The Story Point

by Benny Bottema - Wednesday, January 24, 2018

http://52.48.96.241/2018/01/24/the-story-point/

Considering the discussion regarding story points keep popping up, I felt the need to comprehensively write down what I feel is the value of a story point. Everything is up for discussion.

Contents

What's in a number

A story point is an arbitrary measure of work by Scrum teams, which is used to measure the effort required to implement a story. In simple terms it is a number that tells the team how hard the story is, where hard is a combination of complexity and effort.

The numbers must be in the Fibonacci range, because as the estimation goes up so does the error margin. To compensate our natural false confidence in estimating the Fibonacci range explicitly doesn't allow for precise estimation; rather it goes up exponentially. Another reason why this Fibonacci range works so well is that complexity doesn't scale linearly, yet humans tend to assume so.

Story points are not directly related to time at all, *it is about ratios more than the number itself*. Story B might be twice the size of story A for example, the stories then can be estimated as 1 and 2, or 3 and 8 (*at least* double according to Fibonacci). A team might say: last sprint we did Story B, for the same amount of work we can do story C and D. There's no time involved here, just number ratios (or relative estimates).

The best thing about it is that we can have individuals with wildly different skill sets or speed agree on

the amount of work for a story. It is a universal estimation technique.

How a story point relates to time

If a story point is not directly related to time, then how *can* you tell how much work a team can do in a sprint? After all, a sprint is a fixed time box so we end up converting amount of work to a timeline, there's no way around this need for planning unless you do Kanban.

After a fixed time you can see how many story points have been delivered and the result is called the velocity, like 15 points per week. It's an indicator of throughput. However, you will never be able to tell how much time a specific story or story point will take. You can try based on past performance, but as the velocity is based on the team's average, including general external factors, a single story will never be entirely predictable. All you can say is: "for a sprint of 3 weeks, we are pretty confident we can deliver this much work." (but under promise and over deliver)

Velocity is never constant

Velocity is always team based, averaging the individual speeds. As long as the general setup of the team remains the same (1 senior, 2 juniors, 3 devs, 1 test etc.), the velocity should be somewhat stable and dependable, extreme external disturbances during a sprint notwithstanding. So if you swap out a medior with another medior, the change should not affect the velocity much in a team of 8 members.

Velocity is influenced by many factors, such as develop skills, number of developers, down time of TEST / Bamboo / SVN, communication issues, changing requirements, meetings, sickness, unexpected extra work etc. etc. By taking the average velocity of multiple sprints you can get a cautious suggestion of what a team is capable of in a single sprint. However, know that the environment is always changing, and a velocity of any given single sprint is really a snapshot and should only be used pessimistically.

Sprint zero

When a team just forms, to get a feel for how many points a it can do in a sprint, the team members don't plan at all; they just estimate a bunch of work. They just start and see how many points are finished at the end. That then becomes the initial velocity and serves as a basis for planning the next sprint.

How to estimate a user story

To be able to assign a number to a story point you need a baseline, a reference story. The numbers in the baseline are not really important, as long as you don't end up going above 100 with estimations. If a simple task is baselined at 55 story points, then a more complex task ends up being over a 100. This should be avoided.

User Stories vs (Technical) Tasks

Often, we poker once we have an oral consensus on a detailed technical on-the-spot analyses. It is only natural: team members and especially developers feel more confident about an estimate if they know

exactly what they need to do on a technical level. This is the reason why developers often rule the sprint plannings and other roles are present as witnesses rather than first class participants.

Ideally however, Scrum tells us we shouldn't estimate technicalities, but rather we should estimate functional pieces. After all a User Story represents a functional concept from a customer perspective ("As a user I can…"). These functional stories should be estimated without knowing too much about it technically. This way everybody's input becomes equally important. After estimating how much functionality we think we can do, the developers split up the stories into (technical) tasks.

I have never seen this approach in the wild, though. Somehow all the teams I have been part of gravitate towards technical analyses sessions after which a very precise poker planning is done mostly by the developers. My instincts tell me that the 'pure' approach only works for small companies where the technical challenges are smaller and integrations with other systems are fewer. The more complex a system becomes, the more you need to know about the technical details in order to give a realistic estimation of required effort in story points.

The definition of 'amount of work' / story point

It turns out the definition of amount of work is a moving target and not fixed at all. A team might say: "creating a new web service is 13 points", but it turns out teams don't stick to this definition in practice. In my experience teams invariably start to reduce the number of points the better or easier the task becomes. This is so, because the team *perceives the work as less work a*nd often it also really is up to a certain degree; it's not uncommon to hear the phrase: "oh I have done this before, it's easy now", or: "oh, we can just copy paste it from the other project, it's just 1 point", or: "we have never done this before, let's add 3 points". The problem then is how do we estimate using reference stories.

A reference story is a story *as if you do it from scratch*. A reference story is to be used early in a project. Later on, a team's implemented stories from previous sprints become the new reference stories and you will have a larger pool to compare to: "we don't have a reference story for this, but last month we did this for 5 points".

It's a false conclusion to say: "we have become more efficient at creating web services using copy/paste, so instead of reducing the number of story points, we state that our velocity is higher". It is false, because *the actual work is not the same*. Copy pasting is not the same amount of work as create a web service from scratch and so you shouldn't treat it as such. We should estimate the actual work involved, because that is what a story point is about.

Story points vs business value

Often I see that story points are interpreted as business value. They are not. It is entirely possible to realize 20 story points while offering no business value. The reverse is true as well: you can realize 1sp while offering serious business value.

Business value should be assigned separately if you want to keep track of it for reporting purposes or quick overviews. During backlog grooming sessions, where the user stories are prioritized in advance, a product owner will often have a good idea of the more important stuff, but it can help to have the business value stated explicitly. This is more for the product owner and internal business discussions rather than the estimating implementation efforts; the team just estimates amount of work and the product owner distributes it among sprints.

Another widely system used to indicate business value, or priority to be more precise, is a flag indicating *minor*, *major*, *critical*, *blocking* etc. Platforms such a Atlassian's JIRA has this built in.

Efficiency translates to velocity, not estimations

Leaving aside the 'actual work being reduced because of the copy/paste factor' class of arguments, estimations should never be modified because we became more efficient at some task. If creating a new web service always requires certain essential steps, it will be the same work every time, regardless how fast you have become at it. For the same reason you also shouldn't *increase* an estimation because "we have never done something like this before", or: "we have to do some research first". The amount of work remains the same, but we simply don't know how quickly we can do it. This will affect velocity, but not the story estimation.

Rather than changing the estimation, your team's velocity would automatically go up as you can do more of the same work in a sprint compare to previous sprints.

Especially when research is needed, you should rather pull it out of the story as a time boxed spike. You can't put story points on a spike, because every spike is different. Spikes therefore should not be taken into account when calculating velocity, but it does mean the sprint is shortened matching the time box (or assign story points in hindsight based on the same velocity used to plan the sprint).

Ideally you should always estimate comparing to either reference stories or stories from previous sprints. If you can't find an exact match that's fine, just try to relate your story in terms of effort. Try to avoid the automatic switch to gut feelings, because that's where a team starts to mess up estimates and include invalid value modifiers. If you stick to this, your velocity should naturally go up over time as your team becomes more efficient and experienced.

The reality is: the human brain inevitably relies on gut feeling for estimations

Unfortunately, sticking to reference stories or past stories is easier said than done. In fact so much easier that I haven't seen this followed through even once; teams always tend to forget about reference stories and rely on gut feeling a few sprints in.

The Fibonacci range alleviates some of the off-estimation impact due to this, but really it indicates an impedance mismatch between how teams operate -and indeed how humans think- and how Scrum experts classically assert how estimates should be done. To do this properly, you would need a permanent Scrum

coach to act as estimation police, which goes against the very nature of agile development. You don't want that.

As always the real point is to be predictable, so for a single Sprint relying on gut feeling this might be fine. It makes it difficult though to plan large releases many sprints ahead as many enterprise companies do, since the unanchored-by-reference-stories gut feeling is a moving target. It also makes it impossible to compare teams (more on that further down). One reason this occurs is because inherently teams are concerned with the next sprint and not some dead line in a far-off sprint. *This is by design*, but the company around the team often plans ahead much further and *is* concerned with this. This is the primary reason why at the end of six sprints, business at large can be surprised why the estimations were so far off. The Product Owner plays a key role here to keep business and development on the same page.

External factors influence capacity, not velocity

Some items you cannot estimate beforehand, such as production incidents: either these should be time-boxed or these should be updated during the sprint to match reality, based on the same velocity used to plan the sprint.

Story points are not about time, but about effort; production incidents take time but effort cannot be estimated in advance. The two are incompatible. If you get an incident that paralyzes two developers for a week, does it mean the team has a low velocity and the next sprint they should plan 20 story points less? No! Velocity remains the same, because the team's capability remains the same. Capacity however was reduced. For the next Sprint the same velocity applies, but you might want to reconsider a team's capacity based on external factors.

The reason I keep these two concepts separated is because environmental factors tend to be incidental, yet you want to know structurally what a team is capable of. Averaging multiple sprints alleviates the impact of incidentally reduced capacity of a team, though, but having a 'pure' velocity helps plan better in my opinion; it already is such moving target, let's not muddle it further by including the effect of sickness, you can't plan for that.

One way of dealing with incidental team disruptions is to still assign placeholder story points for incidents based on the current velocity (which effectively translates to reserving a timebox), but update during the sprint to match reality, another way is to reserve an actual time-box for incidents and reduce the Sprint duration / capacity (plan fewer story points).

An important goal of working with story points and sprints is about *predictability*. It is not a number game nor is it a measurement of performance. The objective is that a team can say with confidence how much work it think it can deliver in the next two-three weeks.

To be clear: it is not to say external factors are not part of a team's core responsibilities. Especially in DevOps obviously production incidents are part of their jobs. When it comes to predicting how much a team can do and how much capacity it has, however, velocity should not be influenced by production incidents. For example, a team can becomes twice as fast compared to when it started, even though they do half the work because there are more production incidents (capacity, however, is down to one fourth due to overwhelming external factors). Discuss!

How to compare teams

Because over time reference stories start to fade away in the background in favor of the actually implemented stories in previous sprints, the actual unit size of a story point within a team starts to shift ever so slightly. This is because teams start to rely more on 'gut feeling' rather than methodologically compare past stories or even the reference stories. In my experience this just happens to all teams. And stories never match exactly, so estimations are always relative. Because of this, doing a comparison between teams, even if their reference stories are the same, is like comparing apples to oranges.

Another reason to avoid comparisons between teams is because there are too many external factors in play. An absolute nono is to expect a certain performance from one team based on the velocity of another team!

Never make a team feel they should do more either, because velocity is easily gamed too. I've experienced it several times that a team's seemingly low performance was presented as a graph next to a team that performed well, only to have the first team increase estimates and 'game' the system. A team's performance is not bad or good, it just is. Let the team do its thing and if something is not right, it will come from the team itself (preferably during retro's).

> Use velocity to gain insight *about* teams, not to *steer* teams.
