How to get better at estimating tasks in 10 steps

Originally published on 25 October 2020
https://keepgrowing.in/taking-care/how-to-get-better-at-estimating-tasks-in-10-steps/

Estimating programming tasks is a complex process. There is no one-size-fits-all solution. Furthermore, skimping on resources to get it right turns the entire procedure into reading tea leaves. However, we can improve the accuracy of our predictions by applying the following practices.

1. Expect trust when making decisions

Having the “productive partnerships” value from the Manifesto for Software Craftsmanship in mind, verify that the relationship between a Development Team, Domain Experts and a Product Owner is free from the mis-engagement of management. This advice is extremely helpful in reducing the time we waste on work that doesn’t add much value to the product. Moreover, we’ll be able to produce more accurate estimates.

The team decides "how" to achieve goals, the Product Owner decides "what" is most important. The Domain Experts provide crucial knowledge about their area of expertise. Don't let this balance suffer from micromanagement or unnecessary intermediaries and remember that developers are not just code monkeys. Otherwise, estimating tasks lacks the insight that the team could have brought in personally discussing goals and requirements with the Product Owner and Domain Experts. Ultimately, the estimates are unreliable and the time spent on tasks is wasted because the value provided is less relevant to end users than it should be.

You’re a professional

Ignoring developers when estimating tasks might result in numbers that are appealing from a marketing point of view. However, it shows a serious lack of competence. Unfortunately, there are many projects that have a technological stack, due date and feature list sealed on paper before even a single programmer is consulted.

It makes no sense to have estimates that are not based on reality. To cope with the imminent problems, programmers are asked to abandon rules of software craftsmanship and cease writing unit tests. This is a quick and stupid way to completely destroy a project by making it unmaintainable, unusable, buggy and even dangerous.

The fact that management may not understand the importance of the principles guiding professional programmers does not mean that they can view them as a waste of resources.

Trust facilitates task estimation

One of the most important strategies to ensure that a task actually benefits the project is to allow developers to review both its requirements and design with the strong and direct support from the Domain Experts and Product Owner. As many team members as possible should be involved in this review to get reliable and thorough conclusions that are priceless when estimating tasks.

By default, accept that the responsibility for making decisions about architecture, technology and organizing work rests with the team. The collective code ownership practice guarantees that the whole team ensures project quality. In addition, the extreme programming and software craftsmanship principles greatly reduce the impact of inaccurate projections – it's easy to add new functions efficiently when the existing code adheres to those practices.

Micromanagement is a waste of time that has no place in a dignified and well-functioning organization. Moreover, a project that relies on a manager who used to have something to do with IT many years ago and now has the decisive voice in architectural decisions is dead on arrival. Keep in mind that the more valuable and productive developers are, the sooner they leave an organization that doesn't trust them and doesn't treat them like professionals. Even the most sophisticated methodologies will not help in obtaining valuable estimates when no one wants to work in the organization long enough to learn about the specifics of the project.

2. Consult problem finders

Developers competent enough to identify the nuances and risks of software development process shouldn't be shushed when the team estimates tasks. Often what they have to say does not match our wishful thinking. Therefore, problem finders protect us from committing to overly optimistic estimates. They are invaluable in adjusting the scope of a project and making it more realistic.

Never sign all documents confirming the terms of a contract without consulting with developers. Take them seriously when they point out unrealistic expectations or promises conjured out of thin air. Don't underestimate their cries for more down-to-earth estimates – this protects the entire company from becoming addicted to miracles as the due date approaches. Any wishful thinking about a project deadline or scope is unacceptable.

A real professional will do all it takes to keep their employer from financial consequences of overcommitting. This includes providing the necessary expertise to completely deprive management of any illusions before they seal any promises on paper.

3.Allocate enough time for proper research

Even an experienced team needs time to research new, better, and more optimized ways to implement features. In other words, we cannot rely on implementing familiar and common features in the way that was used three years ago.

Lack of research results in unoptimized, outdated code that is vulnerable to security breaches and more difficult to maintain by younger developers, accustomed to modern, better-suited tools. If you don't include the research phase as part of the assignment you evaluate, you will lose the opportunity to learn about new security issues that could endanger the project or external libraries that would perfectly suit your needs, saving a lot of time for implementation.

A research phase is flexible and it takes various amounts of time for different developers and different tasks. However, ignoring it when estimating tasks will rise maintenance costs as the new code may introduce bugs or fail to solve the real problem.

How to research a programming assignment so the time is well invested

1. Develop a comprehensive understanding of the problem. Book some time to consult Domain Experts, don't be timid about the amount of questions – developers should always be allowed to make as many inquiries as they need.
2. Rephrase the problem from the point of view of users. What value will solving the task bring to the project and the people using it?
3. Create sketches, diagrams, make notes in plain English, enumerate algorithmic steps, write test cases. Ask other programmers for peer reviewing your answer to a problem, verify whether you're thinking in the right direction or need to change the approach. Question the solution design while it is still only on paper.
4. Ask what the possible blockers, difficulties and complications are. Don't attempt to solve them at this point! The sole purpose of this list is to prevent surprise from foreseeable problems. In addition, it will make the research phase more thorough and provide you with a list of exceptions to deal with.
5. Research the place in the existing codebase where the solution is meant to fit. Which parts of the system will be interacting with each other?
6. Be clear about features that the solution won't provide. Mark the boundaries to protect yourself from feature creep.
7. Search for ready to use solutions. Examine available open source and commercial libraries and asses their usefulness, availability and maintenance cost. Ponder on the pros and cons of adding an external dependency to the project.
8. Use a spike to verify that your solution passes unit testing that reflects the most important requirements. Responding to business needs is a very valuable part of our job, but it may be cumbersome. Therefore, we must be able to check many ideas quickly and easily against these tests.

Benefits

Attach the artifacts created during the research to the task description. Make your research explicit to other programmers. Not only will it save a lot of time when someone has to pick up your assignment, it will also provide test cases, a list of exceptions to handle, and details that can be copied into the project documentation.

4.Break down features into easily manageable tasks

Make sure that the tasks are actually achievable in a reasonably short time, e.g. they have been evaluated by the team at a small number of points. As a result, any mistakes made when estimating tasks will appear early, will be minor, and won't disrupt the project schedule.

Don't hesitate to break a task into smaller components. It'll help you define the requirements more clearly and identify all corner cases. Moreover, the whole story will be implemented more accurately, without taking any shortcuts that could harm the project in the long run. This will keep everyone more focused on solving real problems and adding substantial value to the project.

Another key point is that more concise tasks make continuous integration much easier. A regularly updated codebase means we don't have to make as many assumptions when estimating future workload. Moreover, we can follow the progress on a burn down chart more accurately and quickly notice even a slight delay. Thus, allowing us to apply a back-up plan in good time to meet the deadline (e.g. cut the sprint scope, do pair programming, use an open source library instead of custom code).

5. Keep priorities in check

The better defined and divided the tasks are, the easier it is to prioritize them. Estimates are only forecasts based on many assumptions, guesses, and, usually, a small amount of data. It's irrational to expect them to be 100% correct. Therefore, we must be prepared when reality does not follow them. Fortunately, thanks to reasonably short sprints (e.g. two weeks) and well-defined, small tasks, we can quickly realise that the predictions may not be correct. When the priorities are clear, we can engineer a back-up plan for the situation when some of the most valuable features can't be delivered on time.

Separating what's crucial from what's nice to have in a project might even save the deadline as we can easily decide how to narrow the scope of the upcoming release. This way, we can meet the deadline and still deliver a valuable product.

“How Dead Space's Scariest Scene Almost Killed the Game” – the interview with Glen Schofield, the co-founder of Sledgehammer Games and the creator/director of Dead Space, gives a great example of how adjusting a project's scope, maintaining laser focus, dividing problems into manageable chunks and sticking to priorities can save a project.

6. Estimate in appropriate units

Be aware that an hour from an experienced developer gives a different amount of work than an hour from a person learning a particular technology. One hour is not equal to another. Estimating in abstract points and according to the Fibonacci sequence will best reflect differences in task complexity.

Moreover, defective leadership can make programmers prioritize meeting a specific deadline over solving a problem, shifting their attention from understanding a task to its due date. Thinking about how much time we have before a task has to be marked as "Done" on the project board makes it difficult for us to focus on dealing with the actual problem. As a result, many tasks "completed" on time provide only superficial solutions to persistent problems. As we can read in the Atlassian estimation guide:

“Story points reward team members for solving problems based on difficulty, not time spent. This keeps team members focused on shipping value, not spending time.”

https://www.atlassian.com/agile/project-management/estimation

Estimating tasks in hours – risky and costly

A team can TRY to estimate in tangible units without bringing much harm to the project, if ALL of the following rules are met:

1. Tasks have been estimated in abstract points for long enough to allow us to gather a meaningful set of data required to identify recurring patterns as well as to develop the ability to properly divide problems into smaller ones and gather business requirements effectively.
2. Tasks are always small, well-defined and most of them received the same (and low) number of points – there is little or no difference in complexity between them because the team already knows how to divide problems into small chunks.
3. You collected enough statistics for the particular team to calculate how many hours on average a task takes for a given number of points (the greater the number of points – the less reliable the hourly estimate).
4. You realise the statistics expire when a team gains or loses a member – you have to go back to estimating in points.
5. The team is strongly unified in skills and competence within the project's technological stack (never use tenure as a metric for developer's experience), which means that one developer working hour gives a similar result to another developer working hour (highly unlikely).
6. The team agrees on this.
7. Estimating tasks in hours is carried out in each team separately (do not use statistics collected for one team to another).

Consequences

If the team chooses to follow this approach, everyone involved in the project must understand that the estimates have a margin of error to accept and are generally less reliable than the velocity measured in points. In addition, the amount of work required to keep the hourly estimates as close to reality as possible and the number of complications that could destroy our forecasts make this approach less effective and risky. The cost of switching between hourly and point estimating each time a team loses or gains a member greatly undermines the whole idea.

7. Remember that estimates are not transitive between teams

Estimates only belong to those who have committed to them. You can't freely apply them to new team members or even other teams. Take into account all differences in experience and proficiency between programmers – they are not faceless, indistinguishable beings. Expect teams to take responsibility only for the estimates they have committed to. Don't compare estimates between

different groups to pressure one of them to make more optimistic predictions only because the other one can do it.

"Each team will estimate work on a slightly different scale, which means their velocity (measured in points) will naturally be different. This, in turn, makes it impossible to play politics using velocity as a weapon."

https://www.atlassian.com/agile/project-management/estimation

Other people's predictions or management's ideas about the duration of the assignment are in vain without the team's confirmation. Always see yourself and other developers as professionals who can evaluate their own work.

8. Ask for the actual scope and deadlines

Make sure everyone on your team knows the true deadlines and project requirements. Agreeing to work with managers who keep this knowledge to themselves is asking for trouble. There are already enough assumptions that the team must make to maintain an agile approach and deliver software tailored to the ever-changing needs of consumers.

If somebody is deliberately hiding business knowledge to force more optimistic estimates, the project is doomed to failure. If you are consistently in a situation where you learned about a requirement crucial for estimating a task AFTER you had started working on it, you have to take a stand. First, when someone wants to add an expensive requirement to an already estimated task, agree on what can be removed from the sprint scope if the deadline is compromised. Second, make sure that transparency is a value shared by everyone involved in the project. Otherwise, the estimates are useless.

The team should refuse to estimate when communication with stakeholders lacks transparency or someone is lying about the requirements to get better estimates. If developers don't know the context, strategic goals, and priorities of a project, their performance suffers. They can't foresee the consequences of their code and technological choices, leading to unsustainable and expensive code. In summary, agreeing to work in a dishonest environment takes away the opportunity to solve real problems. Instead, we provide only superficial solutions and a rapidly growing technical debt.

9. Use past data to better estimate future tasks

Being consistent about measuring velocity pays off when we need to predict future productivity. There is no point in committing to tasks worth 60 points when, on average, our team manages to complete 30 in one sprint.

Ignoring statistics not only destroys confidence in the team's past performance, but also makes prioritizing difficult. Wishful thinking brings more harm than good. When we disregard the actual team performance, the scope of the sprint may expand in an uncontrolled manner. Trying to force

estimates that are inconsistent with data and facts is an easy way to destroy the project and team. Agree on the techniques you use in your sprint review to gain valuable insight, such as:

“Try, for example, pulling up the last 5 user stories the team delivered with the story point value 8. Discuss whether each of those work items had a similar level of effort. If not, discuss why. Use that insight in future estimation discussions.”

https://www.atlassian.com/agile/project-management/estimation

The hope that we'll automatically "get better" at estimating tasks is unreasonable. We can't even copy estimates for the same tasks! Assuming you repeatedly evaluate and do one type of task, you will also be more efficient in completing it, which impacts each subsequent prediction. If the tasks are identical, you deal with the problem only once and then somebody can reuse the solution (unless you solved the issue poorly). Furthermore, as a developer, you have to work with different technologies, resolve various problems and constantly learn new tools. Therefore, even when working with similar applications, you are never asked to estimate two identical tasks.

We can't effectively learn from the past if we haven't measured progress or recorded any lessons from previous sprints.

10. Review both failed and successful estimates as a team

If you have to explain overdue tasks, take your time to identify the actual sources of the delay. Reiterate over the steps presented in this article and answer the following questions:

1. Who decided on the estimates? Was it the team in agreement with the Product Owner and Domain Experts or was it someone from the management?
2. Has anyone tried to brainstorm possible problems and blockers that might hinder progress?
3. Was the research phase a planned and organized undertaking?
4. What is the size of the overdue tasks? How often was the new code merged with the codebase? In which stage of the sprint did you realise that something won't be delivered?
5. Did the priorities keep changing during the sprint? Who decided on what should be done? Did you have a list of features that are not essential and could be cut from the sprint's scope? Did the team spend time on nice-to-have features before moving on to must-haves?
6. In which units were the estimates made? Did you estimate in hours for a development team of varying skill levels?
7. Did anyone decide to move the work to another team and keep the original projections?
8. Were the goals clear to everyone? Or did the management keep the team in the dark to avoid tough discussions about scope and priorities?
9. On what basis were the estimates made? Do you have any historical data for the team such as its velocity to use as a reference?
10. What conclusions from the last review were implemented in this sprint? Did they work?

Group the problems

Trying to tackle all obstacles at once can be discouraging and ineffective. As a remedy, try to assign the problems you identified to the following categories:

- requirements that were kept hidden from the developers during estimating tasks;
- actual surprises – the problems that could not be predicted;
- what was poorly estimated – include both too optimistic and too pessimistic estimates.

Determine which group is the most numerous. As a result, you will know which area's improvement will have the most positive impact on better task estimation.

Discard what doesn't work, embrace what's worth keeping

Consult every member of the team and use their feedback to identify the estimation mistakes. Furthermore, don't forget about all the techniques and procedures that improved your accuracy. Make sure that the conclusions are clear and everyone involved in the process agrees on them. Transfer the truly beneficial practices to next sprints and other projects if applicable.

However, if the estimates were imposed on the team, the developers don't own them. This limits the space for improvements as they can't do much until the management starts to trust them.

References

1. Manifesto for Software Craftsmanship, http://manifesto.softwarecraftsmanship.org/ (October, 2020).
2. Scrum Development Team, https://www.scrum.org/resources/what-is-a-scrum-development-team (October, 2020).
3. Domain Expert, https://wiki.c2.com/?DomainExpert (October, 2020).
4. Product Owner, https://www.scrum.org/resources/what-is-a-product-owner (October, 2020).
5. GOTO 2014 – Not Just Code Monkeys, Martin Fowler, https://www.youtube.com/watch?v=4E3xfR6IBII (October, 2020).
6. Extreme programming, https://en.wikipedia.org/wiki/Extreme_programming (October, 2020).
7. Software Craftsmanship, https://en.wikipedia.org/wiki/Software_craftsmanship (October, 2020).
8. Dunning–Kruger effect definition, https://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect (October, 2020).
9. Spike definition, https://en.wikipedia.org/wiki/Spike_(software_development) (October, 2020).
10. Burn down chart definition, https://en.wikipedia.org/wiki/Burn_down_chart (October, 2020).
11. How Dead Space's Scariest Scene Almost Killed the Game, https://www.youtube.com/watch?v=BQ3iqq49Ew8 (October, 2020).
12. How do you estimate on an Agile project?, https://info.thoughtworks.com/rs/thoughtworks2/images/twebook-perspectives-estimation_1.pdf (October, 2020).
13. The price of faking agility, https://keepgrowing.in/taking-care/the-price-of-faking-agility/ (October, 2020).