbitrarily set deadlines, scopes, and budgets without consulting the team or the product owner, very bad things happen.

The urge to know about team's capacity, when projects are supposed to be delivered, expected delays, and final scope contents of next releases is too pressing to be resisted most of the time. Furthermore, the temptation to push some last-minute requirement to the team in order to please your client, or maybe your top management, and tell them to "just do it!" will also be very appealing, as they will very seldom fight back, especially if we are talking about teams that have been living a Taylor-Ford culture for a long time and are new to the details of Agile.

On the other hand, with successful Agile implementations at the team level, I very often find desperate managers telling me "they were supposed to deliver in December, and now they say that it's not going to be done before February! And I am not supposed to interfere! Isn't this Agile thing killing us?"

Then we have the first-impression shock of some Agile implementations: when you stop counting work-hours per week[1] and you start focusing on

[1] Let me guess, is everyone reporting 40 h of work-related stuff per week? I have bad news for you.

Á. Medinilla, *Agile Management*, DOI 10.1007/978-3-642-28909-5_7,

delivered value, maybe you'll realize how few things you are accomplishing. And then fear will strike management, who will irrationally want to go back to the old days, where they didn't feel so uncomfortable about what they were finding out.

Unfortunately, for them, one of the characteristics of a Kaizen or continuous improvement environment is "a perpetual state of discomfort." And I promise you a whole lot of it while following the Agile path.

The fact is probably that you were not that good before Agile. On a few occasions, I've found someone who was in absolute denial and stated that they *never* had any kind of delays before Agile and that *all* projects were delivered on time and on budget. As you can imagine, this was plainly not true and was just the reaction of laggards to a change in status quo.

What's worse: some people I have met claimed that, before Agile, if a deadline was to be met, people would drop vacations, weekends, and even work nighttime until the project was ready and that now they don't see "that kind of commitment."

The truth is that, of course, you had several project delays before Agile. Usually they were hidden by unpaid overtime, which was introduced by fear and managerial pressure and was, of course, killing both motivation and product quality: even the most Tayloristic manager will have to admit that the quality of products coded on a Saturday night after 45 h of continuous coding won't be very reliable.

This unwanted and pernicious situation is creating vicious cycles in ways many companies fail to see. Continuous pressure and hurry make people deliver low-quality code as long as it is working – more or less. This introduces bugs and errors that will have to be handled in the future, thus reducing productivity through support time and context switching. The reduced productivity will delay projects, which will again introduce hurry and pressure, thus closing the circle. But it does not stop there: the low quality of products will make clients angry, introducing more pressure into the circle, and we will even lose some clients, which will reduce profit. Profits will also be reduced because of low productivity, and low profits added to a continuous urgent situation leave no time for developers to train, learn, or invest in reducing the technical debt. This will de-motivate software developers, making them less productive, thus reducing even more the productivity and the quality. And the circle is closed again.

This kind of systemic problem, and not just pushing work into people and pressing team deadlines, is the Agile manager's domain.

The Sales Lunch Syndrome

Sometimes, the problem doesn't come from team managers or product owners but from some other part of the organization, for example, sales. Very often, sales people have goals and bonuses related to the number of sales they make. So, for them, this is the main drive and they won't care that much for other metrics like product quality, project profitability, client satisfaction, returning clients, and churn rate. These will be seen as problems for the production or marketing areas but not for them.

So they will sell just anything, and increasing prices or delaying projected delivery dates will not help them do so. Very frequently I have experienced the situation of a crisis in a project whose deadline was set by the client and the sales guy over a copious business lunch with all sorts of spirituous drinks. When the team was not able to meet that deadline, the sales guy was still seen as a hero by the company, as he was bringing new clients and selling projects, while the team was labeled as "not committed enough" and "incompetent" for not meeting the arbitrary and far-more-than-ambitious deadline.

I have many friends who are sales people: it is a tough and difficult job that requires a lot of unusual skills. I have needed to do it myself when creating my own consulting company, and I have enormous respect for sales people. But many of them will secretly admit that if they push unrealistic deadlines to teams, they sell more and also feel like the team will work harder than if you set a more realistic deadline.

Of course, what we have here again is a problem of suboptimization. As everyone is playing only by their own rules and goals, the game sales people are playing is damaging the ability of the company to perform as a whole. A similar situation led to the subprime mortgage crisis of 2008 and the subsequent market breakdown and global crisis: mortgage sellers were paid according to how many mortgages they sold, and they were not concerned about the reliability of the buyers, their solvency, and if those mortgages were going to be repaid at all or not. The rest is history.

The "sales lunch syndrome" can happen in many other ways. Managers can set arbitrary deadlines according to special market dates (fairs, congresses, client meetings, quarterly reports), or maybe clients themselves

will set an unrealistic deadline according to their own needs, and the company will accept it because they fear that if they discuss it, they will lose the contract.

The fact is that the sales lunch syndrome is just another attempt at brute-forcing the system. Try to look at the project from the future: yes, you set a deadline of 6 months; yes, the team accepted the deadline; and yes, 1 month before the deadline the team said that they were not going to make it, and hell was unleashed until the project was delivered on month 9. But if you know that the team did its best during those 9 months, the truth is that it was not possible to deliver the project in 6 months and the mistake was to accept that deadline. If you feel that, looking at it retrospectively, there was a way to deliver the project in 6 months and you didn't realize it, go change your system so the next time you'll take advantage of that strategy. But if your only conclusion is that "the team should have worked harder," "they were not committed enough," or "you needed more resources," you have big chances of brute-forcing the system in a Tayloristic way.

But You Promised!

I have bad news for you: no Agile, Lean, or other kind of method, process, framework, or tool will release you from the responsibility of an adequate customer management. That includes setting a correct context with clients in terms of scope, time, budget, quality, risks, and other project terms and rules for managing the project and dealing with uncertainty, and also managing their expectations.

If you have done your homework, by now you should understand that software development is not a predictive process, so setting a deadline is fine as far as you leave some constraints variable. If you reduce and simplify the project space to scope, budget, and time, that means you can fix one of them, or even two, as long as you leave the other variable. For example, you could set a deadline and a budget and say "by the end of November we will deliver something at this price – but we are not sure if it's going to have 80 or maybe 100 features; what we promise is that it will have at least 80 features, and that those will be the most important on the list."

But if the feature set is fixed, then we have a problem, because fixing a budget, for example, will also fix time, as most of the software development costs are time based. We could fix feature set and time, leaving budget open, and then see if putting enough people into the project can make the deadline

feasible – which may be still wrong, as there is a minimum amount of time and money needed to build anything, and putting more people into the project may not reduce the deadline appreciably (remember Brooks' law!). On the other hand, fixing the feature set in advance has proven not to be an advisable way of developing software, as clients are known to define what they actually need when being able to actually use the product.

> "The best way to achieve predictable software development outcomes is to start early, learn constantly, commit late, and deliver fast. This may seem to cut against the grain of conventional project management practice, which is supposed to give more managed, predictable results. But predictability is a funny thing; you cannot build with confidence on a shifting foundation. The problem with conventional approaches is that they assume the foundation is firm; they have little tolerance for change."
>
> –Mary Poppendieck, *Lean Development and the Predictability Paradox*

In fact, I have breaking news for managers: predictability is not the point! Lean managers moved long ago from the market-prevailing paradigm of being able to predict delivery dates, future capacity, and resource needs to a change-and-adapt way of operating, and now they rule their markets. For them, continuous improvement, increasing value, reducing waste, and letting the system flow nonstop are the main targets of their managerial duties.

To reach that state, Lean companies moved from big to small batches of work, thus reducing uncertainty. So maybe reducing the size of your projects is a good way to start coping with deadlines and predictability. Committing to an iteration is easy and holds a controlled amount of uncertainty that can be soon determined and used to predict next iterations, while committing to a 12-month-long project will very probably move uncertainty as close to the deadline as possible, making it impossible to steer the project when we find out that the original estimates were too optimistic or unrealistic.

Seeing how complex the predictability of software project is, it seems like someone using the "we promised!" argument to steer a project is being unreasonable, non-educated, and childish. It reminds me of the typical conversation that parents have with kids, where they will say, "look son,

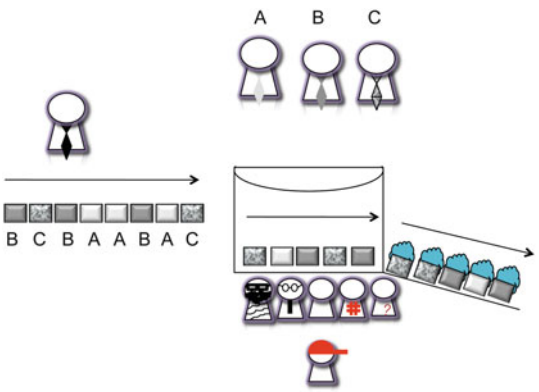
I know I promised to take you to the amusement park, but it's raining badly," and the kid replies, "but you promised!." In this case, the father failed to set the context: "I promise that, if weather and other circumstances allow it, I'll take you to the amusement park." And the process to follow if the circumstances were adverse was not set either.

Yes, clients will complain when a deadline is not going to be met. But brute-forcing the system, as we've seen, is not an Agile option. Be prepared to explain in detail the reasons for the delay, as if they are reasonable we should expect the client to understand them, especially if you provide a way to deal with changes and have something to deliver on the promised date – even if it does not feature all the promised functionalities.

Anyway, complaining, pressing, and demanding is the inherent nature of clients and, as we said before, no method will substitute a good client management approach.

The Two Good Moves and the Three Bad Moves in Pipeline Management

Let's try to simplify demand and see it as an incoming pipeline of small batches of work (features, for instance) that are prioritized and assigned to the team, as in this graphic:



In order to use a pipeline approach to portfolio management, you must be sure first that you have feature-cross-functional teams, so every batch of work is self-contained and can be delivered by any team as "working software" at the end of an iteration. You must also have capable product owners who know how to break big requests in small batches that can be

independently scheduled in a determined order. If that is the case, maybe you can use a Kanban approach for your whole portfolio management, creating a "Backlog" column and a "Selected for Next Iteration" column, where the next features to be developed will be always specified.

Considering all this, if you have an average velocity of five features per iteration, and as long as the product owner is doing a good job dividing projects in similar-sized batches, you can have a fairly good predictability on when each feature of the backlog is supposed to be delivered and also update that information frequently according to the team velocity measured at the end of every iteration.

Now imagine that the first client, A, has an urgent request of three features. What can the product owner do? Well, for me, he has five options: two of them are good, two are bad, and one final option is directly dangerous.

The first good move he could make is to accept the request, put it at the end of the queue, and promise the client that as soon as there is some available space in a still-noncommitted iteration, his three features will be the first to be scheduled. Of course, when I tell product owners or sales people to do this, they look at me incredulously as if I'm mad. I can't understand why, as this is, most of the time, the best thing you could do. Everything in the pipeline was a priority several days ago, so now that today's priority arrives, do the former stop being priorities? Even worse, if we delay the whole pipeline to attend this new urgent request, we won't be able to deliver other features on the promised deadlines.

What we experience here, again, is a case of instant-reward versus delayed gratification. The product owner prefers the instant reward of a happy customer instead of discussing, arguing, and explaining that he can't just stop everything every time the client feels the urgency of an unpredicted need. About the other delayed features, well, we will see that in options number four and five.

Sometimes, nevertheless, the three urgent unpredicted features will in fact be urgent, and there won't be an option to put them at the end of the queue. Then, and only then, should we play option number two: delay all client A's features, and put these three new features instead. Looking at the graphic, that could even mean doing B's feature first and then using the three A slots to deliver these new urgent features, but of course the originally scheduled A features will be delayed in favor of the new ones.

As rational as this may seem, many clients will complain and fight back if three of his older requests are delayed in favor of his own three new, more urgent requests. If the product owner is pressed enough, he will probably start to consider nonrational moves, like option number three: put the three incoming urgent requests at the beginning of the pipeline. If he does so, even if he is not telling the clients, the reality is that the whole pipeline will be delayed in three slots. That's not fair, client B and C will definitively agree, as their projects are being delayed because of the need of client A, which is none of their business!

Having reached this point, move number four is practically automatic and has been often discussed in Agile literature: the product owner bursts into the iteration and commands the team to finish the already committed iteration backlog *and* the three new incoming features.

Of course, again, this is a way of irrationally brute-forcing the system by pushing the problem out of his desk. Very often, the team will not be mature enough to say "no," and when they under-deliver, or deliver something that does not work, the old "you promised!" tune will be sung to them.

In iteration-based implementations of Agile, breaking the team's iteration is considered a capital sin: it breaks flow, introduces context switching and reduces team's autonomy and motivation. When this situation is frequent, many teams have moved to a Kanban approach, where priorities can be set and managed in a smaller time frame, even daily. The trade-off is that maybe changing priorities will be encouraged, and planning in the long term will become more difficult.

We still have a fifth move you've maybe guessed: adding more resources! As we've already showed, programmers are not building blocks, and adding more resources in the short term is more likely to increase costs and delay the project even more rather than to improve anything.

The Two Golden Principles in Portfolio Management

For better or worse, the truth is that portfolio management does not have more complex rules than the ones we've already discussed in the pipeline example. Agile companies manage their portfolio by creating a backlog of pending features or work batches, sometimes in the form of "epics" or group of features. They prioritize them, give some kind of rough high-level estimation of effort needed, and then decide on the top priorities to be

addressed during next iterations. And then, they repeat the process constantly and frequently.

Anyway, there are two basic principles that, hard as they are to follow, will help you to better manage your portfolio:

1. *Trying to please everyone is the sure path to mediocrity*. I think I got this phrase from a similar one used by Colin Powell. He used it in a leadership sense, meaning that if you avoid the tough decisions, confronting who needs to be confronted, and prioritizing some things over others, you'll miss the best results you can get and settle for "good enough" instead.

> "I don't know the key to success but the key to failure is to try to please everyone."
>
> –Bill Cosby
>
> "If I had asked people what they wanted, they would have said faster horses."
>
> –Henry Ford
>
> "You can't just ask customers what they want and then try to give that to them. By the time you get it built, they'll want something new."
>
> –Steve Jobs

2. *If you can't say "no," your "yes" means nothing*. I heard this phrase from Esther Derby on XP 2011 and decided to use it as a way to explain principle number two instead of the one I was using until then: "focusing and effective prioritization means learning how to say 'no' to everything but what you are doing right now." Ironically, trying to please all your clients by saying "yes" to anything they ask for will, ultimately, make them mad at you, because you can't please everyone at the same time.

> "Have the courage to say no. Have the courage to face the truth. Do the right thing because it is right. These are the magic keys to living your life with integrity."
>
> –W. Clement Stone
>
> "Half of the troubles of this life can be traced to saying yes too quickly and not saying no soon enough."
>
> –Josh Billings
>
> "All the mistakes I ever made were when I wanted to say 'No' and said 'Yes.'"
>
> –Moss Hart

Handling the project portfolio is a matter of selecting what to work at next, always considering that, as we will see next, a healthy backlog has more things than those that the company can handle before a given date. Criteria used to prioritize the backlog should mix short- and long-term concerns, including company strategy, market opportunities, available resources, project risks, etc. In that decision, a few projects, epics, client requests, or individual features will be chosen to configure the next iteration, but if we keep changing those priorities every time clients are upset, we will make the situation worse introducing waste, context switching, lower productivity, decreased morale, poor quality, and, ultimately, building a technical debt monster that will eat your company for breakfast.

Sisyphus, the First Product Owner

In many cases, Agile will provide simple and valuable measures on team capacity that will let you realize that you have more work to do than the capacity to do it! Is it time for the product owner to panic?

This always reminds me of Sisyphus. As you probably know, Sisyphus is a character from Greek mythology who was punished to roll a huge rock up a hill, only to watch it roll back down every day. For me, this is a good image to depicture the product owner versus team's capacity.

Product owner sees a team capacity of 10 average features per iteration, and he sees that if they were able to do 12, they would be on time for every project. So the team strives for improvement, investing in reducing technical debt, learning new skills, implementing better tools and practices, etc. And some months later, they are able to do 12 average features per iteration! Woo-hoo! So they live happily ever after, right?

As you have already imagined: wrong. As the team now has higher capacity, the company will be pushing more work to them. So now that the team has a velocity of 12, probably the product owner wishes that the capacity was 14 so they could make every project successful. And thus, the product owner is punished to roll the team's capacity rock up the portfolio hill, only to see how the portfolio always grows beyond the team's capacity.

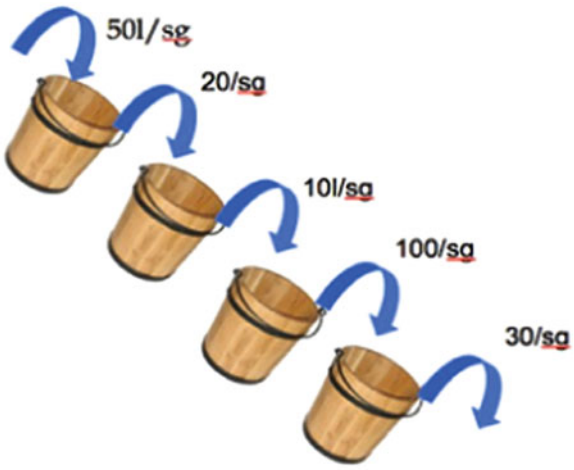
The fact is that healthy companies have *slightly* more work to do than the capacity to do it. If this is not the case, the company has spare capacity that it is not using, which is a terrible form of waste: the company will need to find more clients and work to do, thus increasing the portfolio, or reducing headcount until capacity matches workload.

If companies have *much more* work than capacity to do it, then the delays will be unacceptable. Ways of dealing with this situation include growing the company, which will take some time, increase costs, and reduce performance because of a bigger complexity; or increase prices, which will make the demand lower while maintaining costs and profits, but will not let the company grow. A mix between both approaches will probably be the most suitable way to deal with a growing portfolio.

If the company has a workload slightly bigger than capacity, a work queue will be formed. This queue will be emptied on low-demand periods, and the most tangible effect will be that projects will not be done immediately, as a small delay will be experienced while queuing. In those cases, trying to do all projects at once will seriously affect lead time, as Kanban systems have proven. A more suitable approach will be to select the next critical projects and perform them nonstop before starting new projects. If nonstop development of projects is considered not possible, this must be seen as a major corporate impediment and addressed through an urgent Kaizen program.

Dealing with Bottlenecks

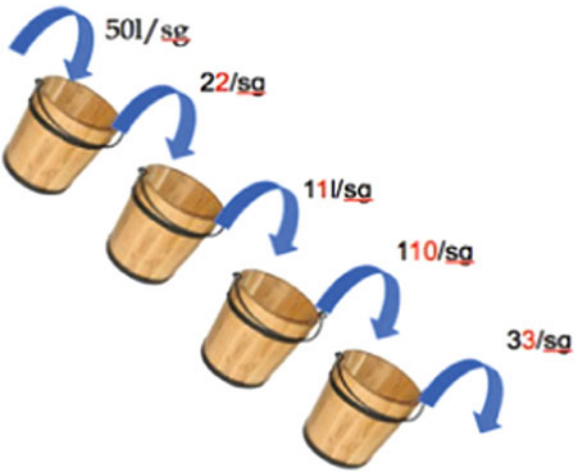
So you have more work than you can handle, and you want to increase your capacity. The wrong way of doing it would be to ask every single part of your production process or value stream to work harder and produce more, which will inevitably bring inefficiencies through suboptimization. Consider, for example, the following graphic:



Maybe you will think that this is an oversimplification of productive processes, but I have found this example to be very useful on frequent occasions when trying to explain how to deal with bottlenecks at value streams.

As you see, there is a bottleneck in the second bucket, which is producing a throughput of just 10 l/s. Trying to introduce more work will only lead to water leakage (waste), and the global throughput of the system will still be 10 l/s.

What happens when you ask every bucket to increase performance by 10%?



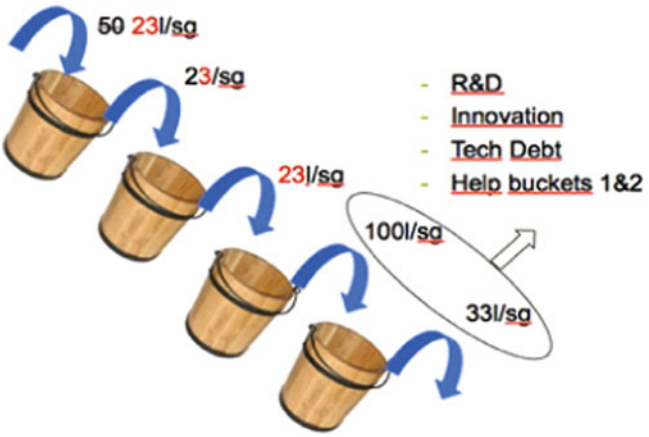
As you see, global throughput has been effectively increased by 10%, from 10 to 11, but the cost has been very high – a total of 16 percentage units have been spent. If those 16 units were used exclusively to raise the bottleneck, we could have increased its throughput to 20 and then use the remaining 6 points between buckets one and two, thus increasing global throughput to 23 – that's a 130% improvement at the same cost! And, by the way, buckets three and four would possibly be much happier than in the first case, where they didn't understand why they were required to increase their throughput with no noticeable results to global performance.

As you see, a good way of increasing your capacity is spotting your main bottleneck and working on it until it is not your bottleneck anymore, then work on the next bottleneck. This is one of the principles behind the Theory of Constraints (ToC). Originally introduced by Eliyahu Goldratt in his 1984 best-selling business novel, *The Goal* (Goldratt 1984), and extended later in

his subsequent books *Critical Chain* (Goldratt 1997) and *Theory of Constraints* (Goldratt 1999), ToC focuses on the premise that bottlenecks can be attacked following a systematic process of exploiting, subordinating, and elevating.

Exploiting means that first of all you must make sure that the bottleneck is operating at its maximum capacity. Orders, jobs, and assignments that can be handled by other parts of the system that are not so constrained should be reassigned, so the bottlenecked resource can concentrate on the parts that cannot be performed elsewhere. Jobs should arrive in the best possible condition, so the resource doesn't waste time preparing or rearranging them, and you must make sure that the constrained resource is not blocked or facing impediments at any time, as the whole system will suffer for any delay in the bottleneck.

Once you make sure that the bottleneck has been exploited, it is time to subordinate the system to the constrained resource. In a production line, or a value stream, nobody should work faster than what the bottleneck can handle. If the previous stage of the production line is overloading the bottleneck, that won't make it more efficient – more probably, it will collapse and crash it. When the system subordinates to the bottleneck, it can happen that previous stages of the production process are waiting for the bottleneck to create available slots, while next stages are idle waiting for the bottleneck to provide something to work at – use those free resources to work at the bottleneck! Train them and let the bottleneck delegate some of its duties to the previous and subsequent production stages.



Kanban systems introduce ToC subordination through the use of WIP limits; WIP stands for work in progress. Once the WIP limit has been established for the bottleneck, nobody can produce more work for the bottleneck than the maximum WIP that it allows. If queues or inventories are allowed, arriving assignments that cannot be handled by the bottleneck will start to form a queue that will instantly be visualized on the Kanban board, hence calling for managerial support and decisions on how to deal with the bottleneck.

Finally, if you've exploited the bottleneck, subordinated the system, and still you need more capacity, you can elevate the bottleneck, meaning that you can add more resources to the system. Remember Brooks' law anyway: the elevation will take place in the long term, not for the current project, which is more likely to experiment further delay because of the elevation process.

Metrics for Capacity

I would like to end this chapter by briefly introducing some ways that Agile teams use to measure capacity.

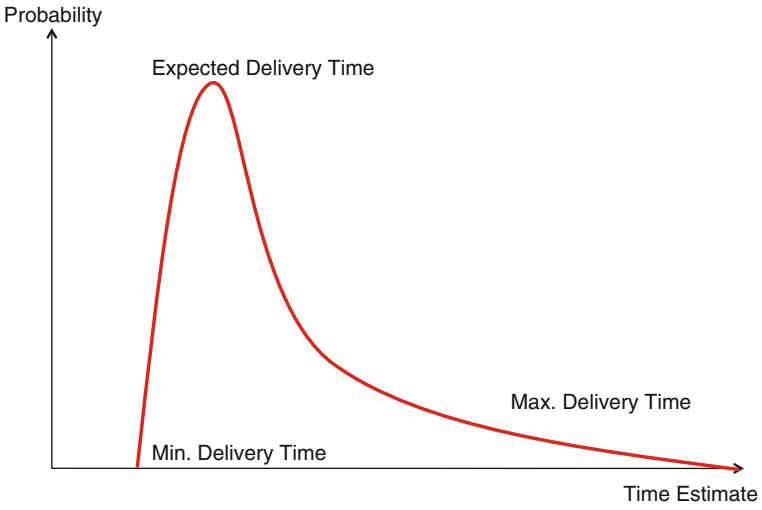
First, you have the good-ol' man-hour. I must admit that I have personal issues with the man-hours, as I always see them as a vestigial defect from Taylor's age, but to be honest, most Agile teams I've known – and I've known them by the hundreds – are still using man-hours to measure their capacity (which does not mean that *the best* Agile teams are using them).

If you use man-hours, there's a trap you must avoid, which is to count the people in your team, multiply for 40 h a week, apply some kind of arbitrary focus factor – time that you ask them to focus on the project – and end up with a number that you will measure them against. If you do that, I predict constant delays and frustration, followed by a lack of trust from management.

The first reason is that teams are not perfect machines. They have good weeks and bad weeks – full of interruptions, priority changes, reports, meetings, mandatory trainings, rework, bugs, etc. They face easy features and features from hell. And they have days when they are creative and innovative, and days when their whole code should be rewritten or directly thrown into the bin.

Managers will tend to take the best-week capacity and say "see? When you work hard, this is what you can accomplish." And then plan their projects counting that capacity for *every week*. Which of course won't happen: hence the delay and the lack of trust.

Instead of calculating a "wanna-have" capacity, you should look at the team's historical data and obtain an average, as well as a maximum capacity and a minimum capacity. With those numbers and some estimation for the backlog, you can provide three important predictions: the minimum time to deliver (using your maximum capacity), the maximum time to deliver (using your minimum capacity), and the expected delivery time (using the average). These three numbers will form a bell curve of probable delivery dates, with the biggest area around the expected delivery time and a certain trend to move to the maximum delivery time, as in this graphic – which I usually call "Medinilla's Law of Project Unfairness" because projects can be earlier to a certain degree but can be late almost unlimitedly.



Anyway, as I said before, I don't like to use man-hours. If you drop the artificial habit of forging the wanted capacity using number of people and multiplying by hours, then it is easier to move to average features or, as Agilists usually refer to them, story points.

But first, let me make a case against man-hours:

- If I work 8 hours today and you work 8 hours, are we producing the same?
- If this product has 1,500 man-hours in it and this other similar product has 1,500 man-hours too, have they got the same number of features? The same quality?

- If I say that I can do this feature in 6 hours and you say 12, is one of us wrong? What estimate should we assign to this feature?
- If 2 years ago I produced 35 hours a week, last year I produced 35 hours a week, and this year I produce 35 hours a week, does that mean that my productivity hasn't changed over the years?
- If a project was estimated in 1,000 man-hours and 500 man-hours have already been invested, is the project half done?

Man-hours are a metric of cost and effort but not a good productivity or capacity metric. So that's why many mature Agile teams move to story points or average features and measure the amount of functionality delivered per iteration. Remember that there is a principle in the Agile Manifesto that reads, "Working software is the primary measure of progress" – this is it!

Once you have some historical data on average functionalities delivered, you can use that average number to make predictions in the midterm, but remember to review those predictions over and over in each iteration, as while you progress on your project you will reduce uncertainty and the bell curve will be more and more precise.

Summary

Managing capacity in a software company is not just a matter of setting deadlines, flogging the team through the project, and asking everyone to increase their capacity. Agile companies are learning better and simpler ways to manage the project portfolio by building corporate backlogs, limiting the work in progress to the company's capacity and then working the system to spot and improve existing bottlenecks. Kanban and the theory of constraints provide a good way to do it.

Overall, the Agile manager must learn to gently say "no" and stop trying to please everyone, as this will most probably lead to angry clients and mediocre productivity in the long term. Instead, a mechanism of backlog prioritization must be instituted and thoroughly followed.

Studying the average iteration capacity of the team is another way of managing workload in the midterm. Special care must be taken not to arbitrarily set capacity goals but to rely on historical information and relentlessly work the system to improve the average capacity. Using productivity metrics based on value delivered to the client instead of effort-based metrics will help to have the adequate information to manage the backlog efficiently.

Things to Try

- Build a company backlog with all the projects and features already scheduled for the next few months. Make it visible through some kind of portfolio backlog board. Ask all teams to include on the board any assignment they are working on that is not shown there right now, as well as any changes in priorities. Discuss with product owners these changes and assignments and see how they affect already committed deliveries.
- Use the portfolio backlog to start a portfolio Kanban, where you can see pending features, ongoing features, and delivered features. Use it to obtain lead times and cycle time, spot bottlenecks, and trigger discussion on how to improve portfolio management. Remember to limit the work in progress!
- Review the deadline-setting process in your company. Make sure that sales people are evaluated not only by sales but also by the quality and results of those sales.
- Make product owners, not teams, accountable for deadlines. Teams should give estimates for features, and they should maintain, if not improve, average capacity or velocity. Product owners should use that information to estimate deadlines, but if the deadline is not met while the team still maintained its average velocity, make it a product owner's issue, as pressing the team won't help you solve the problem – they are in fact performing as they always have been.
- Work in smaller batches. Make analysts, product owners, teams, and everyone in your organization search for ways to reduce the average feature, task, or work package. The smaller the package, the easier to manage the portfolio, and efficiently steer it when priorities change.
- Make sure that teams are delivering working software at the end of every iteration. The definition of working software should be "as close to live-in-production as possible," and you should have working Kaizen programs to make it even closer to "live," no matter what kind of environment you are working in. Telecom companies usually worked for 6 months to a year before they had something they could put "live," and they are moving fast to shorter 2- or 3-month-like cycles, so you should be seriously considering the same.
- Start measuring team capacity per iteration. Build historical data you can use to better predict next deliveries.
- Always provide three estimations (worse, average, expected) instead of one (deadline), thus giving a measure of the uncertainty of the estimation.

Review the bell curve after every iteration, so client has always the best possible information on the current state of the project and risk of delay.

- Learn and teach your teams how to use story points and play planning poker – you'll find several resources online, and Mike W. Cohn's *Agile Estimating and Planning* is the authorized source (Cohn 2005).

Recommended Readings

Anderson DJ (2003) Agile management for software engineering: applying the theory of constraints for business results. Prentice Hall, Englewood Cliffs

Brooks FP (1975) The mythical man-month: essays on software engineering. Addison-Wesley, Reading

Cohn MW (2004) User stories applied for Agile software development. Addison-Wesley Professional, Reading

Cohn MW (2005) Agile estimating and planning. Prentice Hall, Englewood Cliffs March 11, 2004

Goldratt EM (1984) The goal: a process of ongoing improvement. North River Press, Croton-on-Hudson

Goldratt EM (1997) Critical chain. North River Press, Croton-on-Hudson

Goldratt EM (1999) Theory of constraints. North River Press, New York

Krebs J (2008) Agile portfolio management. Microsoft Press, Redmond

http://www.springer.com/978-3-642-28908-8

Agile Management
Leadership in an Agile Environment
Medinilla, Á.
2012, XII, 184 p., Hardcover
ISBN: 978-3-642-28908-8