Estimating considered harmful?

Author: Han Schaminée

Contributors: None yet

Version 1.0

Introduction

For many years, we discuss in the software industry the topic of estimating. A lot has been written about it, but most software teams continue to struggle with it. So, what I write down here on software estimating is not another method. It is more my view on estimating and why and how it can be done in an effective way. It is also necessary to write something, as one of the misconceptions about Agile is that a true Agilist does not plan. I believe estimating can be very valuable when done in a right way and for the right purposes.

I don't want to list all the documents I read and inspired me to write this document, but there is one that really triggered me to think about the question: Why on earth do we do estimating given the amount of uncertainty in software development? I got really inspired by some work of Thoughtworks. Thanks for that. There are many ideas in that document not (yet) included in this discussion paper. Some work to do …

Executive summary

Estimating is often a wasted effort when not done in the right way or not done for the right purposes. But it can be useful if used for decision making and synchronisation rather than steering on predictability.

When we estimate, we better estimate in terms of story points than in terms of man-days. It is OK when story points are not unified over the teams, it is more important teams make their own estimates. And managers should accept the inaccuracy of estimates caused by the intrinsic uncertainty of software development.

Teams should follow a standard process to come to estimates. Feature roadmaps may be estimated with even less accurate estimates.

Why would you estimate?

Estimating is often a wasted effort

Amongst Agilists you often hear that estimation is a waster effort and in fact harmful. There is intrinsic uncertainty in software development efforts, but by generating estimates people often forget about this uncertainty and make software development just an execution of the plan. Progress is monitored against the plans and people are upset when actuals differ from the plan. As everybody want to meet the plan, often quality is sacrificed which makes things only worse. Delay is often seen as loss and creates an impression of underperformance. This impression further lowers the motivation of the software engineer and leads to lower productivity and sand bagging in the next plan. And, as we learned from Goldratt[1], buffers do not help and only further reduce the productivity. So, Agilists are right to consider estimation harmful and make statements like "it will be done when it is done!".

It is good to understand where the uncertainty comes from. Software development is really difficult. There are unclear and changing requirements as nobody knows exactly what the customer or end-user would like to have and what competitors develop in parallel. It is the first time we develop this software and often the team capabilities to develop this software are not known. So, no wonder it is difficult to project what it takes to deliver a successful software product.

In some way, the situation looks like what Manufacturing found in the eighties of the last century. It is also not so easy to predict order income, as market demands vary. Many manufacturing plants used big manufacturing resource planning systems to predict what material was needed on the shop floor, and they were often wrong. The Japanese introduced lean manufacturing and simple 2-bin system where they pulled material on the floor

[1] E.M. Goldratt: Critical chain

when really needed, rather than planning. Manufacturing learned there are other ways to manage uncertainty rather than plan and re-plan and re-plan …

And some software development departments have adopted these methods and use things like Kanban to manage the activities. Activities are ordered by priority, we start with the most important one, work very hard and will finish it in the time it takes. The basic assumption is that when you work very hard on the most important things, this is by definition the optimal path. So, is there no use in estimating at all?

But estimating can still be useful

I have not seen many cases where the software engineer has most of the time nothing to do. In fact, in most cases the software engineers are fully loaded, often leading to a phenomenon of congestions, which I will discuss in a separate paper. So, a decision is needed on what topics a software engineer will work. To make such a decision, it is not only useful to know the economic value created but also how much effort and costs are involved. But if you don't need a decision and you have to do this anyway, there is less need for an estimation and you better use the team resources for something better. Some people say, that when priorities in a company are clear anyway, you don't need estimations.

Another reason why estimations can still be useful is when the deliverables of certain teams have to be coordinated. Although we try to limit dependencies as much as possible, sometimes teams are dependent on each other and synchronisation is required. When production times are very short, it is worthwhile to start the activity in the receiving team only when the sending team is ready (like 2 bin in manufacturing), but when production times are longer, some planning may be helpful to shorten the total time to market of the feature and as such reduce the time to break even of the investment.

The third example of where I believe a planning is as a reference to deal with the intrinsic uncertainty (see also discussion paper on Agile and commitment). When you use the plan as a reference, you can generate early indicators whether or not a commitment is feasible. It acts as a baseline to help assess changes. It may also help to improve the estimates for the remainder of the project and as such revisit the decision whether to invest in this project anyway. But to revisit the decision, we should never take sunk costs into account, and therefore the likelihood the new costs estimates will be bigger than the benefits, will reduce rapidly during the project. It is more likely, we stop the project because of better estimates of the benefits.

Some people will state that the estimation effort itself is already useful as it will force the teams to discuss the specification and get the expectations clear. I believe there are other ways than estimation to force such a discussion.

Last but not least, a plan will allow you to visualise WIP and as such optimize productivity and cycle time.

So, estimations can be useful, but should always be light weighted (do justice to the inaccuracy) and tuned to its purpose.

Estimations should not be used to support a management by commitment (leads to lower productivity), or bonus systems where you can have a bonus when you do what you have predicted or even more. Both examples just lead to lower productivity. This all is easier said than done. Many senior managers are at some moment in their career confronted with the fact they have to manage other domains than where they have substantial expertise in. They hire experts for those other areas and have to rely on them. But they still would like to have an opinion whether or not these experts do a good job. In many cases, the manager decides to judge the expert on predictability, which by definition leads to lower productivity in environments with uncertainty. Although there is value in predictability, in many cases in software development, the intrinsic uncertainty makes there is more value in productivity: you better be faster than your competitor rather than be ready exactly according to plan!

How do you estimate

Story points are better than man-days

I often see teams that estimate and report in man-days. A common mistake I see is that people report they should be half way as they burned half of the planned resources. This is clearly wrong and you better report in

terms of value delivered rather than in costs made. Story points are a better approximation for that value (although still in some way costs based; but we often lack good measures for value created).

There are a few other reasons why I prefer story points over man-day estimates. Many people make man-day estimates to think about the activities to be executed, make a mini-break-down in their head and make an estimate independent of historical activities. Story-point estimates, however, force people to refer back to previous projects and use historical data to size the activity.

Furthermore, a man-day estimate can easily be translated into lead-time and is easier perceived as a commitment. This will lead to sand bagging and a more conservative, buffered estimate. Man-days estimates also seem to demand more accuracy than justified (I have seen estimates like 170.2 man-days). And as we know buffers become true, it will lower productivity. Relative sizing is anyway more accurate over a larger sample size than absolute sizing.

At last, with man-day estimates, it is not possible to measure productivity improvement. I remember a stubborn product manager who always used as unit of estimation what a team can do within a 2-month release iteration. It sounds like the guy at the fuel station who does not care about fuel price increases as he always fills for the same amount! It clearly was one of the reasons for a decreased productivity in his big team. You better measure productivity (throughput per cost unit) in terms of story points and show that the teams can on average do more over time. There will be months with lower velocity, but on average, velocity is going up. One more remark on this: the target for productivity improvement should not end up at the people who make the estimates as your will get a self-fulfilling prophecy and the story points will inflate, in spite of the on-purpose chosen Fibonacci or exponential gaps between possible estimation outcomes.

We don't need unified story points

Different teams will have different reference stories to come to an estimate. This will then also lead to different velocities. Should we normalise the story points? There are pro's and con's but unification is anyway not a high priority, it is more important teams start using story points for estimating by comparing to stories the team knows. And if that means not unified, let it be.

The advantage of unified story points comes when for instance a feature has been sized and not yet broken down in user stories and allocated to teams. These high-level T-shirt sizes are often uniform over the teams.

A disadvantage of unified story points is that managers are tempted to compare teams and judge performance based on differences in velocity. This has never been the purpose of story points and should be considered evil, as it will invite teams to window dress and artificially improve their velocity. There are often many other reasons behind differences in velocity. You better judge team performance on quality and cycle time.

The teams should make their own estimates

In a more traditional, waterfall environment, you often see that estimates are made by architects and the teams are not involved at all. Very often reality turns out to be different and the teams do not even care, as it was not their estimate. As Agile teams are autonomous, cross functional teams and totally responsible for the result they deliver, it is good they also feel responsible for their own estimate. They are also the best to judge to which historical story the story to be estimated most looks like. The increased ownership will increase the chance of delivery within the sprint. But because of the intrinsic uncertainty, there is never a guarantee. And teams should feel trusted they did everything to finish the story within the sprint, but shit happens once in a while.

Managers should accept inaccuracy of estimates

Managers should create a culture where the uncertainty is accepted, not neglected. Sometimes, things take a bit longer and managers should accept. The team should be judged if they tried their utmost to finalise the story in the sprint (attitude) rather than whether they were able to keep the uncertainty away from the well-paid manager. Management by commitment is so easy and not worth a high salary. It only leads to buffers and the bad thing about buffers is that buffers tend to fill them self.

So, with well-defined stories, properly estimated in story points, the uncertainty will make there is variation in the velocity. Let it be. With a large number of stories, it will average out. The best way to deal with uncertainty is to make use of the law of big numbers: the variation of the sum will go down.

There is also no need for best case, likely case and worst case estimation. The uncertainty is reflected in the velocity and to deal with a customer commitment, we better closely monitor the velocity and keep it within a certain range (like any statistical process control in manufacturing).

Teams have a structured process to come to estimates

Teams should define a standard process to come to estimates. The process could look like:

1. Identify base stories
2. Talk through the requirements of the story and document key points (summarise the story)
3. Ask questions like
    a. What do we need to know/ learn before we can start?
    b. Did we do something similar before?
    c. Do we need special set-up for unit testing, acceptance testing? How can testing be automated?
    d. Are there external dependencies?
    e. What are non-functional requirements?
4. Identify a point of relative comparison
5. Reach consensus on size and Definition of Done
6. Check consistency amongst stories as you go along

There are many processes defined to reach consensus as meant in point 5. A common one is based on wideband Delphi[2]: each team member holds a poker card (with Fibonacci numbers) with his estimate. If the highest and lowest estimate are two adjacent Fibonacci numbers, take the highest. If the difference is more, let the team member with the highest and the one with the lowest explain their estimate and repeat the exercise afterwards until consensus has been reached.

Programs use T-shirt size estimates for the feature roadmap

Programs often produce a longer-term roadmap for longer-term alignment with other parts of the organisation or with other parts of the customer project. Roadmaps get less accurate, the further away the roadmap items are. So, very soon estimating the roadmap items become not worth the effort. We recommend to do very high level estimates in what we call T-shirt size estimates: Small, Medium, Large and eXtra Large. The program can use standard mapping from T-shirt size estimates into story point to make a link to the predicted velocity and as such make the feature roadmap a bit more reliable in terms of feasible execution. For instance, small is 3 SP, medium is 8 SP, Large is 21 SP. We recommend to install a feedback loop on the T-shirt estimates from the later story estimates and see if on average the mapping is correct.

If things are really difficult to estimate, we recommend to “take a bite”. Just reserve some capacity in the team (spike), start working on it and build more insight. Then, come back and discuss if you can make a better estimate.

[2] https://en.wikipedia.org/wiki/Wideband_delphi