Case Study

Relative Complexity Estimating in Software Development

Matt Stephenson
Stepthinking Ltd

www.stepthinking.com
matt@stepthinking.com
+44 (0) 7804 138 405

Stepthinking
Common Sense Consulting

Introduction

In software development, accurate estimation is crucial for project planning, resource allocation, and meeting deadlines. On the other hand, textbook agile methodologies promote flexibility, adaptability, and continuous iteration. Over the years I've heard many agile practitioners attempt to refuse to give stakeholders a date by which they will aim to be "finished", claiming that it is "not agile".

Obviously, this stand-off isn't sustainable, because outside of the agile environment there is an entire other set of stakeholders and constraints that need to be satisfied.

Those people who need to plan marketing campaigns around a new product, submit a budget for capital expenditure where developer time is being capitalised, or even just to know when a development team might become available to work on something else all have valid and important reasons why some form of commitment to a date is necessary.

So, the challenge is to find an approach that incorporates Agile principles for software development, while also incorporating up-front estimating and declaring a target date or time period, so that development teams can strike a balance between flexibility and giving their customer some certainty around when they'll get something of value.

The problem with traditional estimating

Traditional estimation methods usually rely on estimating *effort*, which requires an understanding of all of the tasks involved in a project. The complexity and uncertainty associated with software development make it challenging to predict the effort required accurately.

To do it requires detailed discussions about the solution that will be implemented before any prioritisation and execution of the work can be started. It is prone to errors because it is impossible to identify *every* task that will be done, and yet if we want to estimate effort with confidence, we *have* to identify all the work. Otherwise, we must lean heavily on the use of contingency, which doesn't really improve confidence in any date given, because contingency is simply an allocation of additional time for the unknowns. It isn't an estimate. It amounts to nothing much more than a guess, and typically it is late in the process that clarity is gained about whether the contingency will be used or not (it almost always is, and then some!).

When estimating effort in a task, there also has to be some assumption about who will end up doing the work. A piece of work that an experienced developer could complete in 2 days might take an early career developer 5 days, for example, so which do you go with. And what happens if a differently experienced developer ends up having to do the work? So, not only are you having to estimate all of the tasks, but you're also having to do your resource allocations as well (assuming you even have your team yet!).

Nor does effort estimating really account for any acceleration through the project as the team gels together and gains knowledge and confidence. What might take 5 days at the start of the project, when the team is new, might only take 2 days if you were to do it later, so you're also having to sequence the work at the same time as you try to estimate it.

Of course, nobody really does consider which with developer or where in the schedule each piece of work falls, so the estimates have consequently quite generic, further reducing confidence in any date commitment that results.

An alternative approach using relative complexity

Relative Complexity Estimating offers an alternative perspective on estimation by shifting the focus from the effort in each individual task, to complexity of tasks relative to one another.

It relies on comparing the complexity of one story or feature to another, and sizing each using a points scale that differentiates their relative complexities.

Unlike effort estimation, relative complexity estimation does not require a detailed breakdown of all tasks or a precise understanding of the work involved. It does still require a view of the whole scope and some statement of requirement for each feature or component (but how could you do any estimating without that?)

Nor does it require you to assume who will do the work, or where it will feature in your schedule, because none of those things change the relative complexity of a piece of work compared to others. If there is a large enough backlog, and developers of a range of levels of experience, and if the work is spread amongst the team equitably, then level of developer experience ceases to be a factor over time.

Acceleration through the life of the project is also removed as a complicating factor for estimation because after each iteration, all you say is "in order to hit our aspirational date we need to maintain an average point delivery of x points across our remaining Sprints". You can look at historic velocity (for example the average points delivery over the prior three iterations). If that work rate is above the required level to hit the date, then all that happens is you deliver more each Sprint and the average required for the remainder of the project starts to drop. You gain more confidence in your ability to hit the date.

Equally, if you fail to reach the required velocity, the average required points achievement starts to rise, and you get early visibility that the aspirational date is at risk, allowing you to intervene.

This approach allows teams to estimate without the need for extensive knowledge about the specific work involved or the solution that will be deployed.

The theory illustrated

If the team chose a points scale of 1, 2, 3, 5, 8, 13 (a variation on a Fibonacci Sequence of numbers), and sized each piece of work using that scale, where 1 is the simplest work and 13 is the most complex work you can find, it would arrive at a total points score for the backlog. I recommend that the team identifies the piece of work that they feel is the least complex in the backlog first and give that 1 point. Then find the most complex piece and give it 13 points. For this illustration, let's say the total number of points is 500.

The team might decide to put some contingency into that number because they might feel that they haven't identified all the stories needed for the project 20% contingency would make it 600 points.

We now know that broadly speaking the team needs to deliver 600 points to complete the scope of the project.

Under no circumstances must you allow a number of hours to be attributed to a number of points, because then you are just estimating effort by another name.

The beneficiaries of the work might have an aspirational date 10 months in the future for the delivery of some valuable software. If the team is operating 2-week Sprints, that means there are 20 Sprints available. That means that the required velocity for the team is 30 points per Sprint.

Right at the start of the project, the team can declare, "if we achieve an average velocity of 30 points per Sprint between now and the end of the project, we will deliver on our around the end of Sprint 20, which is in line with the aspirational date for the business". Then the team gets started.

After each Sprint, check the number of points delivered, the number remaining and whether it changes the average velocity needed for the remaining Sprints. To begin with, until the velocity stabilises it probably will change, and that's ok. Use your skill and experience to take a view on whether you think the required velocity is achievable (none of this removes the need for you to deploy your own experience in managing the project or stakeholder expectations!).

Within a few Sprints, the velocity will start to stabilise and you will either have a high level of confidence that the date is achievable, or a high level of confidence that it isn't. You will know this earlier than you'd typically know whether a project based on effort estimating was achievable, because with relative complexity you don't have to worry about how much of the work you don't fully understand in detail yet, as long as you have a good handle on the overalls cope and whether one piece of work is "harder" than another.

With that information coming early, you have time to make adjustments either to scope (if you can trim the Minimum Viable Product), capacity (if you have the option to add resource) or the date (if the aspirational date isn't cast in stone). Stakeholders can see these options early, which is always less stressful than seeing them close to the end date.

A case study

I introduced relative complexity estimating with one of my clients last year. They were at the start of a major technology platform project to create a brand-new capability in support of new products. They had historically used effort-based estimation, and were fairly successful in terms of delivery, but this project was of a scale not previously seen for some time, with lots of new work, so estimating in terms of effort was daunting, and wouldn't be likely to achieve the goal of high confidence in achievability and early visibility of non-achievability.

It was expected to last 9 months from the start of Sprint 1 and involved hiring the hiring of 70% of the development and test capacity before the project could commence the development phase. Overall, therefore the project was to be around 15 months including the onboarding of a team.

I introduced the concept, and the team members were mostly receptive to trying something new.

The team chose a modified Fibonacci Sequence for their sizing (the same sequence as above) and identified a piece of functionality that everyone agreed should be the least complex. They then found the piece that they felt was most complex.

The set about sizing other stories with the discussion being centred around whether each was more or less complex than the other things already sized.

Because the backlog was large, they didn't size everything at once. Instead, having sized a proportion of the backlog, they worked out the average number of points, and used that average to apply to the remaining unsized backlog.

This allowed us to have a very early view of a number representing the overall backlog size, which told us we needed to deliver around 35 points per Sprint for 18 Sprints to hit the date of "on our around the end of March". However, we recognised that this wasn't our end goal in terms of estimating, because we needed to size more in order to gain confidence in using an average for the remaining backlog.

As each round of sizing took place, more actual sizes were added, the average was reassessed and applied to the (decreasing) remainder of the backlog. What we found was that very quickly, the average stabilised and so the required velocity also did, which increased our confidence.

At the same time the team set about delivering the work. Sprint 1 delivered no completed story points, so we had 17 Sprints left and all of our points still to deliver (in agile, a piece of work is either done or not done, there is no "almost complete"), so our required average velocity went up.

The first 6 Sprints were volatile in terms of point attainment. It varied wildly, from almost none to slightly just below where they needed to be, and our required velocity crept up as a result. The piece of work that was estimated as the least complicated turned out to take 3 Sprints, because it depended on some permissions work that needed to be done first.

There was confusion initially regarding the definition of complexity. However, we decided that the precise definition of complexity was less important than having a general understanding of whether a task was more or less complex ("harder") than others.

Another concern raised by the development team was whether effort could be considered at all in relative complexity estimation. I clarified that effort could be a factor but should never be equated directly to a number of points. For example, it was ok to say "this is less complex because it will reuse a load of functionality and therefore require less effort", but not ok to say "all 1 pointers should take half a day".

In short, the team was allowed to consider anything they knew, either directly or from prior experience, that could improve their view of the complexity. They just weren't allowed to translate points into hours or *vice versa*.

On occasions during this period of volatility, some developers, working on lower point pieces of work, but finding them to be more complex than estimated, wanted to increase the story points after the fact. We discussed this and agreed that we would *not* do that, because it was akin to changing the complexity based on the effort. All that would serve to do was to increase the perceived velocity (because more points were banked), without reducing the remaining backlog, and give a false sense of progress.

What we agreed we *could* do, though, was take the lessons learned and apply them to similar work in the backlog that had not started yet, because this theoretically improves the quality of the estimating and the view of what work is left to do. In practice, however, we never really felt the need to do it.

Some of the desire by developers to increase the size of a completed ticket after the fact was borne out of a competitiveness for them to deliver their "fair share" of points in the overall team Sprint goal. It was notable that no developer ever asked to adjust the story size downwards, despite there being a number of tickets which didn't take very long despite being estimated with a higher points size.

After the first 6 Sprints, the average required velocity for the remaining 12 Sprints had increased to around 46 points, and we were concerned about achieving that velocity. However, because we had this insight early, my client was able to take the decision to invest in further capacity to the team, and they signed off hiring some more developers and a tester.

About halfway through project, the development team began to express concerns about their ability to estimate relative complexity accurately. They perceived that they were consistently underestimating the complexities of tasks and, as a result, were sceptical about the concept as a whole.

However, I analysed the team's performance in terms of relative complexity estimates versus the actuals to deliver each story. At that point, they had delivered 99 stories of varying complexities. Whilst I didn't show them the actual effort expended for each size category, because I didn't want them to subconsciously attribute a size score to a number of hours or days of effort, I was able to demonstrate that their assessment of relative complexity was actually *very* accurate.

That data is presented in Appendix 1.

What was also interesting was that their assessment of relative complexity was accurate right from the outset, but their work rate improved over time as they gelled as a team and gained in confidence. Their velocity at the end of the project was higher than the required velocity and earlier it had been below, balancing out across the whole project.

The team's perception of being bad at assessing relative complexity came from about six stories that were particularly problematic. It is human nature to focus on the things that haven't gone well, forgetting the vast collection of other things that have gone well. Even with those outliers in the data, the overall assessment of relative complexity was accurate. With those six tickets taken out, the accuracy improved further. In short, those outliers had little bearing on the overall accuracy of the assessment of relative complexity, but did colour the team's view of their own capability.

Over the remaining Sprints, the team's velocity rose to closer to 50 points and we remained confident of delivering "on or around the end of March".

Towards the end (last couple of Sprints), we found that the velocity dropped a little bit because as the remaining backlog was getting smaller, the amount of choice of what to work on for each Sprint was also diminishing and we found that some work had to be done in parallel where there were dependencies between those pieces. Earlier in the project we had elected to work on dependencies first, so other work wasn't blocked by them.

We also reached a point where our stakeholders, who had seen the solution regularly in show-and-tells throughout the project, were now also able to get their hands on it for themselves and use the system in development. That gave rise to some feedback and some small changes to the MVP scope.

Consequently, we added 2 more Sprints and delivered the solution into production 3 weeks later than the aspirational "on or around the end of March" that we had declared over a year earlier. This wasn't through slippage, but because of change. From Project inception to delivery was less than a year, and at the start of that period, 70% of the team had not been hired. The first customer went onto the platform at the start of May with very few reported issues.

All stakeholders were delighted. If you had offered me only 3 weeks movement on the date at the start of the 15 month project, I would have taken your hand off!

The team carried their velocity and their estimating approach into the next phase of work, which allowed an aspirational date to be set for that. At the time of writing that is progressing and broadly on track.

Benefits of Relative Complexity Estimating

Adopting relative complexity estimating provided several benefits for the development team at my client.

- The approach enabled the team to make estimations based on a relative scale, eliminating the need for a deep understanding of every task and any assumptions about "who" or "when".
- This allowed for quicker and more flexible estimations, reducing the time spent on detailed task breakdowns.
- It facilitated better visibility and tracking of project progress. By setting a target of achieving a certain number of story points per sprint, the team could gauge whether they were on track to meet their overall project timeline.
- This early detection of potential delays allowed for proactive adjustments and better alignment with stakeholders' expectations.
- It enabled us to manage our confidence levels and intervene early when we saw issues with our confidence in deliverability.

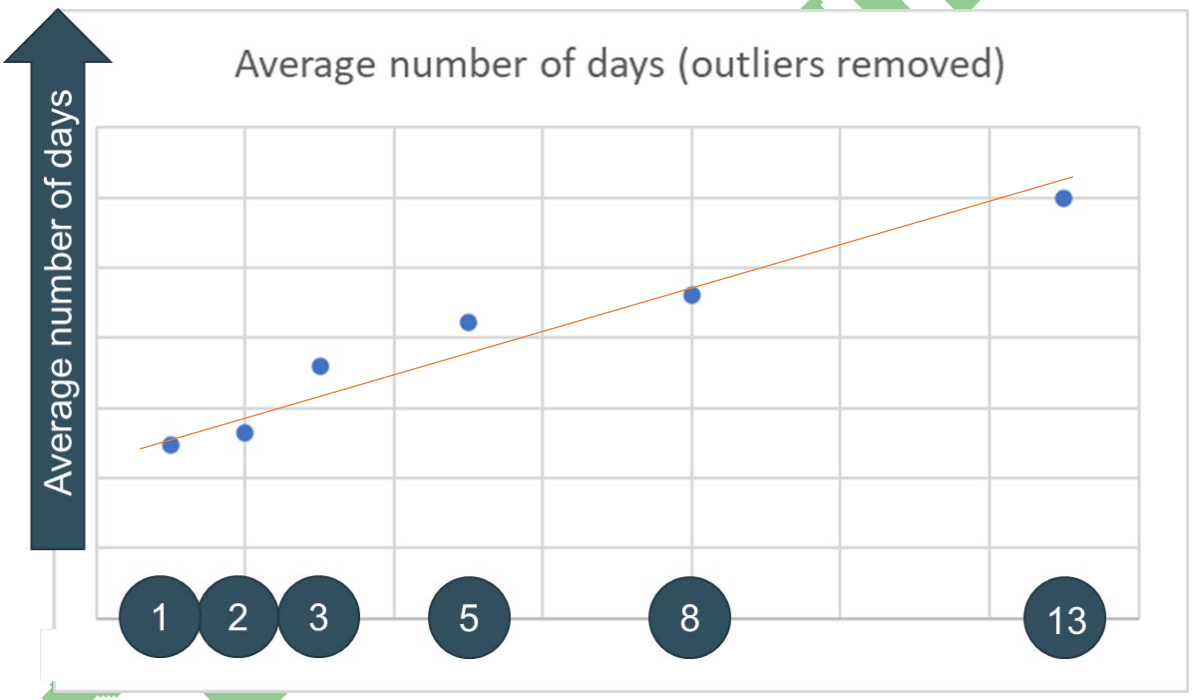
Appendix 1 – Data to support defence of team's estimating accuracy

When the team began to doubt their own capability to estimate well, I analysed their performance data to that point.

This is a plot of the **average** number of days taken for **each story point option** in our Fibonacci scale for every ticket with a status of "Done" as of 17th January 2023 (around 99 tickets), excluding 6 outlying tickets that spent considerable amounts of time blocked and had elapsed open times of between 70 and 106 days.

The distribution around the line of best fit shows that the relative complexity was well understood by the team when estimating.

The clarify, this graph shows that the work that was estimated at the start to be the simplest took on average the least amount of time to actually deliver, and the work that was estimated at the start to be the most complex took on average the most amount of time to actually deliver. The rest in between also followed the desired pattern. Therefore the estimation of relative complexity was accurate.



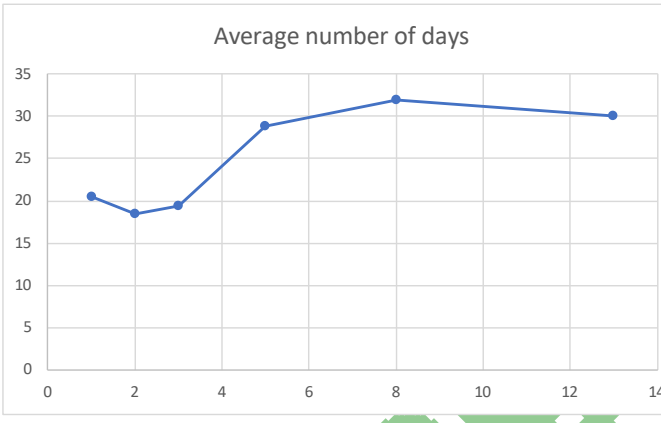
Relative Complexity estimating does not care how long each work item takes, only that each piece of work broadly takes more time than those considered less complex and less time than those considered more complex.

If we look at the affect of acceleration over time on this approach as well, we can see below that the relative complexity remained stable and well understood, but the actual amount of time spent on each size of story improved over time.

This table (below) shows the performance of delivery of the first 45 stories in the overall backlog of 99. In other words, the work done on the first half of the backlog. Whilst the average number of days for each story size is high, the relative complexity is still roughly accurate. The skewed 1-point plot is because some of the very early stories considered to be simple had a lot of pre-requisite work that needed to be done and this data analysis doesn't take into account "on hold", only "elapsed". Similarly, there was only 1

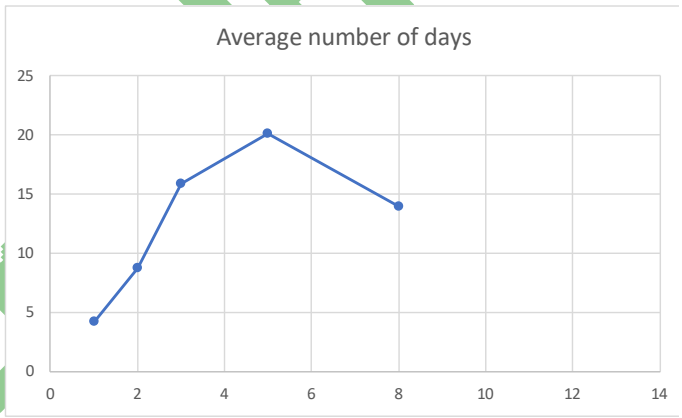
item of work sized at 13 points, and it so happened not to be as complicated as originally thought.

Story point size	Average # days
1	20
2	18
3	19
5	29
8	32
13	30



If we do the same analysis of the *last* 44 tickets in the backlog of 99, we see this:

Story point size	Average # days
1	4
2	9
3	16
5	20
8	14
13	Not applicable



Although the 8 point story average is lower than the 5 point one, it is explained by the fact that there were only 2 tickets sized at 8 points in this half of the backlog. They could just as easily have been sized as 5 points. Most of the stories were sized at 2, 3 or 5 points, and it was sometimes an arbitrary decision which score to give them.

The learning here was that the assessment of complexity relative to other stories was good right from the outset, and all sizes benefited from acceleration as the project progressed. This could not easily have been reflected in effort-based estimating where neither the developer who would do the work, nor the placement in the schedule for each piece of work, was known at the start of the project.