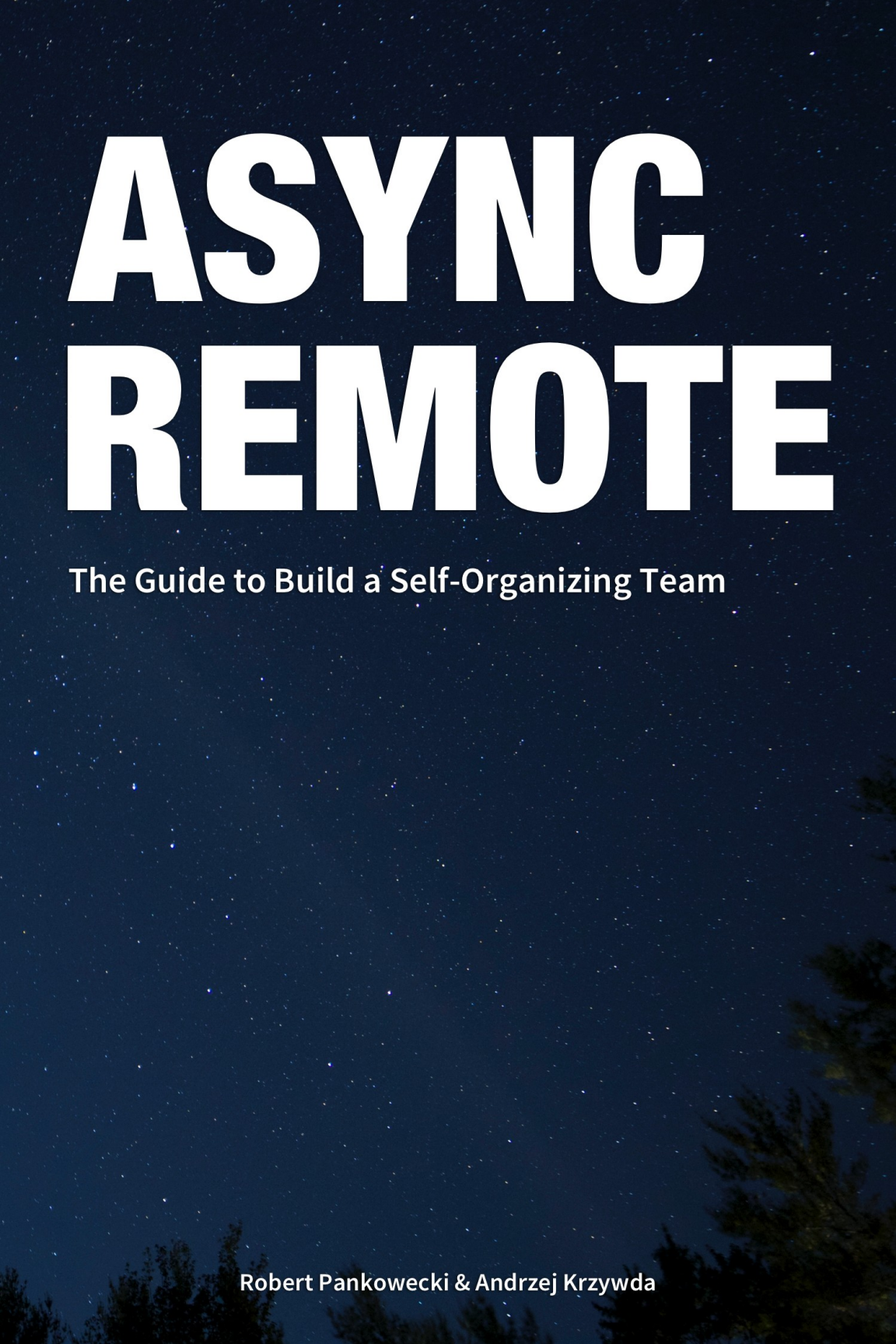

ASYNC
REMOTE
The Guide to Build a Self-Organizing Team
Robert Pankowecki & Andrzej Krzywda

Async Remote

Arkency Team and Robert Pankowecki

This book is for sale at
http://leanpub.com/developers-oriented-project-management

This version was published on 2016-04-04

Tweet This Book!

Please help Arkency Team and Robert Pankowecki by spreading the word about this book on Twitter!

The suggested hashtag for this book is #agile.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#agile

Contents

Story of size 1

A lot has been said about task estimation in IT projects. Every team and every tool seem to approach the problem from a different angle. You can estimate it in time (hours/days) or in story points (whatever it means). For example Pivotal allows you to set the size to 1, 2 or 3, which sounds very simple. Some methodologies recommend going with Fibonacci sequence i.e.: 1, 2, 3, 5, 8, 13, 21, etc. Others suggest: 1, 2, 3, 5, 13, 40, 100. Oh, and here is another one yet: 3, 5, 8, 13, 40, 100. I think that no matter which numeration appeals to you, you will eagerly find some justification for using them.

Except that there is no justification. Whenever you get to a story of size 5, you don't really have a story at all. All you have is some blurry vision of what you need, and as a result, you are unable to specify the task more clearly.

Those numbers mean nothing at the end of the day. Or perhaps I should say: at the end of the week, when someone asks why only 10 points were delivered this week when usually 20 are to be delivered. What do you need story points for anyway? To blame people for poor estimates or being lazy at the end of the week? To track if people are working? To guess what is going to be implemented next week, even if this is still just guessing?

And what happens when you have estimated them wrongly? Will you change the original value when marking the story as finished? If you do (and I know teams who do), you will always end up with about the same number of story points per week, because in fact, you are just using the points to track time and you work about the same amount of time every week. So it brings you no real value whatsoever.

Because of all the hassle related to story points, we have taken a different approach. We try to create stories of size 1 only. It's the

smallest task that still brings business value. If you can split the task into 2 subtasks, and they still bring value to your business, then it is not a size 1 story - *the story should be indivisible*. Let's have a look at some of the benefits of this strategy.

It's easier to track progress. No matter what tool you use for that, after splitting tasks into very, very small subtasks you can track progress easier. If you split a ticket of size 3 into two or three stories you will be able to track it better. Instead of being notified that something is done after 1.5 days, you will be able to see part 1 finished within the first few hours. Give it a few hours more and the second part will be also done. By the end of next day you will know that the third part was also finished. You won't feel the urge to interrupt anyone and ask for the status. Updating status often and easily is one of the most effective forms of asynchronous overcommunication.

Marking tasks as *done* is rewarding, refreshing and motivating. Some teams give programmers tasks that are going to take a week or longer. It is very tiresome for developers to work on such features for a long time. And there are many reasons why:

- People start feeling alienated from the team. Alienation often happens when developers work on distinct parts of the system and have no need or reason to communicate with each other.
- Every day you put more and more effort into the task but it still is not finished. You can neither say it is *done*, nor that it is *undone* . Just *started*. When someone asks for the status, the developer must either go into details to describe the progress or simply state: *I am working on it*. Either way, having unfinished tasks at the end of the day is stressful for both the developers and the managers. I would even say that tasks longer than 1 week are detrimental. They are so big, involve so many tiny changes and are so time demanding

> that you for the most part while working on them can't see the end of them.

When you deal with lots of small tasks, the contrast is very sharp. You take a task, spend a few hours on it and it is *done*. There is no state between. No *"70% done, boss"*, or anything like that. You take, you do it, you leave it, you forget about it.

Working on smaller tasks related to different parts of the project help propagate knowledge. When you have people working for too long on specific parts of the project, *Collective Ownership* declines. People specialize in modules or parts of the codes such as payments, billing, backend, frontend, etc. … And later you end up pairing people with tasks that seem to be suitable for them based on the knowledge they have, because you think the job is going to be finished faster that way, but that's not what happens. What happens is that people look at the same code again and again and it is their own code. And you can't really learn by reading your own code. You don't learn anything about code readability when others don't read it and have no chance to understand it. You don't know the quality of your tests when others don't start changing things and see what broke. And when you keep working on the same code for a long time, it becomes boring. And the code starts to be filled with your personal patterns that suit you very well, but might not be understandable by other team members. So working on smaller tickets brings fresh eyes to different code areas and helps catch bugs and improve readability. Also because people constantly read each other's code (in practice, when working on it, is not the same kind of thing as reading Pull Requests and saying *OK*), they synchronize their mindset. They learn from each other, mentor each other and establish best practices.

Keeping stories small makes people more mobile across different projects that your company is currently working on. It means that when a project is progressing slowly, while another one doesn't need much manpower at the given moment, you can shift your

developers more freely between the projects. It's way less cognitive overhead to start working on a project story that is going to take 2-4 hours vs. joining a project only to find out the work involved will take one week, And there might not even be enough "flexible time" to delegate people to a different project for a whole week. One or two days to help in a different project is usually OK, however, for any programmer to provide value in a new project in such a short time you need to have small, self-explainable, understandable tickets to begin with - a minimal overhead to join the team.

With small stories, it's also easier to jump onto them during a busy day. Let's say you're having a doctor's appointment or a meeting? No problem. You can still take up a small task later that day and deliver something of value to the business.

Let's now consider a simple example and see the difference for ourselves. Imagine we have got task A estimated as 5 points and task B estimated as 3 points. We look at those tasks carefully and split into multiple small stories: A1, A2, A3, A4, A5, A6, A7 and B1, B2, B3. Some really interesting things might happen as a result.

You develop a better understanding of what this task really is about and how time-consuming it is. This gives you a better base for estimating and tracking it properly.

You get a chance to prioritize things more accurately. Instead of just saying *"do A and then do B"*, you can now prioritize it as follows: A1, A2, A3, B1, B2, A4, A5, B3, A6, A7, B4. So you might find out that different parts of the task bring different value to your business. And that the sequencing of tasks is a little more complicated. Maybe by the end of task A6, you'll have come up with something entirely new, (such as some new business opportunity C1) and prioritize it over A7 and B4. And when you do so, all other things are already on production, providing you value. The developer won't say *"I am halfway through a really big task"*. You won't have to choose between dropping task A completely and doing C instead. You can manage your priorities on a daily basis.

Having small tasks minimizes your risk of not delivering. If the task is harder to implement than initially projected, then it's only a difference of between 4 and 8 hours, not a week or two weeks. So everybody is getting the feedback from the tasks quickly. And in case of personal misfortune, when someone gets sick or injured, the situation is easier to manage. You simply don't get stuck halfway through a long rewrite of a module as your key developer suddenly bails out.

And the most obvious benefit - when things are delivered swiftly, the business profits faster, while the feedback loop is shorter.

The only downside I could identify is that implementing things in such way might take longer time. However, it is my personal opinion that the price for having small, working, deployable value-providing tickets is very small, if not non-existing.

Links:

- Don't Take Partial Credit for Semi-Finished Stories[1]
- What is a story point[2]
- Twenty ways to split stories[3]
- How to split a user story - flowchart[4]
- Zeigarnik Effect[5]

[1] http://www.mountaingoatsoftware.com/blog/dont-take-partial-credit-for-semi-finished-stories

[2] http://agilefaq.wordpress.com/2007/11/13/what-is-a-story-point/

[3] http://xp123.com/articles/twenty-ways-to-split-stories/

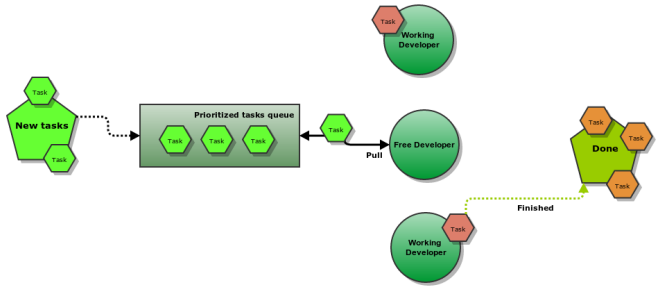
[4] http://35qk152ejao6mi5pan29erbr9.wpengine.netdna-cdn.com/wp-content/uploads/2012/01/Story-Splitting-Flowchart.pdf

[5] http://www.psychwiki.com/wiki/Zeigarnik_Effect

Stream Of Work

Leave tasks unassigned

The most common way to manage a project is to take a bunch of tasks and assign them to developers at the beginning of the week, and then, let them work on the tasks for the rest of the week. We believe this is not the most effective model, based on both our experience and on tasks scheduling theory. We suggest that you leave your task unassigned and let developers start working on it when they are finished with other tasks. Instead of pushing tasks onto them, you want your developers to pull them from the prioritized list.



The webserver analogy

A similar subtle difference emerges in case of webservers with multiple workers where one task takes longer than others. In push model, when one worker is busy processing the request, they can still receive next requests which will queue waiting for the currently processed one to be finished. In pull model, the first available worker will take the first request that needs to be served and the queue of requests is global for all workers, and not worker-specific.

And this is a good analogy as to how you might want to manage your project and split tasks between developers.

The first reason for that is exactly the same as in the webserver example. You do not exactly know upfront how long the task (request) is going to take to be finished (processed). Yes, you do estimate things. And yes, you do it wrongly. Just as everybody does. Case in point: two developers (John and Martha) are assigned two tasks each. John is working on tasks A and C. Martha is supposed to be working on tasks B and D. The priority is A, B, C, D, and task A proves to be more complicated than originally assumed, so John is still working on it. In the meantime Martha has finished working on task B. If she goes with her assigned task D, she will be working on the task that is not the most valuable for the business. After all, a higher priority task C is left to finish, and she could start it if it was left unassigned. This is not rocket science. It's what you hear at first classes of *tasks scheduling* courses. And still many projects choose the ineffective way.

Availability and efficiency

You might not know or be sure about the availability of your developers. In some projects, you've got people working for you full-time so you think you know their availability. But even in such projects the efficiency of developers can differ. It does not matter how many hours people work. What matters is how much they can accomplish during that time. And this varies greatly. Sometimes people feel good and motivated. Sometimes they are worried about some recent events and more easily distracted. Not to mention the fact that random events happen. People get sick, accidents occur. Welcome to real life.

In our company people are usually assigned to two projects at the same time, so this is especially important for us. We work on two projects so that people have access to more broad scope of tasks and experiences. And when they feel tired or bored by the situation

in one project, they can use the other one to rest. We will work at least the minimum of hours we have agreed to with our customers (usually more), but we cannot guarantee the exact number. Our developers are free to schedule their time and decide how much or when they want to work within their week (though with the minimum number in mind). So we prefer not to have our tasks explicitly assigned, but rather jump into the project, take the most important task to be done and deliver value quickly. So this model allows us enormous amount of freedom and flexibility.

Quit the self-reinforcing loop

Many project methodologies advise you to let the team assign the tasks. What I see mostly in practice is that tasks are assigned by the same person over and over again, usually by some kind of project manager. Doesn't matter. The effect is the same in both cases, because people are naturally flawed. The managers will usually assign tasks to people whom they personally find most qualified. Developers will try to assign tasks that they feel most comfortable with. In the long run, people end up specializing in particular code domains, while your *Collective Ownership* declines.

The only way to escape this self-reinforcing loop is to assign tasks differently, changing the criteria. You don't need the same person working on the same module or part of the code again and again. Let people try something different. But there is one crucial element we already learnt in the previous chapter: the story must be small. If you have a small task that might take two hours to finish by your frontend developer, but you give it to your mostly-backend developer, they might be curious about it and willing to finish. Real tasks are great learning opportunities. It's my personal conviction that programmers are always willing to learn and improve their skills. But do not expect a warm welcome if you try to assign a challenging week-long task. Allow people to start easy - the smaller the tasks, the easier it is for people with varying tech-backgrounds

to finish and learn from them. And should you refuse to go that way, you might put the whole project at risk if one of your developers leaves the team for any reason, be it a different project or company or just vacation or a sick leave.

Take the first task

So let's say you've taken heed of our previous advice and as a team, you're now facing a list of unassigned tasks. As human nature dictates, developers will still be tempted to pick the easiest or most compelling tasks from the list. And that is exactly what we want to avoid. So there is one simple rule that everyone must follow for this whole system to work: *"Take the first task"*, where *first* means an unstarted task with the highest priority. The developer should spend no time thinking about which task should be done next when grabbing new story to begin working on. Just look at the list and see what is on top. It should be a no-brainer.

Do not estimate

I must mediate as to what is written in this part.

What do you have in the end?

With one-point stories, unassigned tasks, and developers taking the first story you end up with a smooth, and more scalable system to distribute the work. You do not rely on estimates and guesses. Developers can more freely juggle projects that need attention with their personal lives. There is very little overhead to join the project and to deliver small, but measurable value.

Instead of tasks being pushed to them, the developers themselves are pulling them out when they are free to start something new. Developers are always working on the feature that has been ranked highest in priority based on current state of knowledge.

Links

- Fred George - Agile is the new black - Railsberry 2013[6] (Vimeo)
- Why are software development estimates regularly off by a factor of 2-3 times?[7]
- Software projects have an inherent bias towards unpredictability[8]
- The #NoEstimates movement[9] (xprogramming.com)
- 'No Estimates' in Action: 5 Ways to Rethink Software Projects[10]
- Peopleware[11] (Wikipedia)
- Peopleware: Productive Projects and Teams[12] (Wikipedia)
- Peopleware: Productive Projects and Teams[13] (Amazon)
- Collective Ownership[14] (extremeprogramming.org)
- For Workers, Less Flexible Companies[15]

[6] http://vimeo.com/68689393

[7] http://michaelrwolfe.com/2013/10/19/50/

[8] http://gist.io/6949301

[9] http://xprogramming.com/articles/the-noestimates-movement/

[10] http://www.cio.com/article/print/742684

[11] http://en.wikipedia.org/wiki/Peopleware

[12] http://en.wikipedia.org/wiki/Peopleware:_Productive_Projects_and_Teams

[13] http://www.amazon.com/Peopleware-Productive-Projects-Teams-3rd/dp/0321934113

[14] http://www.extremeprogramming.org/rules/collective.html

[15] http://www.nytimes.com/2014/05/20/business/for-workers-less-flexible-companies.html?hp&_r=0