Agile 101

Practical Project Management with Scrum

Antonio Martel

Edition (English)

October 2016

Version

3.0

Translation

Begoña Martínez

Translation Editor and Proofreader

David Nesbitt

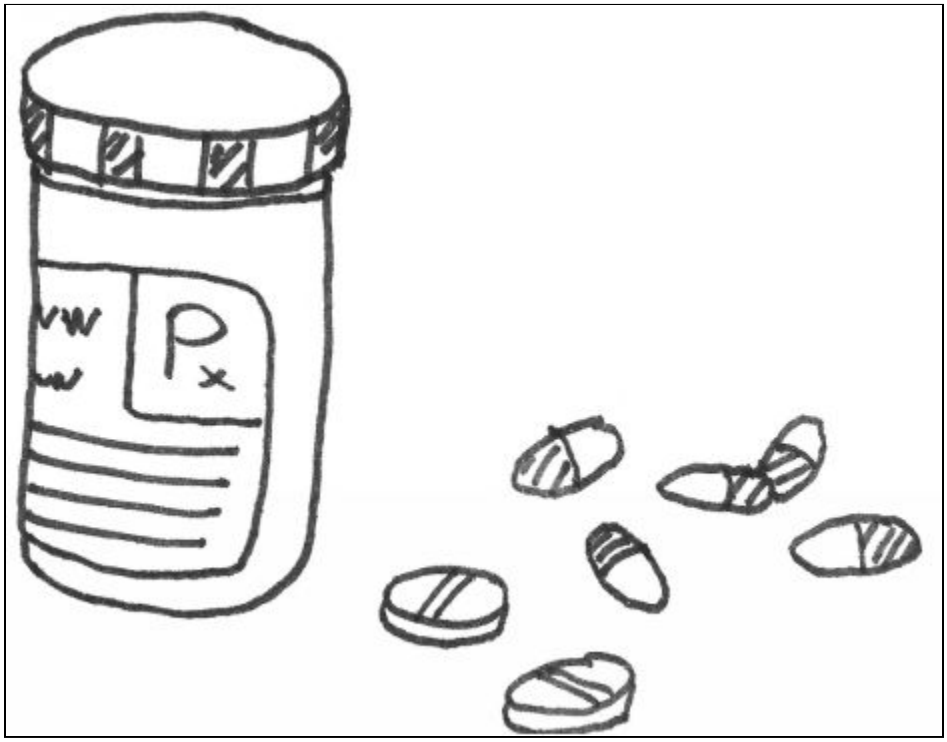
Cover and back cover design

Alex Caja Almonacid

Rights

INDEX

Foreword - Hey, Scrum worked!



"The real reason for use of pressure and overtime may be to make everyone look better when the project fails."

Tom DeMarco

That's the first sentence in Henrik Kniberg's book Scrum and XP from the Trenches: "*Scrum worked!*" It has also worked for us, the teams that develop environmental and safety (Police) projects in the department of Public Administrations of a Spanish software consultancy company.

It's difficult to quantify how much, but ever since I started using Scrum back in 2009 (yes, I know, I wasn't an early adopter), we've seen a steady increase in successfully finished projects, not only in our case, as we're the software providers, but also for our clients. They have received products that grow step by step and adapt to their real needs.

Delivering part of the work to the client every two weeks and asking the client what they want us to work on the following two weeks has given our daily work routine a certain peace of mind. This relative peace has helped us, in part, to get rid of that twinge of chaos that appears when great deliveries have to be made at the end of a project. It reduced stress for developers, but it also gave peace of mind to all the stakeholders, who could regularly see what were we working on and how the developed project looked.

Working for the public administration wasn't always easy. Even if they are normally reluctant to try new work methodologies, we were also surprised by their flexibility. Indeed the road from 2009 to today hasn't been easy. I've made lots of blunders and I've had to learn difficult lessons along the way, but I think the result has been worth it. From that year to the present day, Scrum has become the recommended methodology for many projects. European and International top job portals are full of ads looking for "*Agile*" experience, and I'm happy to say that, increasingly, Scrum is the requested methodology in technical specification sheets in the Spanish administration public tenders, and in the RFQs of European international projects.

The road I followed to learn Scrum and Agile has been difficult but gratifying. I foresee many more years of guiding projects to a successful finish.

INTRODUCTION

"*A bad beginning makes a bad ending.*"

Euripides

Your company has just appointed you project manager. You've heard about Agile methodologies, Scrum, Kanban, and lots of other things you'd like to try. You've started reading all about artifacts, principles, and manifestos, but nothing is crystal clear just yet.

You don't know whether your lack of knowledge will put your project at risk, but probably what worries you the most is that all these things are nothing but buzzwords, the current fad, and that they won't have any real effect on your day-to-day work… or even worse, that such a big change in what to call things and how to do them will make your team, your clients, and your company reject those ideas altogether. Luckily for me, or unluckily maybe, I have also gone through all these problems, so in this book I will tell you how I solved them (when I've been able to) or, at least, what I tried and what eventually failed.

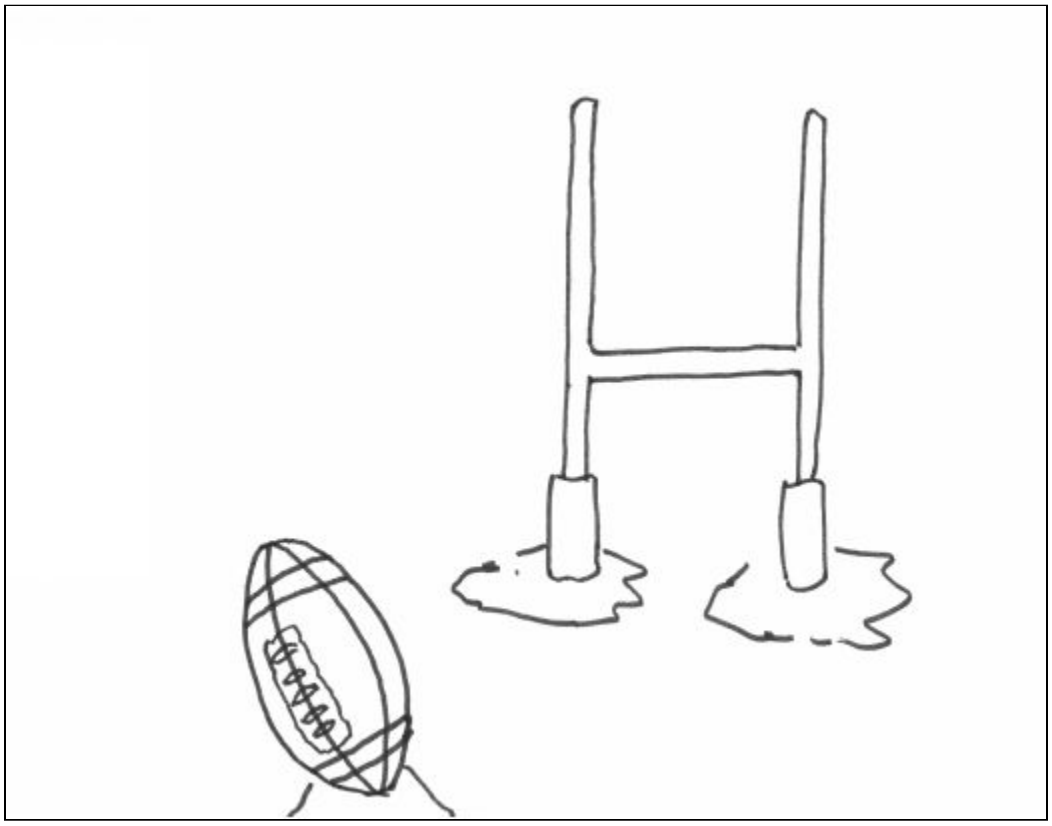
Don't expect this document to be an exhaustive Scrum guide. I don't know everything about Scrum, and the implementation I follow is far from perfect. For a thorough knowledge of Scrum, I recommend following some of the popular Scrum guides that you can find either online or in books like the one by Henrik Kniberg, not to mention the one by Jeff Sutherland, creator of Scrum.

Even though I follow a survival Scrum style and I still have lots to improve, Scrum has worked for me. Using it, we have managed to gain credibility in difficult projects, projects that were not working the way we expected. We managed to reduce the amount of stressful situations that, luckily, now happen far less often, and we have increased the satisfaction of the clients who, sprint after sprint, get their money's worth. We have not been able to follow Scrum methodologies to the letter in every project we have worked on, but just trying to be more Agile we have improved our results in the projects we are working on.

When we start using Scrum, we all tend to use the terminology of this framework, talking about daily sprint meetings, product backlogs or burndown charts. We follow our Scrum guides to a tee, when in fact no one understands them, not even ourselves. Not understanding the spirit behind these rules, we end up creating a waterfall project—like we always have done—disguised by jargon that makes us look more innovative.

Rather than telling you how to do things, or explaining in detail how many meetings you should have and how many minutes each one should last, in this book I want to describe what the purpose is behind those meetings, what the artifacts described in Scrum guides are for, and what you might get out of them. I'll help you understand the Agile principles that underlie this way of working and that really make this work. I hope you find this book very useful!

BEFORE WE START, LET'S SEE WHAT SCRUM IS



"How do you manage a project when you have received no previous training in project management? Well, by suffering a lot."

Albert Cubeles

Scrum is a framework or a set of good policies for project management. It is based on a study carried out on development processes successfully used in Canon, Xerox, Honda or HP —which in the ‘80s were the Google or Apple of the time. The findings of this study indicated that these teams had very general definitions of the scope of their products and that they had to ship them in very little time. They were highly productive teams which followed very similar work patterns.

In 1986, Takeuchi and Nonaka, authors of this study, noted down all the common patterns they could find and defined this way of working as a unit that passes work from one to the other so that each person does their bit, somewhat like in rugby when the front line passes the ball from one person to another until they score a touchdown. It is from rugby terminology that this framework ended up with the name Scrum.

In the early ‘90s, for the very first time, Ken Schwaber and Jeff Sutherland presented a description of the methodology that today is known as Scrum, which summarized their experiences and the best industry practices.

Scrum is based on partial and regular deliveries of the final product, starting with the features that are most important for the client. Scrum is thus recommended especially for projects where early results are necessary, or where requirements are constantly changing, and competitiveness, flexibility, and productivity are key (and where are they not?). This prevents deadlines from being exceeded for longer periods, and allows reacting before costs go through the roof.

The way of working in Scrum is based on artifacts (or tools, I’m afraid I don’t like the word “*artifacts*” very much), on a series of meetings that have to be convened throughout the project, and on some roles that the project members must play.

Artifacts or tools

Product Backlog: It is simply a list of tasks to be performed. It will specify, broadly, each of the features that the final product must have. This list must be defined and prioritised by somebody who is aware of the needs that our product will address, or someone who represents the stakeholders, those with a vested interest in making sure that the project works (the product owner).

Having this task list available in a visible place and going through it every now and then will allow us not to lose perspective on the amount of work pending. Striking off items from this list will give us a feeling of progress, that we are nearing the end of the project. It is a feeling similar to the one you get when you are studying, and you jot down on a piece of paper how many topics you should review for an exam. Crossing out a new topic every time you are finished with it makes you want to start the next one so you can cross it out, too. It also allows you to see at a glance how much you still have to do and whether you can finish everything on time before the exam.

Sprint Backlog: From the long list we have defined before, the Product Owner will shortlist some tasks that the team can do in the next 2 weeks. That fortnight is called a sprint, as during that time we are going to run a short race to prepare and test all the tasks that the team committed to finishing during that period. The usual duration of a sprint should be from 1 to 4 weeks.

Deliveries every two weeks will make our work easier to tackle, as compared to just one big delivery at the end of the project. Establishing small, frequent milestones will will keep us from getting winded when faced with the amount of unfinished work. Can you imagine how it would feel to pay just one mortgage instalment a year instead of every month? Many would be reckless and would have to tighten their belts the last few months, scraping together a few coins here and there to make the payment on time. Others would put a small amount aside every month or every week, so that they could reach the deadline without last-minute distress. This philosophy is quite similar to the one we follow with partial deliveries in every demo of the sprint.

Burndown Chart: When the Product Owner, in the Demo Meeting after finishing a sprint, authorizes some or all the tasks that had been scheduled, sets the estimated pending work to zero, so there are fewer tasks to complete and the chart goes down (that is why it is called burndown). When the chart hits bottom (there are zero pending tasks in the Product Backlog) we have finished the project. Congratulations! (It looked like that day would never come.)

In my experience, the Burndown Chart is not only helpful to see whether a project is going well or not, but it also motivates the team to improve the slope of the graph. If in the current sprint the graph had gone down a lot, the usual attitude would be "*if this is OK, watch what we do on the next sprint."* If, however, the slope in the graph was quite horizontal instead, the team would say "*the work for the next sprint is not that hard —we'll be able to finish all the stuff we couldn't finish for this one, and also add all the points of the next one.*"

Roles

Product Owner: This person establishes and prioritizes the list of features that should be developed in the project (the Product Backlog) according to the needs of the people who pay for the product. The Product Owner represents the client's perspective and needs. This person is in charge of describing the tasks or features listed in the Product Backlog and in the Sprint Backlog. This role is normally played by someone chosen by our client to represent all the people interested in having a final product that is useful to them: users, managers or executives in the client's side —that is, the stakeholders.

A Product Owner who clearly defines the functionality required of the product, who knows the business well and sets objectives that are clear, straightforward, and free of unnecessary complications… is priceless!

Scrum Master: A Scrum Master is not exactly a traditional project manager. That role is shared between the Scrum Master and the Product Owner. The most important mission is protecting the team from

interruptions while they work to complete the sprint and helping to overcome any setbacks that might prevent the team from reaching the sprint goal. The Scrum Master will prepare the meetings and make sure they are productive. He or she will also assign tasks to the members of the Development Team and will follow up on those already assigned.

The Development Team: They are the members responsible for delivering the product. At the end of each sprint, they must deliver the user stories, reviewed and checked. Otherwise, they will fail in the Demo Meeting (and we'll all feel a bit embarrassed).

Meetings

Sprint Planning Meeting: Before starting the work that has been planned for the following 2 weeks, you will meet with the Product Owner (and with the people from the client's team that they consider appropriate) and set the priority level for each task that must be completed in that time (if the team thinks that the tasks can be done in that period of time). The Product Owner will provide details on the chosen tasks, explaining anything to the team that might be necessary to carry out the work.

If the work capacity of the team is taken into account, this Planning Meeting will be able, amongst other things, to fit production to the holiday periods of the team members. It's as simple as explaining: "*In two weeks we promise to deliver just 3 of these tasks, instead of 5 as we normally do. Alberto and Pilar are on holiday next week.*" That is normally not a problem because the Product Owner would have been receiving product deliveries after each sprint in a consistent manner.

Daily Meetings: The Development Team and the Scrum Master will meet for 15 minutes every day, preferably early in the morning, in order to answer the following questions:

□ What has been done since our last meeting?

□ What needs to be done before our next meeting?

□ What is keeping us from doing the best possible work?

I normally work as the Scrum Master in several projects and to each of those daily meetings I take a printed sheet with these three questions, where I record the answers of the Development Team. At the end of each sheet I add a section called "Tasks for the Scrum Master." There I can note down everything that the work team needs in order to go on, or what is making their work difficult. Some common examples of these notes: "*Call the client and get the client list (tomorrow we start that task!)*" or "*Ask for an export of the database (we need a copy before we start crunching data).*" It is frequent for me to spend the rest of the morning trying to solve these problems.

Demo Meeting: After we finish each sprint, a meeting is set with the Product Owner to present a demo of the work that has been finished in the previous two weeks. In that meeting, the Product Owner will review what is being shown and will decide whether to sign off on it.
In order to mark a feature as done, it is important for it to be completely finished and error free—a number of tests from the Development Team and a full validation from the client are necessary. If we didn't do

it this way, we would get to the end of the project as per the Burndown Chart (every task would be at 0 pending hours) but the project wouldn't be finished because some features would remain incomplete or have errors to be corrected.

Regular deliveries allow clients to see how the project advances as well as what has been done in each sprint, so that they can decide later, in the Sprint Planning Meeting, what they want to see in the following Demo Meeting. This helps build trust with the clients, instead of the letting them lose sight of the development company once the requirements are set, and meet months later with a finished product that would be very difficult to modify.

Retrospective Meeting: After each sprint and demo, the Scrum Master and the Development Team meet to look at problems that might have come up in the previous two weeks, to find out why the demo meeting didn't go as well as planned (or maybe it did!), and to figure out whether there's anything that could be improved for the next delivery.

It is constructive for the work team to feel free to describe what they consider to have been the main problems, even if that means a bit of embarrassment for the Scrum Master. Problems normally appear in our "*blind spot*" as Scrum Masters and we need someone to help us see what is wrong.

Principles behind the Agile Manifesto

Nearly 15 years ago, in 2001, 17 software developers met in Utah. They were critical of the then popular software-development models, which they considered rigid or heavy. That meeting included people like Kent Beck, Martin Fowler or the Scrum founders. They had decided to meet in order to talk about the new techniques and processes to develop software.

That meeting gave rise to a number of principles governing the new alternative (and Agile) methods they were proposing: the Principles behind the Agile Manifesto.

One of these principles is:

Our highest priority is to satisfy the customer
through early and continuous delivery
of valuable software.

Clearly, this is what should be done in any type of development. We start every project with that in mind, but soon people start getting into a rush. Our director asks how many hours we've spent (and we've spent a lot already). Our client asks when all the features will be ready. Our workmates ask if they can take that two-week holiday we owe them.

It starts to look ugly. We're in a mess again. We have to redo the Gantt chart with the new delivery deadlines, we have to look for another set of dates for those holidays (even our own holidays), and we have to give the director loads of explanations. "*The client requested lots of changes.*" "*The technology was new.*" "*We miscalculated.*"

Here's where we should be firm and ask the team for some extra effort, and decline little changes requested by the client. If it isn't written exactly like that in the contract we signed or the minutes of a meeting we had six months ago, we're sorry but we can't do it.

The thing is, clients also did this. They wrote a very clear contract in which they stated each and every feature that they wanted (or the ones they wanted when they drafted it). Maybe they no longer needed those features or had realised that there were other, more important things, but they're in the contract and they should be done. The project cannot end and leave 30% of undeveloped functionality.

At the moment, we have lost sight of what should be our top priority in our job: early and continuous delivery of valuable software. From here on, there are just tough negotiations and a contract that we would try to fulfil as soon as possible with the least amount of damage possible.

The contract might have tonnes of clauses and stipulations, but regardless of what it says, what the client wanted on signing it was a solution to the problem at hand, not an argument about whether to implement one functionality or another.

But if we deliver our software early and often, we can let clients test it and use it and tell us what they think. We can allow them tell us if there is something missing or what could be improved. They will want us to implement things they find they now need, and they will be delighted in turn to take out that *.rpt* file-exporting feature that no one remembers requesting or knows why it was requested in the first place.

When we finish the project, the client will have a product that really solves their problems, one that has been evolving while they learned, and one that they have been able to use and test from the early stages of the project. It sounds better for both the provider and the client, right?

Requirements might change

A few years ago, before completely applying this whole Agile thing, I worked in the development of a web application that should manage the client's worksheets through a series of steps, a wizard that would lead users through the phase they were in.

We made a full analysis, we collected all the client needs, and from there we compiled a list of features to be implemented. All very normal. Correct, coherent features that solved specific problems. With a thick requirements document and their respective designs, we started to implement these features, one after another.

Back then, we tried to be a bit Agile (in a way) and every two weeks in pre-production we showed how things were going. It didn't look half bad. It was easy to use and we even got congratulated by clients on the work they were seeing.

We were happy like that, so we went on until we developed, with no small effort on our part, all the features that were included in the analysis document that had been approved. And we deployed them into production…

And as soon as the administration staff used the application for a few days, they realized they could have done with some changes: include a table in each screen with the original documents, so they wouldn't lose sight of them; avoid going out to another menu when creating an entity; adding more controls at the end to avoid mistakes when filling them in, etc... Most of them were minor changes that wouldn't take more than two or three days of development, but we had already exhausted our budget (and more).

Amongst the features we had developed from the beginning, there were some complex ones that allowed complicated changes between workflows and that were difficult to develop (and test, in order to be able to check every possible case). I asked about them months later but nobody knew anything about them or which menu they were in. It wasn't considered important. When they had needed something like that, they had cancelled the process and started a new one.

When we started the project, neither I, nor the client, nor the users had an exact idea of what would work well. We simply couldn't know for sure. What could have been done to avoid this? Well, we should have deployed the basic features to production as soon as they were ready. That way we would have easily seen all these little changes that would have made the application much more practical and easy to use.

From that moment on, in new projects, when users see the need for changes like these, they ask me about the possibility of including them. They have no problem giving up other features for which they no longer remembered the reason for including in the initial contract or that no longer made much sense.

The feature list is drafted from the beginning in the Product Backlog. The client is aware of this estimation and simply says: "*Take out feature 18, which would take 8 days' work, and change it for these other two features which would take 6 days. We save those two days in case we need some other change.*"

That is another Agile principle:

> *"Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."*

In general, it is advantageous for clients because they get a product that works and is not broken until a new contract for corrective maintenance is drafted (which would involve paying extra money). But it is also an advantage for us as providers, because we offer a project that is a good reference, in which we are aware of and can account for every hour we spend, and we avoid delivering a product that depletes our budget but still lacks important features.

“Yes, but that’s no longer of use to me. Now we have to change all this.”

SCRUM'S PROS AND CONS

"There is nothing more difficult and dangerous, or more doubtful of success, than an attempt to introduce a new order of things."
Machiavelli

Disadvantages of using Scrum

How many technologies have quickly popped up and then vanished even quicker? They all promised to be revolutionary, the new silver bullet that will end every problem in the software industry. When we get as much hype as has been lavished on Scrum and Agile, our first reaction is to raise a critical brow.

If you are adopting Scrum in your project or company, it is best for you to be informed beforehand about its drawbacks:

The team might be tempted to follow the shortest route

When we have things to deliver every two weeks, and in the latest deliveries the projected functionality wasn't ready and the speed of the team does not indicate that we will finish the next one on time, we have a problem. We can succumb to the temptation of finishing the pending tasks in any way we can and leave a "*technical debt*" behind us. Everything has the appearance of going well, we create more features, we start new things, and the client is happy because deadlines are being met.

But every time you leave a small hole of a deficit behind you, I can guarantee that you will stumble on it again and make it bigger. If new things keep caving in this way, the "*debt*" will become a pit. Sooner or later you'll have to stop everything and fill in the pit—pay back that loan and with extra interest because of that late payment. The project doesn't finish when we were so close to the end, and the *burndown* chart appears to have a horizontal asymptote at 0 (sorry again, Math 101 Calculus).

Do you need to have delivery deadlines way in advance?

This is one of the main criticisms that Scrum normally receives. In a way, it is logical that Scrum cannot give you those deadlines. At the beginning of the project, you cannot predict when you're going to finish, if you're making it easy to change the thing being built as time goes by. But, do you prefer a product that you know "for certain" will be finished in 12 months, but is built over the ideas and opinions you had a year earlier? Maybe you prefer a product that could be finished in a similar period of time, but that you have been able to steer towards your real needs, and you've been able to use and test before the final delivery deadline.

Stress!

We cannot sprint for our whole lives. We deliver something and the next deadline is just two weeks away. Then another one, and another one. If we have to run like this for many miles, we will *sprint* really hard for the first deadline, but we will walk for the last one —and that, with luck. It is something we should bear in mind during planning, from the start.

Is your team able to self-organize?

One of Scrum's principles is that the team members must make their own decisions and self-organize. Also, one of the requirements is to avoid having teams that focus just on specific tasks, like analysis, testing, design, documentation, development etc., but rather to make sure that every member of the same

team can carry out all of these tasks without depending on external teams or members. Can we always have a team like this? What happens if we don't have one, or if we lack a key component?

Scrum's advantages

I don't want to become one of those Agile evangelists, preaching everywhere about how good and modern it is to be agile. But now that we've discussed the disadvantages of Scrum, we should look at Scrum's advantages. I'm going to focus on one of its brightest features: regular deliveries. Thanks to these deliveries, Scrum will make the following possible:

Clients can start using their products

The client can start using the product even if not all the elements have been built. If 20% of the new features have been developed—which are the features that will be used 80% of the time according to the Pareto principle—users could start enjoying the product.

With the feedback of users after trying out one of the deliveries, we might realize that some of those features are far more important than 80% of the remaining tasks in the Product Backlog. Somebody could say "*but the draught of the new regulation no longer requires a signed approval form*" or "*actually what we need is a button that would cancel the procedure*" (we actually received these two comments in real life).

We can decide where we're going

Businesses change, needs change, regulations change. And what was key when the project was signed might not be so important six months later. The client can pursue new goals, what to do in each new sprint and what we should focus on in our next delivery.

Divide and conquer

Colossal tasks require colossal efforts. If we must deliver just a part of that task every two weeks, our load will be lighter. Smaller, more manageable tasks make us feel that our job is easier. In each delivery, we have the feeling of having advanced one more step towards the final goal.

Fewer surprises

When we see our product grow, little by little, we all get an idea of what we are doing with it and if it is going to be useful or not. Also, we will know pretty accurately at what speed things are being delivered and how long it will take to finish up. Should we correct our course? We would know that in weeks.

Deliver what the client needs

One of the basic principles of Scrum (and also one of its main challenges) is accepting that clients can change their minds about what is or isn't necessary. With Scrum, we strive towards providing a flexible response, admitting that clients themselves may not have put their finger exactly on the problem, and that often, as the project advances, they may come to realize what they truly need. That is why, among other things, Scrum is considered an Agile methodology.

The list of requirements and features created at the beginning is open and can be modified at any time. It contains rough estimations of the amount of effort that each feature requires. Before each iteration, or sprint, a group of these requirements is set as the goal for that period. Two or three weeks later,

requirements may have shifted and the new goal might be in another direction, rather than the one that was set a few sprints back.

It is a new way of working, and a bit hard to adopt, for both the client and the provider. There are other, apparently easier, ways. We can always go back to the traditional formula. First we analyse the problem for months. When we've determined what should be built, we start developing it for a few months more. When we're finished, we cross our fingers and deliver our final product to the client.

After this step, who hasn't heard sentences like: "*but this is not what I wanted, there's this thing missing here…,*" or "*no, it wasn't like this, you misunderstood me*" or "*yeah, it's fine, but I'm going to call the Director, actually he's the one who has to sign it off.*" After months of work, of stress and rushing around to deliver by the deadlines, clients have not received what they need and we need to work even more to try and patch up the proposed solution.

How Scrum helps to build big projects

Revisiting Raúl Hernández González's post "*Learning how to build cathedrals"* I started reflecting on that well-known story about building cathedrals. In the story, two workers are chipping stone to build a cathedral and somebody asks them what are they doing. One complains about how hard and never-ending it feels to build the wall he is working on while the other one simply replies "*I'm building a cathedral.*"

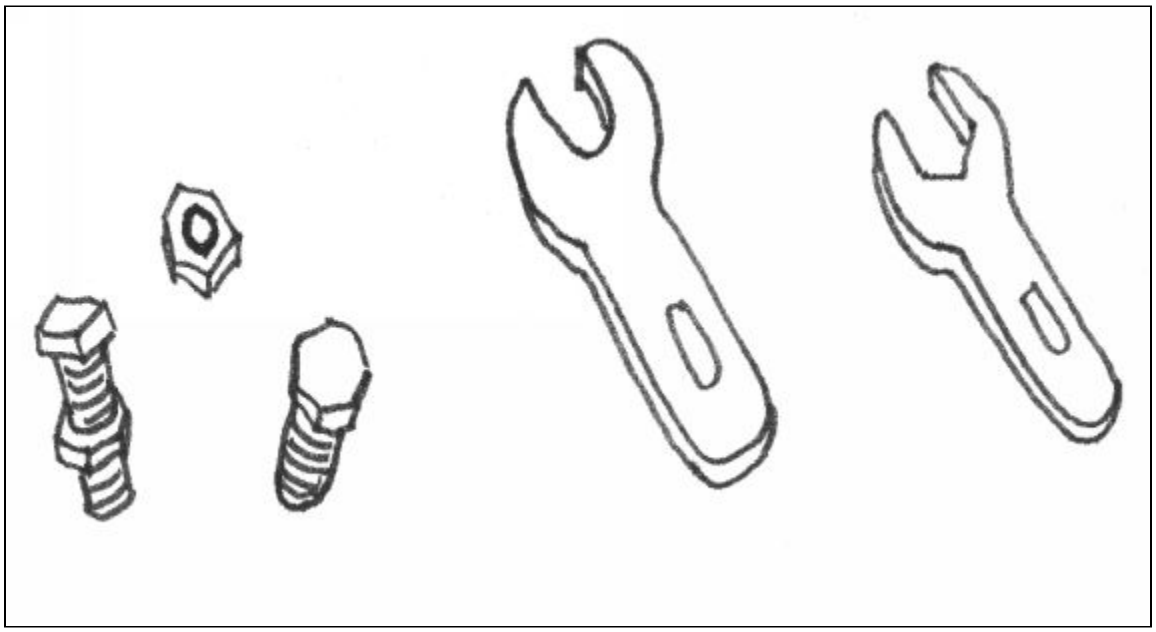
I have wondered if it is possible to be agile when you are building something so big. I think that the answer is yes, you can. At least I can think of a few advantages of being agile when you are immersed in a project that is the size of a cathedral:

With a framework such a Scrum, you'll always bear in mind the final goal and the features it should have. In the Sprint Backlog you will have a series of requirements such as the creating a ground plan, raising the vault, decorating doors and windows, and many other things that will give shape to the final cathedral.

But once the goal has been defined, we have to start chipping stone. At the beginning of each iteration, the team will choose which tasks can be started from the list of things that are pending construction (and that has been prioritized by our client). We will define a smaller goal for the next few weeks, so we'll have a new goal, a much closer goal that will help us not to despair at the distance that still separates us from the final goal. It is the moment to pick up our hammer and our chisel.

When our pending tasks are divided into small groups to be solved in fixed amounts of time, they are *timeboxed*. This is a *divide and conquer* strategy. Plan your most immediate goal and what you will do to solve it without getting overwhelmed with all the work still pending. A Burndown Chart will also help you see how the amount of pending work is diminishing and that, however slow it may seem, you are making progress towards your final goal: the cathedral.

SCRUM FOR ALL KIND OF PRODUCTS



"All ideas, even sacred ones, must adapt to altered realities."
Salman Rushdie

Scrum in a maintenance project

There is a common doubt among those considering using Agile methodologies to manage a project: would it be possible to apply Scrum to projects where, on top of the upcoming and planned tasks, there are frequent interruptions to solve maintenance problems, to correct errors or to resolve issues? Many project managers face these types of problems and use several approaches to tackle them. Let's see some of those approaches:

Short sprints

With this solution, we will keep the tasks that have already been programmed for that sprint. If our sprints are 1 or 2 weeks long, we will be able to make the unplanned tasks a priority on the to-do list for the next sprint. If the sprint is one week long, we will start urgent tasks after 3 days, on average.

In an ideal world, this would work but it would pose some disadvantages. Not every issue can wait for a few days to be solved. A whole service could depend on it.

Low load factor

If we know that as a general rule we will have unplanned tasks or issues that we must solve quickly, we can lower our load factor during the sprint so we have "*room*" to solve these problems.

If in each sprint the team has the capacity to solve 10 planned stories, we could promise to deliver just 7, so the team has time to solve urgent issues. That way we won't fail when it's time to deliver what we promised, sprint after sprint.

In my opinion this solution can be useful in some projects but it could create another problem. Let me explain. Normally the load factor is 75%. If we lower it so we're able to devote 30% of the team's time to urgent tasks, we should apply a load factor ranging from 40 to 50%. On this 40% we're using Scrum but, how are we managing the rest of the team's time? What do we know about that pile of tasks that we're solving sprint after sprint?

A different team for each type of task

This solution consists of having a Scrum team for the identified and planned tasks, and a Kanban team for the urgent issues. With Kanban we can add new tasks to the TO-DO column as they arrive, and we will follow them up until they get shifted to the DONE column. With this the team will be even more agile, reducing the overhead in meetings and planning that we could suffer with Scrum.

We still have a dilemma with unplanned tasks, which are, well, unplanned. There will be times when the Kanban team providing support will be oversized, because there are very few issues. But then a week later the number of issues and their importance could be so high that we will need any help we can get.

To minimise those risks, we could rely on the previous solutions. We could have short sprints, so the Scrum development team can plan the tasks for the next iteration. We could also use a lower load factor so the team can have a bit of room in their planning to give a hand if needed. In the same way, the support

team can help with planned sprint tasks when their TO-DO column starts to look empty. To make the most of this, it would be good if all members in all teams rotated, so that everyone knows every aspect of the job.

Scrum for fixed-price projects

One of the most common questions we ask when we're wondering whether we should adopt a methodology like Scrum is how to face the fixed-price contracts that are so common in public procurement and many other sectors. One of the Agile principles says: "*we welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."* This means that using Scrum we will be flexible with features we must develop, and we will admit new ones that were not in the original contract listing. This sounds very good but… the final price of the project is not going to change!

I work in the public-procurement department in a software-development company. That means that in at least 90% of the projects I work on I have a public institution as a client. Contracts are normally defined by a document with specific administrative specifications and another one with technical features. These contain all the services that are being hired, the characteristics of the product to be developed and the maximum price of the tender. They include even the technical features of the software and the profile of the projects' participants. It is all very narrowly defined. It is the price that the public administration must pay to avoid arbitrariness.

If our company has been lucky enough to win the tender, then both of us, provider and client, have to face the harsh day-to-day reality of projects. Circumstances change. New legislation comes up that affects the project. Things that were deemed correct a few months ago are no longer valid, because needs have changed. What should we do? If we adapt the project to every change that the client needs, then we providers have to bear the brunt of additional cost. If we refuse to change anything for the client, always waving the contract and its specifications in their face, we risk delivering something that might be perfect for last year's decree, but—for that very reason (or for any other reason)—is no longer of any use.

The way to avoid this, or at least the way that has worked best for me so far, arises from the way of working in Scrum projects. At the beginning of the project I make a list with the features to be developed, and give them a score that the client knows right from the beginning.

I normally give the example of a large jar full of ping-pong balls. Each ball is a feature that must be developed. The jar is full. If the Product Owner wants to include two new features, I ask that person to tell me which other features—or ping-pong balls of a similar size—should we take out of the jar. Normally this vision solves the predicament and the Product Owner does not resist giving up other features, now recognizing that they aren't important. If it all gets recorded in the meeting's minutes and, most important of all, if the Product Owner understands what is being given up and what will be gained instead, it is normally a comfortable solution for both sides.

Scrum's sprint 0 and the initial design

Sometimes we're in projects in which we need to prepare a large number of things before starting out. Can we start building without stopping to plan or analyse what we're about to do?

When we read about Scrum, we get the impression that all texts start directly with development, but…what about environment, team or office preparation tasks? What about design? Shouldn't we stop to think about the whole design and architecture of what we're about to build before we start programming?

About preparation or "*before*" programming. Some people call it "*sprint #0*", the sprint where you prepare your tools, install the stuff you need, train the team, etc.. It can be a longer sprint (3, 4 weeks or so). Some detractors criticise this sprint because it is not agile, and because it hides some work that is taken care of in the same way as the rest.

It is true that it can be very comfortable, because we tell the client that we'll be back in 3 weeks to have the preparation meeting for the first sprint, and by then we have organized it all. I have also used those zero sprints in my first projects, but now I prefer to solve those issues also in an Agile manner.

I try to make every one of those tasks (installing the development server, installing the demo server, creating a project and tasks in Trac or Redmine, installing the Integrated Development Environment or IDE, etc.) part of the first sprint and, if possible, have a set deliverable at the end of each demo.

In that demo, we could show, for example, the project-management tool already installed in the final URL and the task list with the tasks ready to be assigned, or we could launch the IDE in that demo meeting, and check whether the environment has been created correctly and whether Jetty executes a "*Hello World*" application correctly.

Another important step is the initial architecture design. If we make a huge design with all the application architecture (known as a *big up-front design*) before we start programming, we will be returning to a waterfall design. Firstly, we analyse everything, three months later we do all the design work during a couple of months, and then we program everything in one go until we finish it six months later. If we finish all work and show it to the clients and they say: "*that is not what we discussed a year ago…*" then we have a problem.

It is better to design one of the modules of the application, or one of the new features, then draft the specifications, and finally send them to development. While part of the team implements them, analysts can gather requirements for another module or another feature, and go on doing so sprint after sprint.

There will be sprints in which we won't have new stuff to analyse and others in which the development goes a bit faster than the specifications, but normally there is always some task ready to be carried out while Product Owners sign off the latest requirements, or until those requirements are completely defined.

If we work like that, new modules or features could be regularly deployed to production or preproduction, so users will be able to take advantage of them without waiting until the very last feature of the application has been designed.

How to write an Agile book

Yes, you can also write a book in an Agile manner. Not only software can be created this way. All these Agile things can be applied to a myriad of fields, not only to technology. In this chapter I will tell you how I used this work philosophy to write the book that you have before you:

Focus on what's important, delete the non-essentials

Nowadays books—especially technical books—are going through the same phenomenon as CDs did a few years ago. You bought a CD only because you had heard that song on the radio, but, how could a CD have less than 20 songs (and cost less than 20 dollars)?

It had to be filled up with something. Maybe the last few songs did not have the same quality as the two or three first hits in the disc? Well, it was necessary to justify the price, to pad it a little. This could make you hate the author for those last three songs in a duet with Tom Jones or the techno-pop version of their first hit.

The same can happen with books. We feel better if our book is 500 pages long, and we write it for two years but, at what cost to you, the writer? And for the reader who buys it? Probably that reader is only interested in the chapters on Agile quotes or about Scrum's advantages.

Something of the sort happened to me while I wrote this book. It had very few pages and I was tempted to add some filler chapters. They weren't really as interesting as the rest and they weren't as close to the main subject of the book. I ended up taking them out. The first version of the book was about 67 pages long, but I chose to keep it that way instead of having readers think that the book was too long or too boring.

Real artists ship!

This expression, attributed to Steve Jobs (Real artists ship!) refers to the fact that real artists are not constantly re-thinking their work until they have the perfect painting or the perfect sculpture. They ship their works, put them up for sale, and then see what the market is really interested in.

Have you written a tutorial on how to install Pentaho? A Wordpress configuration manual? What are they doing in a drawer? Get a cover (there are hundreds of websites for that), format it, write an attractive description, and put it up for sale at Amazon's KDP, at iTunes or wherever you want.

It is only 30 pages long? Well, then sell it for one or two dollars. If you sell a few you'll get a return for the 20 or so hours that it took you. Also, you'll have learned lots of things about digital marketing, sales, royalties…and also about which kinds of subjects sell books and which don't.

Perfection is a vertical asymptote

No matter how much you try to write the perfect book, with the perfect cover, with the exact price to maximise sales and no typos…you won't be able to do it. Perfection is a vertical asymptote (sorry again, Math 101 Calculus).

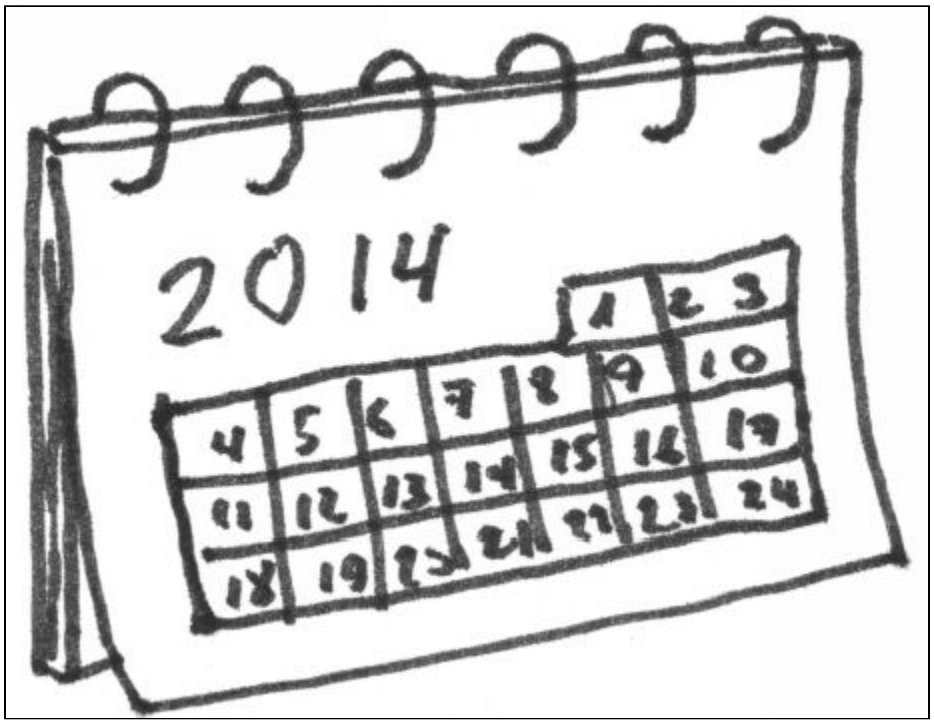
The better you try to make it, the higher the cost will be for you. When you've put an enormous amount of hours into it, putting in another enormous amount won't get you any closer to getting a bestseller.

When I put the book up for sale, I realized from the first opinions on the book that users thought it would be a Scrum manual. I had to learn from that and change the description to make it clearer. Small changes in the description had very high impact on sales.

In the same way, when I updated the book to its second cover, sales increased by 30%. Sadly it went further down when I changed it to the third cover. It was more expensive, and in my opinion prettier, but apparently Amazon's users didn't like it.

Your leads will tell you if they find the content, the cover or the subject interesting or not. You can think it over again and again, but until the book is not in the virtual shelves you won't know what works and what doesn't. Avoid the paralysis that sets in with perfectionism and don't overthink it. Put your book up for sale and let them decide (remember, real artists ship!).

ESTIMATES WITH SCRUM



"A project can only be estimated with precision once it's over."

Seen on Twitter

Estimates with Scrum

One of the things that gives us more headaches when we decide to adopt Scrum (and also when we don't) is how to make estimates, and how they will help us know how we're doing or when we'll finish the project. In Scrum you normally estimate twice (yes, twice) but not in the traditional way, which consists of one single and well-thought-out estimate that took hours or days at the beginning of the project.

This way to make estimates is somewhat controversial, but I'll try to explain it in the best possible way. We carry out a rough estimation in points, not in hours, based on the initial feature list. We estimate the effort —or difficulty— that we believe each of the project's requirements would take. We measure it (or, more accurately, we calibrate it) following for example the Fibonacci sequence (1, 2, 3, 5, 8, 13, etc.) or even using the sizing chart that you normally see on t-shirts (S, M, L, XL, XXL, etc.).

By the Cone of Uncertainty Principle, no matter what estimation we have done, no matter how much time we've devoted to it or the methodology we've used, our estimate will without a doubt be wrong. If it turns out to be right, it will probably be just a coincidence (unless we know each and every detail of the project). Rodrigo Corral explains it very well in his blog post *Intuition-led project management, or why project management is so difficult* (in Spanish "*Gestión de proyectos guiada por la intuición, o por qué gestionar proyectos es tan difícil*")—the rest of the post is really worth it, too. Then, why should we devote so much effort to it? We'll just give 1 or 2 points to simple features, 3 or 5 to the medium-difficult ones and 8 to 13 to the difficult or very difficult tasks, and that's it. (Do you need more points? Then maybe you should break down that feature into smaller ones).

This will save you enormous amounts of effort when it comes to estimations. And we may as well say it: this will also reduce the team's stress (they are normally afraid of being wrong with estimates). With a high-level estimation with points, we avoid some of the drawbacks of hour-based estimates. For example, if we indicate that a certain task will take 40 hours, the rule that work will expand until it takes up all available time will apply to this, too. On the other hand, we will avoid the common mistake made by Scrum newbies (yes, I made that mistake too) which consists of thinking that "*if 40 hours have been spent on this task, the burndown chart must descend 40 hours.*" This would give the false impression of going forward in the project if the task is not completely finished and approved by the client, and we actually have 20 more hours to go until that item is successfully completed.

The second estimate will be made when we plan the *sprint* tasks. In this planning, we will know the project much better and what each task implies. We will be able to do an hour-based estimate then, even though some teams still decide to estimate on points at that juncture, or decide not to estimate at all.

These types of estimates will allow us to calculate the ROI (return on investment) of each feature, so clients can decide which kind of task they want taken care of in the next sprint. For example: a Product Owner knows that feature A will take 2 work weeks to finish and places the value or importance of the task at 100. But maybe seeing that, the Owner decides to prioritise features B, C and D, valued at 50 points each, and that can also be done in those 2 weeks. That way the client could get more value faster: 150 points of value for the final product, instead of 100.

More on estimates

When we talk about estimating using Agile techniques, some things are difficult to accept for those of us who have been planning and calculating timing and deadlines for a few years.

We can find several techniques for Agile planning: *Planning Poker*, clothing sizes for task classification (S, M, L, XL, etc.) or *Team Estimation Game*, but, let's face it, they're difficult to integrate in a traditional development environment, and even harder to integrate for the clients who purchase our products (or at least, for the clients' I've been able to work with).

We're not used to seeing a team "*playing cards*" for a session of *Planning Poker* (at least, I am not used to it). I do, however, agree with the philosophy behind those techniques. I believe that they're becoming more and more popular partly because it is necessary to dispel some myths regarding estimations. Let's try and do just that:

We do estimates wrong

Can we do them right? To estimate is to predict how long a task will take or what materials we will need for a certain job. When we do an estimate with high uncertainty, as for example when we use the data included in a public-tender document, it's not so much an estimation as a bet.

We need estimations—that's obvious. We must try to predict what will happen, so we can make decisions. But unless we've gone through that specific task many times before, with the same team and under the same circumstances, we need to be aware that our estimates will very likely be wrong.

The more effort we put into an estimation, the closer we'll be to reality

Estimations have a cost and that cost is high. When we don't know every specific detail of every bit of the work we'll be taking on, does it make sense to spend hours and hours to guess about tasks that we do not know very well? When you finish your next project, do this exercise for me. Review the time spent on every task. You'll see that you estimated 20 hours on tasks that took you 100, and you'll find that —with luck— you spent 20 hours on tasks that you thought would take you 100. You'll see that there were some tasks that no-one marked as done. They simply weren't necessary. However, hours are spent on tasks that were added later and that no-one considered necessary in the initial estimate.

We'll estimate better with this or that technique

There are many estimation techniques but very few will be as useful as comparing the time spent in similar projects, the decomposition of the work in smaller tasks, or the conclusions of experts (it would be better if we have several of these).

The fact that we're using complex estimation techniques could give us a false sense of security about our estimate, and prevent us from striving for higher reliability. That could actually cause us higher costs (see above) where we invest time that would have been better spent doing something else (like reducing uncertainty).

How to make estimates come out the way you want (don't fool yourself)

Now, let's hear a quick story about how we fool ourselves when somebody requests a quote for a job. It may have happened to you at some point.

A client has requested a quote. He has an idea in mind and wants to know how much it will cost and how long will it take. He made some calculations in his head and comments briefly on the numbers he is considering.

You arrive to your office and meet with the experts in these types of projects. After mulling it over for quite a bit you agree on how long it will take and the level of difficulty. You've had to do a little convincing, so they're not so overly cautious, because after all it doesn't look like a difficult project.

However, the final quote reached is way higher than the client had foreseen. Furthermore, if that vision were to be true, the project would not be finished by the deadline . You'll need to reduce that estimate by at least 50%.

Here's how to do that in 5 easy steps:

You've done this before. You'll be faster this time. You've carried out similar projects on other occasions, so you've surely learned from your mistakes. Also the problems that came up other times won't necessarily happen this time (10% less). Are you sure you won't have some *new* problems?

The clients said they wanted something simple. It won't be that hard to do. We'll do something basic —we'll cut on the complex stuff (10% less). The clients know their business well and it's easy for them, but, what about you? Do you know everything you need to know about their business?

We'll really keep an eye on costs. Some other projects weren't monitored very closely, but this time you'll be especially watchful of every hour spent, and you won't let costs go through the roof (10% less).

We'll put our best technicians on it. It's a major project for an important client. We'll put our best people on it (10% less, even though you don't know whether they'll be available on the date you need them).

We'll make an extra effort. If push came to shove, and we saw we were about to miss the deadline, we would ask every stakeholder to make an additional effort for a few weeks. If it's necessary, we'll even add some extra technicians halfway through so we can finish on time (10% less). Remember Brooks' Law: *adding extra staff to a project that's late will only delay it even more*.

And there we are. With just a little bit more work, we've been able to cut the initial estimate by 50%. Actually, to make the estimate match the budget, we've convinced ourselves that we can do the work in half the time. Sad to say, stats show that software projects take an average of 120% more time than initially estimated, and that final costs are normally 100% more than was budgeted at first.

The anchoring trap

When we make a decision, like when we estimate, we give a an extraordinary amount of importance to the very first information we receive. A comment from a client or something we've heard about a similar

project can make us anchor our proposal to that first piece of information, even though some other more objective detail would have led us to a different number.

George Siedel, a professor in the University of Michigan, has a test he normally runs on his students to show the effect of anchoring in their predictions. He asks every student to do the following:

□ Write down in a piece of paper the last three digits of your phone number. Add 400 to that number and write it down in the same piece of paper.

□ Attila of the Huns was one of the most terrible conquerors of our era, until he was defeated by the Romans. Was he defeated before or after of the date you noted down in your piece of paper? Write in your note "Before" or "After".

□ Now write down the year in which you think Attila of the Huns was defeated.

Funny enough, those whose number plus 400 was between 400 and 599, predicted that Attila was defeated on 580, on average. But those whose result was between 1200 and 1399 estimated, on average, that it had happened around 1340. They had connected the two things, but what does the year of Attila's defeat have to do with our phone number? Nothing at all. But we all naturally tend to be influenced by the first information we receive.

This is what happens when clients request a quote but say they can only spend €20,000. We do our calculations for the cost of the project influenced by this number, in pretty much the same way as the example with Attila and our phone number above. If the real data when the project finishes shows a much higher number, did we execute it wrong, or did we estimate it wrong because we were anchored to those €20,000?

Do not place much trust on your experience in similar projects. Experts also fail on their estimations. Two researchers asked specialised pulmonary surgeons for the probabilities that a certain patient would develop a pulmonary condition.

They asked the first group if the chance was higher or lower than a random number from a card—let's say 20%—and then asked them to add their own estimation. The second group was asked the same question, but with a different random card —let's say 50%.

You can imagine the results. The calculations of most members of the first group were close to the first number shown on the card. And in the second group, they were close to that 50% that appeared on the second card. The random number that was shown to them influenced their prediction much more than all their previous knowledge on the subject.

Be very careful with any information that may anchor your estimations. It could become very costly for you.

Overconfidence

Who hasn't ever been wrong when estimating the cost of a project? We carefully divide and subdivide work, with a detailed list of small tasks. We consult our senior colleagues about the most complex ones, and we consider the time to prepare the environment, and many, many more things. But still, we get it wrong, and quite frequently, we get it wrong by a large margin. What's happening? Don't we know how

to estimate correctly? We can't lose money on every project, but we can't inflate our quotes "*just in case,*" either, because we'll start to lose clients.

There are many reasons why we get it wrong time and again when we estimate. But maybe overconfidence is one of the main ones.

Human beings are optimistic by nature. We trust our own abilities too much, and we tend to think that we are immune to small daily disasters. We're not. When we calculate mentally how long will it take us to finish a certain task, we normally just count the time needed to write the code. But we often forget about the time we need to test, deploy, document, and discuss the code, to solve that bug in a certain browser, to write the user manual, and endless other things.

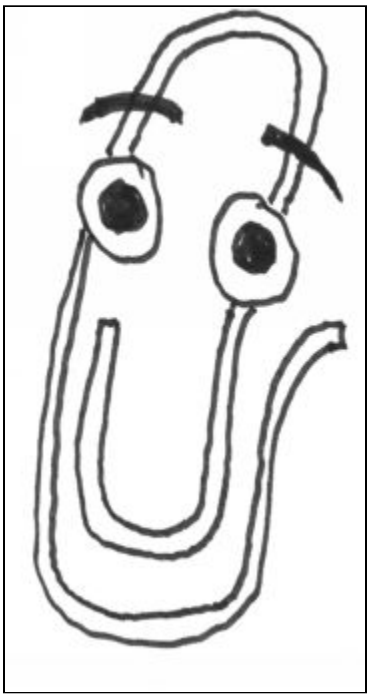
I've got bad news. You're too optimistic, too. You don't think so? Humour me with this test. If I ask you what year Mozart was born, you won't be able to give me an exact number. But if I make it a little easier and ask you for a range of years in which he was born with a 90% probability, you might just dare to venture an answer.

If we made this test a bit longer and we added another 9 questions in which you could give an answer with 90% probability, with questions like the length of the Nile or the diameter of the Moon, what do you think would happen?

If your estimations were correct, you should have gotten 9 out of 10 questions right… but they probably weren't. Most people get two or three questions right. Don't worry, we do tend to be overly optimistic, and we choose a range that's too narrow to get right (we should have chosen a probability range of being right about 30% of the time). It happens to all of us. Actually, the people who lack this optimistic bias are considered clinically depressed.

If you're curious, Mozart was born in 1756, the river Nile is 6,738 km long and the diameter of the Moon is 3,476 km. This overconfidence in our estimations or predictions is normally studied in negotiations and sales processes of every kind, from real estate to start-ups. George Siedel, from the University of Michigan, explains it in his Coursera online course. Have a look, I highly recommend it. Nearly everything in life is a negotiation (project management, too).

THE ROLE OF PRODUCT OWNER



"Are you sure you need a Clippy-like assistant on your app?"

Many titles could be used for the person who acts as Project Director on the client's side: the person with the product vision that the company needs, the one that represents the company in the work team. If that person belongs to the external organisation that has ordered the work, the title which fits best might be *Product Owner*, like in Scrum. That's because that person literally owns it, is the "*Owner*" of what will be built. If this person belongs to our own organisation—someone who will represent the "*Client*" in the project—the title normally used is *Project Manager* or even *Business Analyst*, depending on the company doing the hiring and naming. Personally, maybe because of the type of projects that I normally manage, I'm more comfortable using the title *Project Director*.

Whatever we decide to call these people, their contribution to a project is key to its success. They decide which features the project will have, which ones are indispensable and which ones are not. Maybe they will be the users of the final project, and thus will have a very clear idea of which elements would be key to create a useful tool that they use daily at work. However, its usefulness shouldn't be limited to just themselves or their department, but rather it should also take into account the requirements of the whole company.

When we start a new project, the Project Director on the client's side and the users of the final product have lots of ideas and great expectations about the new system. Sadly, no project, no matter how big it is, accommodate each and every suggestion from its users. Some ideas would be irrelevant for most users, other features might be too expensive to implement, or they would take so long to build that the project would be delayed for too long before being delivered. It's here where the Project Director on the client's side, who knows the market well and has a clear vision of the product, will have to prioritise the most important features and discard those that provide less value to the final product.

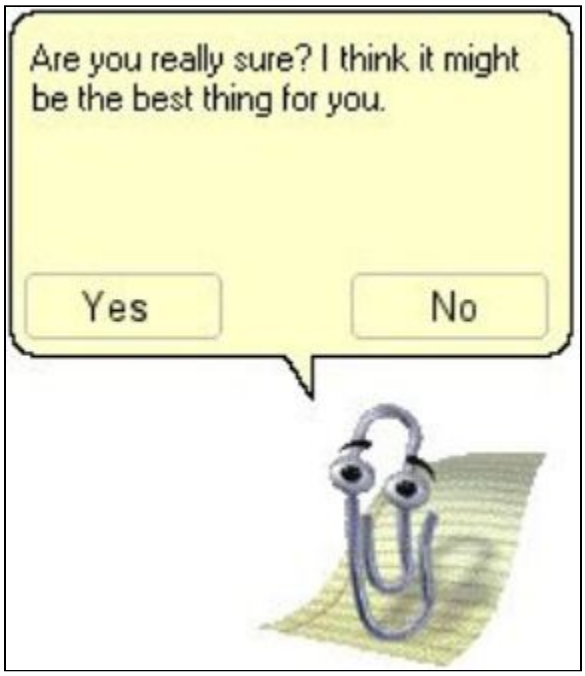
To explain how far-reaching the responsibilities of Project Directors are, let's imagine the following. Our company decides to contract the development of a new airline ticketing system. In the company, they have decided that Elena will be the Product Owner or Project Director, and she will have to transmit to the chosen company all the requirements and needs that the new system must deal with.

Elena has a very clear vision of what she wants and is extremely enthusiastic about the project. So are the IT, marketing, and sales department heads, who have prepared an exhaustive list of features that the new product should include. Even in informal conversations, every day the future users of the system make new feature requests. Elena doesn't want to miss anything important and conscientiously writes down all these requirements in a notebook.

Two months later, Elena realises that the team carrying out the work in the development company is delivering from 4 to 6 new features every week, and this seems to be their maximum capacity. She is happy with the work they are doing, but she has a problem: in her notebook she gets about 10 new feature requests every week. The list is growing and soon she will need a new notebook.

First of all, she asks the work team to double their efforts and deliver 10 features a week. Unfortunately, not many more features are delivered, and worse yet, she can tell that the quality of those features has gone down. On occasion, they even have to stop everything and correct previous deliveries. If things go on like this, frustration and stress will take over the project.

From now on, Elena must decide what should and shouldn't be done. Not every feature will be included: at least not now. Some will have to wait for future versions. And of course, the idea of including a *Clippy*-style assistant to the purchase process for flight tickets will get a clear "*NO*."



Source: original image

But, which features should she have developed first? Which criteria should she apply? Elena realises that there isn't a direct relationship between feature cost and value added to the final project. Some features are easy to build, but some others take a very long time to develop while not actually providing a lot of value to the final product. If she asks the work team directly how many days each of those features would take, she will know what was the approximate cost would be. If she asks the users to score importance of each feature from 1 to 5, she will be able to calculate the value for her company.

If there are two features that have the same approximate cost, but one has value 1 and another one has value 3, they will clearly add the second one. If there are two features with approximately the same value for the company, but the first one would be ready in 5 days while the second one would be ready in a few hours, it's also quite clear which one they should work on first.

Elena knows that the value and cost of each feature aren't absolute values, but just approximations. We don't know how much an apple weighs, but we know it's about 5 times bigger than a strawberry and that

people like the strawberry more (at least, the people who will pay for it like it more), so the strawberry would be built first. It's an easy way to decide what provides more value, faster, to our project.

Communication is one of Elena's main responsibilities. She must be in contact with the work team, but also with the people in her company who have something to say about the product that is being built. She has to be able to say "*no*" when necessary: she has to be practical in this sense, making this decision swiftly and directly. And also, she should know her business inside out, to make sure that everything that gets built will actually provide value to the company. I would say that Elena has quite a lot on her plate, don't you think?

KEEP IT SIMPLE, STUPID

"So, you are saying that my design is too complex?"

Some time ago, I had a conversation with an IT colleague, about the sector nowadays. Something reminded me of an Andrés Diplotti cartoon I'd seen on Google+. In it you could see a client reviewing the application that the programmer had delivered. The client says: "*It's all fine, but I don't see a Cancel button*". The programmer replies: "*I don't program for cowards who cancel at the last minute.*"

I've been a programmer for a long time. I'm still a programmer, in part. So I can recognize this kind of arrogance (maybe not to that extent). I suppose that it's a sin of my youth, a brashness that's been mellowing with age and that some of us already felt in college.

Back then I had just discovered software-design patterns in the book *The Gang of Four*. Our applications started to be filled with design patterns, the more the better. It looked like we were in some sort of contest. We had to see who used the most obscure, complex pattern —the one that would be the hardest to understand. With this in mind, I can understand the posts and articles I read now. They discuss refactoring old code to get rid of design patterns, and about how expensive it is for some projects.

As the years have gone by, I have suffered the KISS principle (*Keep it Simple, Stupid*) in my own flesh. Not just with patterns, but also with the customization of on-screen graphic components. Or with "*phantom requirements.*" This is a term I once heard from a fellow worker. It refers to those features that no one has requested, but that you consider important for your clients. Things that "*most probably*" they'll end up needing. It's the malady known as *featuritis.* It makes us think that the more features or characteristics our app has, the more valuable the client will find it.

A long time ago, a client's project manager commented that someone on another work team had told him that a feature he wanted was impossible. I told him that practically everything can be done, provided enough time is spent on it. I wasn't clear enough. I should have added: but is it reasonable to use all that time for that feature? You can use practically any programming language to send a rocket to the Moon. But, do you really want to go to the Moon, or do you want a finished app by the deadline?

The 80/20 rule

How can we avoid *featuritis* and concentrate just on the tasks that will make us more efficient? Many years ago, approximately a century ago, someone called Pareto realised that 20% of the pea pods in his garden amounted for 80% of his crop. The remaining 80% of his pea pods amounted to the other 20% of his pea crop. Intrigued by this ratio, he also checked that he could use this rule to other fields. He went on to discover that, back then, 20% of the population had 80% of the income. The remaining 20% of the wealth was in the hands of the poorest 80% of the population. This principle, based in empirical knowledge, has been in use since then. It helps improve efficiency and productivity in economics, commercial distribution, marketing, engineering, and, naturally, in software development.

For example, we start to develop our product with a huge list of requirements and features as a base. Yet we know that only 20% of those features will be used 80% of the time. A large chunk of the remaining features will only be used 20% of the time. We can apply this principle to development time. Using it, we can deduce that if it takes us about 10 months to build a product, in 2 of those months we will be able to develop 80% of the features. Thus it would take us about 8 months to solve the remaining 20% of the most complex and difficult features. This leads us to ponder: will people use these features? Are they really indispensable?

Say we manage to identify the most important features, and we keep them as simple as we can. Wouldn't we be able to dispense with 8 months of work and deliver a highly effective product in 2 months only? We already know that 20% of our effort will produce 80% of our results. Wouldn't it be worthwhile to sit for a moment and think for a bit about what we should devote our time to? I'm sure it would.

Broken windows theory

In the late 1960s, a university professor carried out a psychological experiment. He left two identical cars in two different neighbourhoods. One in a poor, high-crime neighbourhood in the Bronx, New York. The other one in a rich, quiet area of Palo Alto, California. Shortly after it was abandoned, the car in the Bronx started to have all its parts stolen. First the radio, then the tires, the mirrors, and anything of value. Yet the car abandoned in California remained undamaged. The study did not end there. When the car in the rich neighbourhood had been intact for a week, the researchers broke one of its windows. This escalated quickly, and soon afterwards the effect was the same as it had been in the Bronx. Theft and vandalism swiftly reduced the car to its bare bones in both neighbourhoods. A car with broken windows sends out a message of disrepair and negligence. It spreads the notion that *anything goes*. Every new act of vandalism reinforced and broadcast the idea that nobody was taking care of that car. That nobody cared.

A similar idea can be seen in the training of health professionals. If a patient or an elderly person has a stain, or gets dirty somehow, that person should be cleaned. Even if it's only a little spot. If patients stay in that state, they will be less careful themselves in keeping clean. They will tend to feel that it doesn't matter anymore, it's already dirty. So their general state of hygiene, self-esteem, and thus health level will start to decline.

The results of this psychological study are applicable in many situations of our daily lives: from the care of our home to the maintenance of our car. But they're also applicable to our jobs. For example, if you're a programmer or a project manager, and you allow for a new version to be deployed. But you haven't had it sufficiently tested. Or if you leave that technical "*debt*" unpaid in the code now —because "*we're in a hurry.*" You'd be broadcasting to the whole team the feeling that anything goes. That "*it's enough as it is*" and that one can leave quality aside. Sooner or later the phone will ring with complaints from users. Soon we will see a lengthy queue of reported bugs in our queue.

When we apply this theory to software, it's helpful to explain the term *entropy*. In physics, *entropy* is the level of *disorder* or randomness in a system. Say you see badly indented or commented code. Or a bad choice when naming a variable or some bad design. Fix it as soon as possible or schedule its correction for the near future. Take some sort of action to limit disorder or it will expand. It will create the feeling that any patch or quick fix is valid.

If some code (or thrown-together code) will make your system deteriorate quickly, impeccably written and designed software will have the opposite effect. It will make new programmers avoid breaking something so beautifully built. They will make an effort to put in place the best code they can write. Now you know: make your life and your project easier by fixing broken windows as soon as they appear.

Too small to fail

Kaizen, or continuous improvement, is a well-known term in business or industrial organization. It's now spreading widely in every sector of the economy, but it also applies in social life or medicine. This philosophy arose in Japan. Their industry had productivity and quality problems after the end of World War II. They looked for solutions, and brought experts such as Deming. Yes, the same Deming of ITIL

and the "*plan-do-check-act*" Deming cycle. His mission: training hundreds of professionals and engineers in quality and improvement systems. These methods had already been applied in the USA. But Japan developed them and improved them. They refined them to such an extent that years later they could return them to the Americans themselves as new work philosophies.

The principle behind Kaizen is to carry out small continuous improvements. The results of those changes are analysed and the fine-tuning resumes. In that way the productivity or quality of the task being performed can be improved. Making small changes little by little is far more effective than trying to tackle everything in one go. This way you avoid the fear of big change—and the procrastination that normally goes with it —when faced with the idea of starting a transformation. These small adjustments, done continuously, end up becoming a habit and generating permanent results.

The idea is to make changes so easy that it would be hard to fail in their implementation. First we created the habit of changing. Then we add new changes or nudge the milestone of a change a bit further, so we can improve incrementally. It's important to apply adjustments one by one. The idea is to avoid the complexity of choosing what to apply and when. That way we will be able to analyse the result of each of these minor improvements. If we apply several at the same time, we won't know which one has worked and which hasn't. Or whether the effect of one has cancelled out another.

In your project you can decide to deploy every SCRUM, TDD, unit testing, continuous integration, and so on and so forth all at once. But probably the only thing you'll manage to do will be to drive your team crazy. Yes, all those practices are good for your project, but, are the most basic ones working? Do you have good version control? Are you delivering software every two weeks? It's better to start by the simplest ones or the ones that you consider most effective to improve your situation. Then you will be able to add the others.

Even far away from the field of software, there are companies that use Kaizen and Lean techniques in production. Take, for example, SPB, manufacturers of household brands like *Bosque Verde* in Spain. Their industrial manager declares:

"SPB has improved its productivity by 15% without investing in new machinery or carrying out staff cuts. The systems we have applied have enabled our company to reduce our expense accounts from 15 to 25%. We have improved our raw-materials management by 45%. And we have reduced the stock period of our product to eleven days."

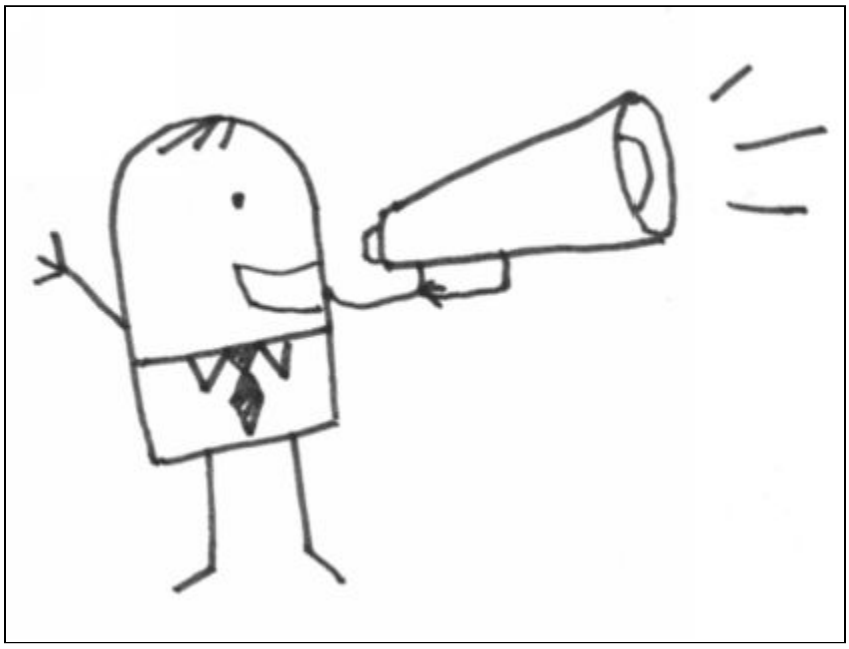
SPB had foreseen that to improve, they would need great investments and opening new factories in northern Spain. But instead they have focused on improving their productivity plan.

We can apply such changes to improve our health or personal life too. If we have always wanted to write a book or a blog, we can set out to write 1,000 words a day. But most of us will probably abandon that idea in only a few days. However, if we set out to write just 50, the change will be so easy that it will be hard to fail. First, establish the habit of sitting and writing for 15 minutes a day. Then we can take the challenge further and make it bigger, little by little.

In my case, I have also applied similar techniques, but I've hardly noticed. When I published my book it hardly sold at all. For weeks and weeks it had just the few sales I considered F&F (Family & Friends). But I didn't just park it there and forget about it. I changed it to a different Amazon category. Then I changed the cover. After that, I included some new chapters. Later, I improved the description until I found one that seemed optimal. Some of these changes made my sales increase up to 50% in a week. If I had just left it there, the book would never have reached decent positions in its categories. I would have concluded

that the content wasn’t interesting. Or that it just could not sell well without a professional publishing house to support it.

STORIES TO ILLUSTRATE AGILE



"Who wants to work till late again? Please raise your hand!"

Small start-ups

Ana had been laid off from a software-development company. After some time thinking about it, she managed to convince David and Alberto to join her in a project they had been discussing since college: founding their own company to develop software.

They were lucky that their first client soon appeared. It was a dairy-distribution company that had decided to give them a chance. The Head of IT in their area knew them from their previous jobs, had good references about them, and thought that it would be a wise idea to increase their vendor database by including a young local company. He would give them a small, non-vital project. If it worked well, many more would follow.

Ana, David, and Alberto did not consider it a small project at all. They had no experience managing projects of that size on their own, and were afraid they would "*choke*" on it. They had offered a very low price to build a stock-control tool, so they had to be very competitive if they were to make money with the budget they had.

There was also another added risk. The international consulting company that normally developed software for the client would be a hard opponent to beat. If the project went wrong or was late, the stock-control tool would end up being integrated as one more module in the ERP that the consulting company had already deployed for their client.

Ana and her partners decided to follow an Agile methodology like Scrum to try and reach the level of efficiency that would help them succeed in their first project. They knew that if they spent too long analysing possible features and then spent months and months implementing them, the project could be cancelled as the client would not be able to see short-term results. Their main target was to get something into production as soon as possible: something useful for the client, even if it didn't have all the requested features.

With this idea in mind, they started to develop their software. Every 2 weeks they showed the head of IT and the warehouse managers what has been accomplished during the previous fortnight. It looked good, but the company hadn't started to use it yet, so they didn't know if it would work or not.

Soon Ana suggested deploying a pilot version to production, one with a pilot version that would allow them only to add new stock, to record stock entries and withdrawals, and to draft a simple stock report. There wouldn't be any complex functionality that would foresee demand or any SMS alerts when the stock for a given product was low.

When warehouse workers started using the new application, doubts arose, but also it became clearer what was useful and what wasn't. After so many meetings, no one had thought that it would be useful to add a history of the product's delivery so they would know how far in advance a product should be ordered before they ran out of stock. Or that the order screen should include current orders of the same product, to avoid duplications in ordering.

The next items on which to work were quite clear now. Also, they constantly received new application-improvement requests. They had seen that if they integrated the warehouse-management tool with the invoicing software of the delivery vehicles, they would avoid typing every product twice, once for each

system, which would save them lots of time and errors. This was not foreseen from the beginning, but they reached an agreement. They wouldn't develop the demand-prediction tool, which would be postponed until a later version was developed, but instead they would add this integration with the delivery agents' mobile devices.

Cooperating with the client, accepting changes in requirements, and frequent deliveries of working software made the project a success for the dairy distributor. Now, at this local branch, they had a piece of software that did its job and also lightened the load on the warehouse workers and the delivery salespeople. They heard about the project in the Madrid headquarters, and were considering implementing it in other locations. Who knows, maybe this would mean a new project for Ana, Alberto, and David's start-up.

The engineer and the bridge

In the 5th century BC, a certain Roman engineer was appointed to direct the construction of a bridge over the river Leza. The bridge was important for the area. People and goods would be able to use it to cross the river, saving time and avoiding travelling long distances looking for a safe place to ford the river.

The engineer received this appointment with great enthusiasm, and went on to plan an estimate for everything that would be needed for the construction. He calculated how many builders, cranes, pulleys, scaffolds and centrings would be necessary. He could clearly see that with 50 builders the bridge could be finished in 4.7 years.

With the 600,000 sestertii that he had received to execute this assignment, he quickly bought all the necessary materials according to his calculations, and had them delivered to the construction site. He recruited the 50 builders he had estimated he would need and sent them to the river bank to start work at soon as possible. There was no time to lose. The sooner it was begun, the sooner it would be finished.

Soon, however, the workers told him that they didn't need so many pulleys and cranes, but that there weren't enough pickaxes and saws for the job. "*I can't do anything about that,*" said the engineer, "*most of the money has already been spent and there isn't much left to buy new materials. You'll have to adapt to what we've got.*"

The construction went on, and soon afterwards it became clear that things weren't going as planned. The foreman told the engineer that the stone quarry where the stone was being cut was too far away, and that the road was full of bumps and potholes from last year's flooding. It just took too long to bring the stone in carts to the bridge. "*You'll have to make an extra effort,*" said the engineer, "*there's no time now to fix the road.*"

The foreman also informed the engineer that it would be necessary to build arches in the bridge, to reduce the amount of materials used, and, most of all, to improve its resistance. "*There's no time for frills,*" said the engineer. "*We're late. We'll add a little extra mortar in the pillars, and that'll be it.*"

The news about the construction of the bridge came to the Prefect's ears. He despaired when he heard about how slowly the work was progressing, so he sent for the engineer. The Prefect needed to justify his actions in Rome, and demanded that the bridge be ready for Saturnalia. The engineer explained that, in order to achieve that, he would need at least another 50 workers. As he was also pressured from above, the Prefect agreed and committed to send the new workers a month later.

Back at the bridge, the engineer gathered all the builders and asked them for an additional effort to comply with that new deadline. The workers could not understand how were they to work twice as fast if they didn't have enough stone, and in any case they always had to wait for the mortar to set.

Not even with the new workers could the bridge be built on time. There weren't enough tools for everyone, the stone was still slow reaching the worksite, and collapses were common because of the hurried construction process.

In ancient Rome, a bridge's builders had to stand underneath it while an entire legion crossed it. It's a good incentive to build firm, solid bridges, don't you think?

The Ant and the Cicada

One summer a cicada was resting in a field, when she noticed a small ant running around here and there.

- "*What are you doing?*" asked the Cicada.
- "*The Queen has asked me to build a new anthill!*" replied the Ant.
- "*But they already have one there, just a few meters away,*" offered the Cicada, puzzled.
- "*Yes, but it's getting a bit small, and in winter, with the rain, the things we have collected get wet.*"

The Cicada, ready to learn from her hard-working friends, decided that it would be a good idea to prepare for the winter. The leaves and sticks she used to build her shelter the previous year were getting dry, and it wouldn't hurt to have a new home to spend the winter in comfort.

She quickly started to plan the wonderful shelter she would build. Now that she had time she would build it with every possible comfort. She would be the envy of the whole meadow. The stories about her new home would be heard far away, further even than the cypress hedge. It would have spacious windows so the light would flood in, and she would design a clever pulley system that would save her many trips to and from the barn to store what she had collected.

The Ant, however, had a plan. In August she would build the first room, which would be the barn. That way, if the rains started early this season, she and the other ants would have at least some extra space to store their grain. Then she planned the rest of the work according to its importance. If she didn't have enough time to finish it all by the deadline, she would have built at least the most important parts.

In September she would dig a hole that would later become the hall were the first larvae of the year would live. In October she would build an additional room to increase barn capacity. Lastly, in November she would finish the work by digging a hall where she could grow fungi.

In the meantime, the Cicada was worried about having a dining hall that was big enough, and a structure as solid as necessary to hold everything she had imagined. She would run back and forth looking for the materials she would need to build all the things that she had planned in her head.

In October the first rains came, but the Cicada didn't even have a basic shelter to take cover in. Also, water ruined some of her first constructions that were halfway built, and she had to start again. Then November came and the weather became really ugly. She had to hurry up and started working from sunrise to sunset.

Those first October rains made the Ant think that something could go wrong, and she decided to test whether the work that was already done would be useful. She started to fill her new barn with grain and

soon she noticed that the access hole was too small for big leaves, and that it should be a bit higher if they wanted to avoid water or mud coming through. In order to fix this, they had to stop doing other things.

Winter came early that year, and the Ant realized that she wouldn't have enough time to build the last room, so in the main anthill they decided that they should open an extra entrance, in case the main one got blocked by a stone or an enemy.

The Cicada, however, was surprised by the bad weather. She didn't really know what was finished and what wasn't. Also, what she had considered finished had not been tried out, and now there was no time to improve it. Winter was here and no more work could be done.

In her new shelter, under the leaking roof, the Cicada promised herself that this would never happen again: "*Next year I'll start work even before the Ant,*" she said to herself.

Real artists ship!

Sometimes, and quite often, we delay delivering our work because we believe it is not ready. We think that we still have things to improve or correct, that we still have to work on it. More often than not, that actually comes out of fear of failure. Fear of our product meeting the market and nobody buying it, because it is not perfect, or it is not good enough. It can also be fear of our product being criticized, because either the product or the book or our software has some typo or small bug. "*I will only publish the latest blog post when there are no possible ifs, ands, or buts.*" Other times it's just perfectionism leading to paralysis.

A pottery teacher in an art school divided his students in two groups and told them what they had to do to get an A+ in his class. He told the first group that they had to do 100 vases in order to get the highest mark. However, he told the second group that to get that A+ they had to make the best vase. This last group devoted enormous amounts of time and energy to debate and read about the best techniques for creating vases. But the best vases were actually done by the first group. They had devoted their time to practice, practice, practice and made 100 vases. Their technique had improved. It's the same with your blog, your app or that design you've been thinking about. Let them meet the real users and let those users give you a thumbs up.

The world of software is full of products that were developed over years until they had every feature a client could imagine, in the way they were initially designed by their creators. Sadly when the software met the market, the market had its own ideas and needs. Very few people used every feature in those applications. Other features were heavily demanded, but none of the creators had thought about them when those products were first designed.

There is a Spanish proverb: "*perfect*" is the enemy of "*good*." When you have a small but already useful product, ship it, show it to the public, get your first clients. They will tell you what's missing. Learn about that, launch a second version, then a third. If you can't get at least a small client base with your first version, maybe it is not the right time or place for your product. At least you'll learn lots of things you didn't know yet, and you won't have wasted months and months of your work.

IF YOU CAN MEASURE IT, YOU CAN IMPROVE IT

"Plans are only good intentions unless they immediately degenerate into hard work."

Peter Drucker

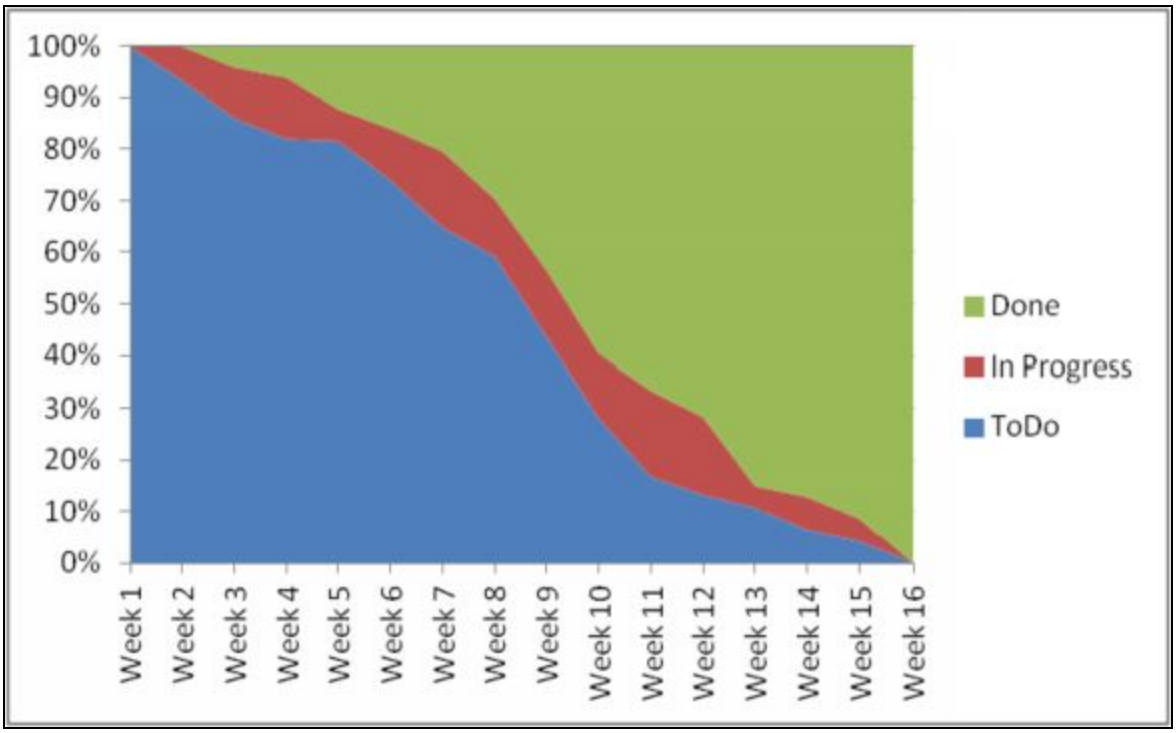
The fiasco of the healthcare.gov website has already become famous worldwide. The site which was meant to be a main asset of Obamacare, the national health plan, included in the election manifesto of the American president. It intended to improve the way in which a major part of the population made their health plans: the part that couldn't afford them.

The project, developed by a company called CGI, is now finished. The result, 292 million dollars have disappeared down the drain. There's a website that collapses when a few hundred users connect at the same time. It also makes frequent mistakes when calculating the monthly plan of many of its users.

An expert team analysed the failure of this project. They found a lack of metrics or dashboards to check on the progress of each of the development phases. Project Management had no idea about the current state of things. This called to mind this classical axiom about quality. It is normally attributed to Peter F. Drucker: "*If it cannot be defined, it cannot be measured. If it cannot be measured, it cannot improve. And what cannot be improved, eventually deteriorates.*" There are many variations of this statement and many versions about its authorship. I prefer to use a somewhat more positive version: "*if it can be measured, it can be improved."*

A basic metric for project management with Scrum is to add up how much work is still pending. Every day we can add up all our pending task estimates for the current sprint. This will give us the total amount of pending work until the end of the current iteration: the Release Burndown. With a bit of luck on the next day we will have finished some new tasks and the addition will result in a smaller number. If we represent this on a graph, it will go down until it reaches 0 on the horizontal axis. The steeper the line, the faster our team is going. We can do the same with the Product Pile if we add up the estimates of all the pending features.

Another useful and simple graph that we can use is CFD: the Cumulative Flow Diagram. First we take the data from the Kanban board, where we represent how the project is going. Then, each week, we jot down how many tasks we have under way and what state they are in. Imagine, for example, that we have a board in which we only have tasks in three states. These states are: *To-Do*, *In Progress,* and *Done*. The graph would look something like this:



At the beginning of the project, all tasks would be pending (in blue, *To-Do*). At the end of the project all would be *Done* (in green).

This representation has not been chosen randomly. I have observed in several projects that the graph tends to have this shape. The speed is slow at the beginning (not very steep), when the team is getting together. Then things gear into high speed once the initial friction has been overcome. At the end, it is slow again, when some things that had been considered correctly completed are found not to be, and that some other things should be corrected. This means that the definition of Done or Finished was not applied correctly.

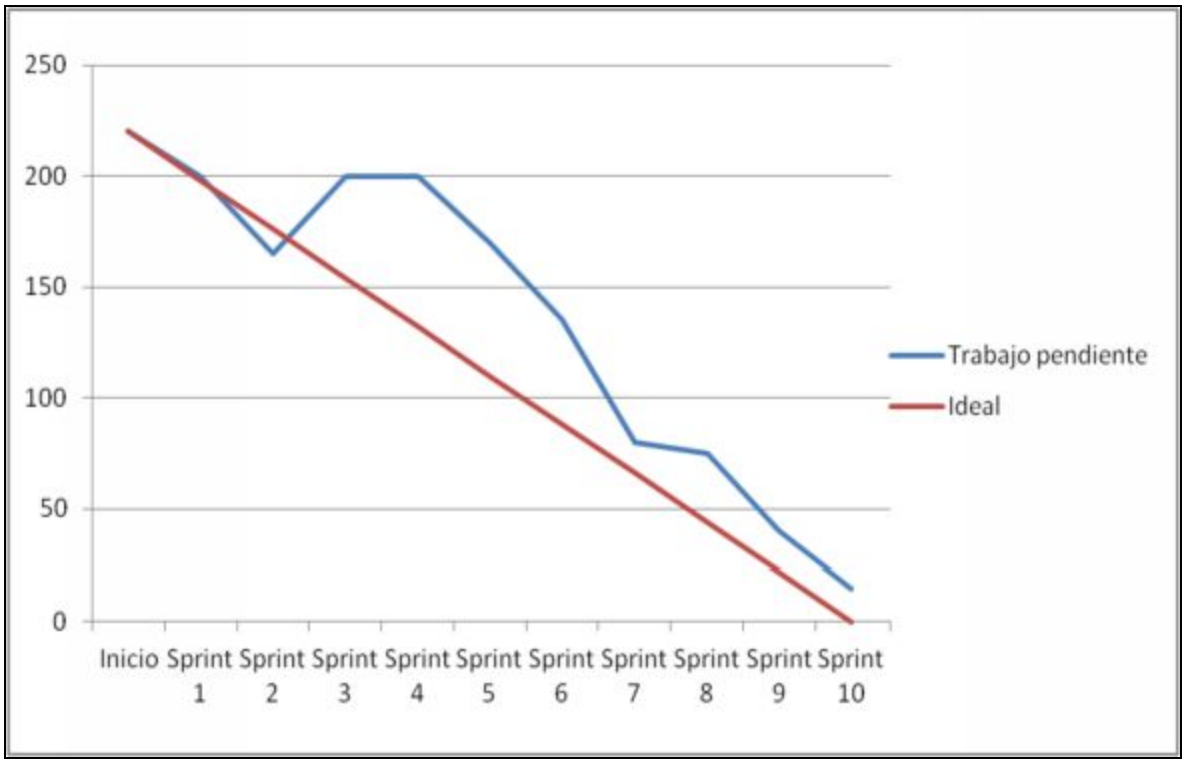
The burndown chart

A burndown chart is a very useful tool, and it's really easy to maintain. It will allow us to see at a glance how the project is going. The curve is steep, we're fine. The curve is not steep at all, we're not good.

It's not only useful, it's also very easy to understand. On the horizontal (x) axis, we have the number of work weeks or sprints we've predicted. On the vertical axis, we have the amount of work pending. The higher the line is, the more work we have left to do. If it is close to zero, we have very little work to do.

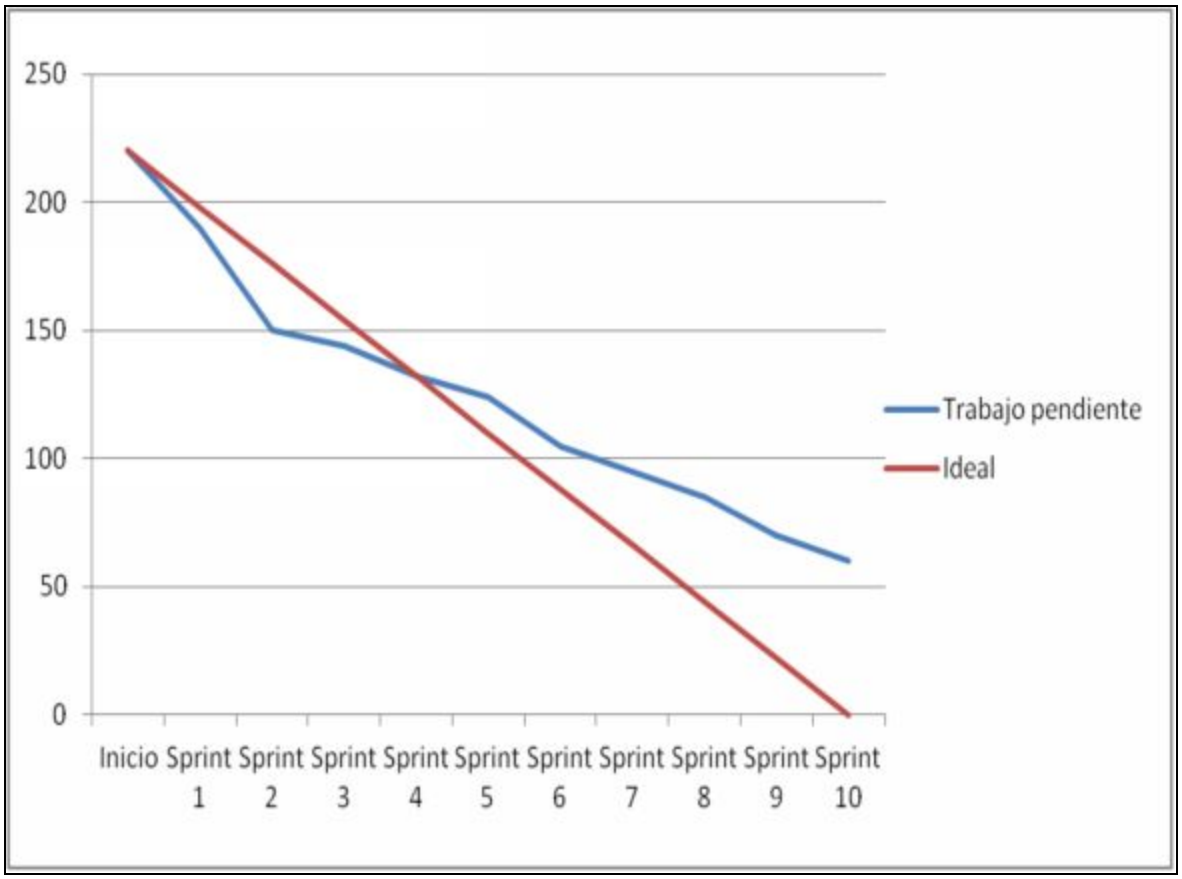
Let's have a look at the different types of graphs that are common in projects. We'll try to explain what might be going on or what can we do about it:

The graph that goes up instead of down



Sprints 1 and 2 were OK. But the team has re-estimated the amount of pending work. It's way more complex than initially thought. Or maybe some new tasks have been added to the project in sprints 3 and 4. Luckily, during sprints 5 and 6 the tasks progressed quickly. The project will not suffer any significant delay despite the change of plans.

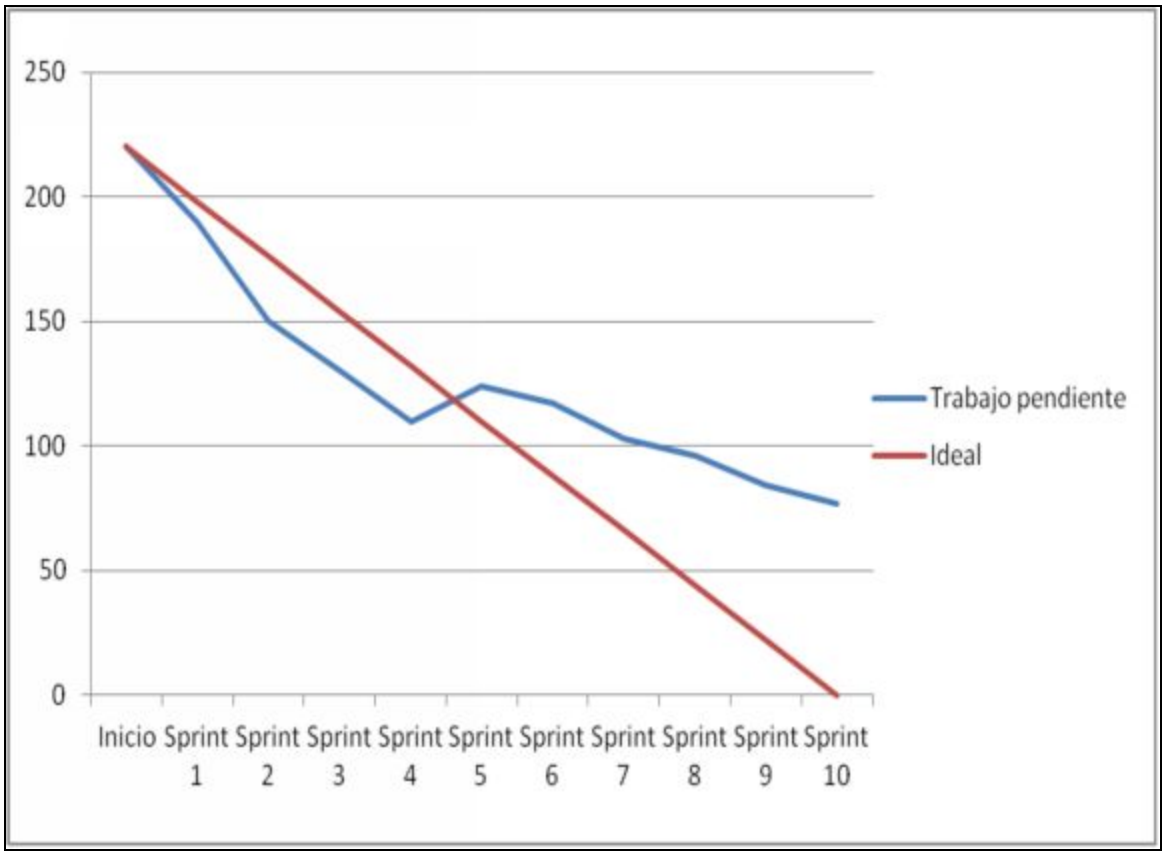
Quick at the beginning, slow at the end



Even though the initial sprints produced results very fast, the third and fourth sprint reveal the real speed of the team. This may happen for many reasons. Maybe they started by the less complex tasks at the beginning. They tackled the tasks they were less uncertain about, and left the most demanding for later. Maybe there was a mistake estimating the time needed to complete those tasks. Or maybe they were considered much simpler than they ended up being.

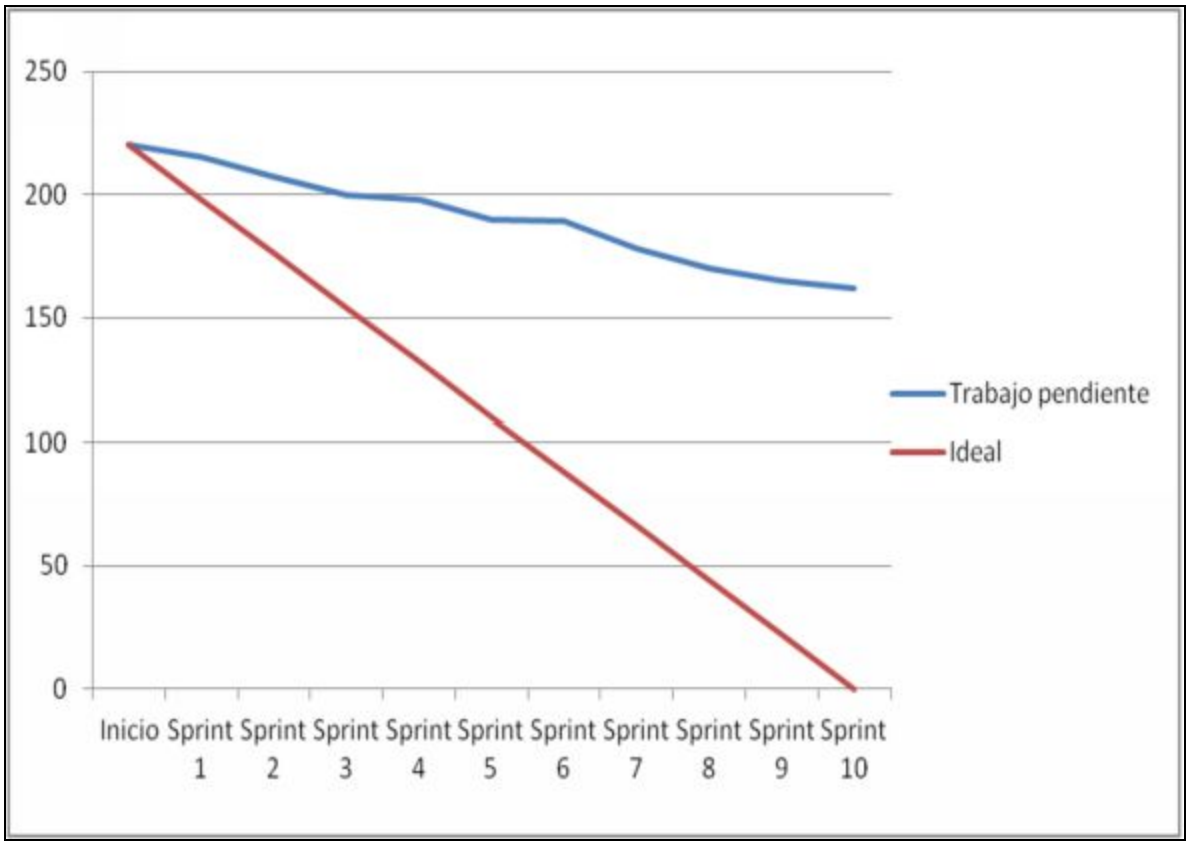
As we can see in the graph, the project will not be finished on time. If we want the work-pending line to reach zero, we will need at least 4 or 5 more sprints. The team is finishing an average of 11 points per week. In sprint 10, the moment initially set as the deadline, there are still 60 points left. So if this speed goes on, we will finish by sprint 15 or 16.

Pending work or work done



In this graph we can see that the pending work line dips very quickly during the first sprints, but descends far more slowly in the last ones. What may have happened here is that the pending work has been mixed up with work done. This is the usual result of the following thought process. "*If the task is estimated to take 40 hours, and I've worked on it for 25 hours, then there are still 15 hours to go.*" If we do this, we could work for those remaining 15 hours and the task wouldn't be finished yet. The 40 initial hours were just an estimate. Actually, what we should have recorded is the number of hours that we think we still have to go. Even if there are way more than the 15 that we have left from the initial estimate.

The graph that never goes down



This is the most feared graph. It is nearly horizontal, and calculations of when can it might touch ground can cause panic. There might be many reasons for this, but the main ones could be:

□ The team working on the project is inexperienced. Or the technology used is completely new for them, so they can't complete the tasks in the time initially projected. The graph will tell us, from the first weeks on, that the project is not going well. This way we will know that we should face the problem. We can provide help in the shape, for example, of a more experienced new member of the team. Adding people to the team, especially when the project is quite advanced, also has its risks.

□ Another possible cause is that the definition of Done has not been used correctly. Some tasks may have been deemed finished while still containing errors. And so the client or Product Owner does not consider them finished. Thus the points are not awarded in the Demo Meeting. Maybe the Product Owner is not marking tasks as finished, and instead is postponing the final approval indefinitely.

LESSONS LEARNT

"Failure becomes success when we learn from it."

Malcolm Forbes

Mistakes are made all the time when we manage projects—at least I have made a few. In this section I will list only some of them. They are the most relevant or the most confessable ones, depending on your point of view. I've messed up in many other ways, but this chapter had to have a limit. So, here they are:

First the problem, then the solution

This is the order: first you identify a problem, and then you come up with a solution. Never the other way round. It is quite common to hear about a *modern* solution that we apply to the project with enthusiasm. No matter whether there was a problem to solve or not. What is the use of deploying the new and sophisticated software for Test Driven Development in our daily work, if your problem with that project is way more basic? Or if you don't have a tight grip on the version control system?

Counting the steps instead of looking at the road

We all know that in some projects the number of hours spent tends to go through the roof. Only rarely do the predicted number of hours and the actual ones coincide. Our first reaction is normally to double-check every hour registered. Then we draft complex graphs that show us how well we're doing versus the estimate…. But does this actually help us finish the project on time?

Imagine that someone hired us to carry a heavy load from point A to point B. We estimated that it would take 10,000 steps. What is the use of knowing that we've walked 5,000 steps already? Do we know where we are, or if we've been going in circles? Wouldn't it be better to raise our head to see whether we are on the right path? Maybe double check that the road leads from A to B, or see whether there is an easier way to get there?

Phantom requirements

Sometimes we ask our team members to change what they've already done because "the client won't accept it like that." Or because we think the product should work in some other way. Our colleague had already finished a proposal that took time and effort. It's better to see whether the client likes it. The client will decide whether it's right or wrong. Investing time in unsolicited corrections will only delay delivery. Careful, though —this doesn't mean that we shouldn't review or check our work for quality before showing it to the client. All I'm saying is: "*if it ain't broke, don't fix it.*"

Spreading the need to rush

Project Managers normally work on several projects at the same time. This implies that they juggle several clients, with their needs and deadlines, and thus accumulate tension and stress. Personally I try not to spread the stress to other members of the work team and to supply all the serenity I can. Of course, every team member must be aware of our delivery deadlines and needs to know which job we should

finish for each client. Adding the need to rush to the daily routine only brings quality problems in the final product. Or creates things that are only half solved and that you will have to solve for good later on. This will also make us spend double the amount of time that those tasks would have required in the first place.

Could you make it easier?

One of the first mistakes in a project is to start every task as soon as possible without stopping and planning which steps should be part of it, and without determining whether they are truly necessary or whether we could simplify them somehow. I mean not only implementing it in the easiest possible way, but also simplifying the task in itself. Some examples: For this complex system that interconnects several computers…. Isn't there already some sort of pre-defined standard? Should this parser be able to solve third-order equations? The best way to spend your time is reducing the complexity of the work to be done.

Five lessons learnt in project management

Every time I finish a project, I try to make a balance report for myself. I look at the things that went wrong and the things that went right. The things that I tried and worked, and those that I shouldn't repeat. Even those times when that final balance came up negative, maybe the project was worth it if in the next one I don't make the same missteps again.

I have combed those balances and extracted five of the most important lessons I have learned. Some are beginner's pitfalls. Others I should have solved by simple logic but matters weren't so obvious when I was in the middle of things. Here they are:

Agile works

It's difficult to quantify, but ever since I started using Scrum in my projects, the number of hours spent per feature has gone down —and in my opinion quite considerably so. We were able to gain 80% of Scrum's advantages with three things. First, the daily 15-minute follow-up meeting. Second, keeping the graph of the state of the project that tells us whether we're doing well or not. And third, delivering software every the two weeks. Also, every fortnight, clients seemed to feel more comfortable when they received the items that we had agreed on two weeks before. In July and August it was not possible to meet. So every week we delivered a simple two-page document with the percentage of completion of each feature. This document included two paragraphs explaining why those percentages were that way. This helped us to regain credibility in a project that was quite difficult —especially in September when they could check that those percentages were real.

Not everything is positive in Scrum

Being a Scrum Master takes far more work than simply being a project manager. In those daily 15-minute follow-ups you will be told about lots of difficulties that the team needs to overcome before they can continue. Trying to find solutions will take the rest of your morning.

And having a deadline every 2 weeks can be exhausting. You cannot sprint for months. Six months later you start trotting (and that, with luck). You need to take this into account from the first planning meeting on.

Last, but not least, Scrum is not magic. Whatever method you use, you are going to need a team trained for the job at hand, ready to roll up their sleeves and work, and willing to contribute. Teams like this don't grow on trees. If you have one, your job as a project manager is to get out of the way as much as possible.

Adding more workers to a project that's behind schedule will only delay it further

This was sufficiently covered a long time ago in the book *The Mythical Man-Month*. Even so, it's still necessary to highlight it, as there are hardly any exceptions to this rule. No matter how tight the deadlines are, nine women cannot create a baby in one month.

Who owns all this?

No matter what we call it —designating the Product Owner, identifying the stakeholders, engaging them—the end we're going to need to know who is going to validate the project in real life. And I do not mean who is going to green-light the invoice. You'll also need to know who is going to use the product. The project will be a success only if, after delivering it, it works and it's useful.

At some point during the project the area manager gave us all the documentation, validated the partial deliveries, tested the whole application, and congratulated us on our work. Regrettably, when his secretary came into the training session a week before deployment, she said: "*that's of no use to me: that's no longer the right template, as I need to gather different information now.*" This meant a week of extra hours and additional effort, on top of the risk of deploying a product that could be unstable.

Minimum Viable Product

If you already have something that could be helpful to the user, deliver it, deploy it, put it up for sale. Don't wait until you have each and every one of the features finished. If you remember the Pareto principle, with just 20% of your planned features you'll be able to cover 80% of the uses that your product will have.

When it's in production, you'll be able to get feedback from the users. They know with more precision what they need, and you'll know where you've failed and how you can fix it. If you bet on just one final delivery, you only have one bullet to hit the bull's eye. If we had done this, it would have saved us a lot of problems with the product of the previous example.

Lessons learnt from others

An old Spanish saying says "*no one learns a shred from another person's head.*" But let's try it anyway by reading some experiences other people encountered while managing projects.

In the previous section I've told you about my own mistakes in managing projects. In this section I will tell you about the main lessons Jeff Sutherland learned while helping companies like Nokia, Patient Keeper or even Google to adopt Scrum:

In Google, some of the teams adopted Scrum little by little. They did this because they feared resistance from their engineers. They thought that introducing a formal process in teams that were already highly productive would not speed them up but slow them down. They started implementing Scrum using only the most basic techniques. If Google can do it, why can't we?

In Google, they also had a certain resistance to meeting daily. They thought it would be time spent in vain. They started with meetings that lasted a good deal longer than the initial 15 minutes. At first each engineer took quite a long time to explain what he or she was doing and the problems being faced. After a while, when all the members had explained their job and were aware of the job of the rest of the teammates, the meetings were much more fluid and helpful. To my relief, in their simplified, initial Scrum, they didn't use a burndown chart for every iteration. They used only one per delivery. That made me feel better. I hadn't been able to keep the Kanban board updated daily for each sprint (yes, I know, *Scrumbutophobia*).

The burndown chart helped them realise how long they would spend in developing a feature. For a feature that they initially estimated at 40 points and 3 weeks, they saw that in the first week they had covered only 8 points. In the second week they managed to finish only 7.5 points. But one of the managers thought that they had the feature ready in the third week. Then the graph made it clear that it wouldn't be ready in that amount of time.

Jeff Sutherland recalls a project for the health sector which would be accessed online simultaneously by hundreds of doctors and other users. In that project, the definition of *Done* became more and more stringent. At first, they just had to pass the unit tests and acceptance tests. But later on, for something to be really marked as *Done* or *Finished*, it had to meet this condition: "*deploy to production, and if the phone doesn't ring for 1 hour, the new delivery is complete.*"

Why projects fail

There are many reasons, and if we stop and think, I'm sure we'll be able to come up with a few. But there's one less obvious one. Stats seem to indicate that if we reduce the size of our project we will be able to increase its chance of success by 50%.

A study in projects over 1 million dollars indicates that it's 50% more probable that they fail than projects under 350,000 dollars. How come we hadn't thought about this before? Small projects are easier to manage and execute than bigger ones.

In big projects that will take more than a year, we tend to establish meetings, milestones, and reviews with more frequency. We meet monthly, bi-monthly or even every three months. That periodicity is not enough

to take the project's pulse, to check whether we're working in the right direction, whether the things that are needed are being delivered, or whether we're veering too far from our estimates.

With projects that take more than a year, or even less, there's an added risk. The project nears completion, but the goals and needs of the client's business have changed, when compared with the original reasons for commissioning the project. If we develop a mobile application, will the market be the same in a year when we start to sell it? And if we wanted to make an app based on the current labour legislation, will it be useful in a year or two when our product ships? It seems best to consider big projects as "*action programmes,"* and subdivide them into smaller projects. Each would then be delivered as one part of the overall result.

I would add one more point. Big projects tend to be planned and budgeted in this way because they set out to achieve ambitious and complex goals. Unfortunately, with each extra order of complexity, the time needed to solve it multiplies.

If we look closely, the IT world seems to go in a different direction. It tries to maintain projects as simple as possible. A few years ago, the most popular software projects have menus filled with features, characteristics, and settings. Let's remember for example every version of MS Office or MS Outlook. Most projects nowadays, no matter whether they reside in the cloud or in our mobile devices, are much simpler: a few text boxes and an OK button. And that's it!

David Karp, founder of Tumblr, said about this subject: "*Every feature has a maintenance cost. Having fewer features allows us to focus on the truly important ones, and in making sure that they work really well.*"

I think it's a great outlook. How many features have I ended up developing, that I had considered so useful! But later they were never used at all.

Agile: that's for nerds

In IT, there are very few things as fashionable as Agile, Scrum or Kanban. Many companies are adopting these techniques in their projects, but not all are successful. Often they fail miserably at the first try, and decide never to try again. They say: "*Scrum is not for us. We already tried it and it's no good for our projects.*" Or: "*The client didn't like it. They wanted their Gantt diagrams and their end-of-year delivery deadline.*"

Normally this is not attributable to our type of project or how special our requirements are. We tend to think our projects are very specific or more complex than any others, but most of us do not work on such unusual projects. Adopting Agile techniques is not easy. They represent an important change in the way we work. When things start to go astray, many will point the finger at the Scrum Master and those weird techniques that are detouring them from the planned course.

If you decide to leave all this behind and go ahead with Scrum in your next project, here are some of the problems you will find:

Clients: this is for nerds

Not every client will have ever heard about Agile techniques. Many will probably not care much about how you and your team get organised. They want the results they have paid for. For that they will supervise your work and demand deadlines and commitments. Their plans did not include hearing you speak using weird terms.

In the first meeting of one of my first Scrum projects, I set out to explain the methodology we would be using and how we would work internally. I started speaking about deliverables, sprints and Scrum Masters, to then go on talking about daily meetings and burndown charts. I hadn't explained much yet when I looked at their blank stares. I skipped the rest of my points and went straight to the conclusion. Nobody noticed or asked any questions about it.

Since that day I do not dwell on many details about our procedures. Of course I don't use any terms that might sound strange to anybody unfamiliar with the methodology. To my surprise, after a while many clients end up asking me what this way of working is called. Those adapt quite quickly to sprint times and spikes.

Managers: Scrum is anarchy

The main concern of the manager of an IT company is normally that you develop a good product. But even more so that you do it on time and within budget. If you do not comply, you can dig a huge hole in the company accounts. On top of that you may lose a potential happy client. It's too risky to leave it in the hands of new methodologies.

It may be soothing to hear a boss speak about calendars, cost control and occupancy, instead of about frequent deliveries and working software. I don't question that those concepts are important too. But they are of little use if clients don't have a working piece of software they can rely on. The project manager will have to juggle his or her way through billing reports, hour reports, or estimates. But most of all, the manager should not forget that the main job is to deliver working software.

Effort and practice

Good frameworks, good methods, and good tools will help you make your job easier, no doubt. But don't forget you'll have to clock quite a lot of hours on top of that. Hours of work and effort to try and make things a little better, a little faster or a bit higher in quality, every time. Your client, your team, and your family will be grateful.

Many years ago, the world started to investigate the plasticity of the human brain, its capacity to regenerate or adapt. Apparently, with practice and stimulation, some areas could improve their results.

Back then, great geniuses were asked to donate their brains to science. —after they died (funny enough, nobody wanted to donate them *before* dying).

Scientists wanted to study what made those people special. Once they discovered what it was, they thought that we all could stimulate and exercise those areas of the brain, and thus become geniuses. But after studying many brains, they couldn't find anything special. Neurons were not faster, they didn't have a higher number of connections between them, nothing. The only thing they found is that one area of the brain was hyper-developed, bigger than normal. It was the area attributed to memory, the part the brain used to remember things. But scientists had already seen that area developed, like in London taxi drivers. London is a huge city, with thousands and thousands of streets. Taxi drivers are required to memorize all the streets in the city, plus know the shortest route from one point to another. Clearly, London taxi drivers are not geniuses (at least, not all of them). So, what made all those Nobel-prize winners who donated their brains to science so special?

A few years later, another theory appeared that tried to explain this phenomenon. It was "*the 10,000-hour rule*" from the book *Outliers.* This theory states that people that excelled in their professions or in sports had invested at least 10,000 hours in perfecting their technique. The book showed that the first violin in an orchestra had 10,000 hours of practice, the second violin 7,000 hours, and so on and so forth. The same numbers were valid for the main surgeon at a hospital or the golfer that was all the rage that year on the course. Macauley, a great basketball player from the 1950s, who went on to be the first MVP in an All-Star match, said: "*When you are not practicing, remember, someone somewhere is practicing, and when you meet him, he will win.*" But (there's always a *but*), this theory doesn't explain everything. We all know basketball players, golfers or football players that have spent more than 10,000 hours training and will never get to regional third class. I wouldn't want to live close to the violinist who after 2,000 practice hours is still studying beginner's pieces.

And here's where we get to a third theory. This one explains that what these people had in their heads was nothing but curiosity. Yes, simple curiosity. That is, the drive to construct, to know, to understand, whatever we want to call it. This is what made them geniuses. The ones who donated their brains to science had enough enthusiasm to spend 10,000 hours, or 20,000 hours of their time, or even more, to do something they were passionate about. Trying new stuff when the old stuff didn't work, and aiming for these new things to take them along different paths.

So say you want to become, for example, an expert in project management. You are already aware, more or less, of the effort you have to put into it. But make sure that you really like what you're getting into. You'll invest lots of time in it. If you're not passionate about it, it will be hard for you to reach an advanced level. And 10,000 hours repeating the same thing will not help you more than the first 50 hours you spent learning how to do it.

AT THE END OF IT ALL

"The methodology, procedures, and policies in your projects should work for you, and not the other way round."

Think like a Project manager Blog

Project manager… and now what?

We have just been appointed project managers. They've given us that role because we've had good careers as technicians in our field, and they've rewarded us with this responsibility. Actually, we don't know much about whatever it is we have to do from now on. We only know that the project manager is "*the one who gives the orders,*" the person who always demanded that we carried through with our responsibilities. We look it up on Wikipedia, and we find that our mission is to achieve the project's goals by managing budget, time, and scope… no big deal!

Up to this point, we didn't know much about the cost of a project. Our boss dealt with that, although we were always told that money was tight. About the time needed for the project, the only thing we knew is that we were always behind schedule. We missed deadlines. We knew it was quite normal to work over several weekends every time there was a delivery. About scope, it was also quite common that the thing we ended up building was quite different to the one we envisaged at first. We always had new requirements coming at us. All this is now our responsibility…. How will we ever be able to make this new project stay on track?

We can adopt a monitoring attitude. We can focus on a precise, exhaustive accounting of every expense and hour spent. We can assign and unassign resources, and just demand the work team to meet the required deadlines. Unfortunately, sticking to a timeline drafted before knowing nearly anything about the project will not make it any easier to meet those deadlines. Noting down in our incidence-management tool that we've already spent 2,000 hours on the project will not tell us whether we're building the product that the client needs, or even whether we're going in the right direction.

The longer I spend managing projects, the more convinced I am that the project manager's job doesn't consist only of demanding explanations and insisting that people fulfil their responsibilities. The job is to help finish those projects. Making everyone's job easier and looking for ways to facilitate the work. It's not simple, but that's why we've been given that responsibility.

Our job would be easier if we always had an experienced team that could successfully dive into any task. We wouldn't need to spend weeks in training or have to worry about making the job easier. *They* would make our job easy. We would just have to sit back and wait till they finished the project. But, where would we find teams like that?

Sad but true, talent and experience do not grow on trees. You could start by paying more and more to attract the best. But your competitors will do the same as well so that they can keep their good professionals. And let's not fool ourselves, your company is not Google. This is an option only for

companies with huge profits. And no matter how much they pay, there will always be excellent professionals working for others.

Your work team and you (as you are a part of it), with all the difficulties that you may have, will have to do the best job you can. That's your main task now: to get the best out of your team. I'm sure they'll be able to do great things, with a bit of effort. You know: "*in the long term, it's not the most brilliant who succeed, but the average talents that make a habit out of defeating laziness.*", (Albert Cubeles).

We can always do something to make the best use of the team you have. For example:

Help people be more productive

Procrastination, or laziness when faced with the most difficult tasks, and the minor chaos of organising our daily tasks are challenges that every worker faces. Yes, project managers too.

To avoid this, I believe that following Agile methodologies such as Scrum have helped me quite a lot. Every two weeks we have to deliver what we're working on. A delivery that has been reviewed and checked. We can't leave everything until the end of the project and then see what works and what doesn't. It would be like designing and building a car in one piece and then sell it without having tested whether the motor works or whether the electric system can start the car and switch on the lights.

Another practice that I believe helps to improve productivity is the habit of holding daily meetings with the team members. This helps us stay aware of the things we have to do every day and of the problems that are coming up, but most of all it helps us see how close we are to the next delivery.

Give the team peace of mind

This was mentioned in *The Mythical Man-month* by Frederick Brooks. It's vital to set aside enough time for each task. If we give less time than necessary to finish a task, not only can the quality of the project be compromised, but we could actually take longer than we would have taken if we had planned appropriately.

If, due to external pressure, we agree to finish in three months a task that would normally take six, it may end up taking nine instead. When the first three months are gone and we see that the task is not finished yet, pressure and stress will increase. That in turn will take the quality of the product for a scary ride. We will also be tempted to add more people to the project so we finish earlier, but that is not free. Several members will have to stop for weeks or months to train the new members. And during that time, no one (not the new members, not the old) will be able to make any real headway on the pending tasks. We would maybe also be tempted to give new members less training, making it doubtful that they would be productive before the project is over.

Another way to give your team piece of mind is by not interrupting. This is one of the things that I've found hardest to learn. The daily hustle makes us think that everything that arrives in our inbox is even

more important than the thing that came before it. We spend all day asking everyone to stop what they're doing and put out the latest fire.

Constant interruptions prevent workers from having the sort of calm needed to finish tasks that require concentration. To avoid these interruptions, I strive to:

□ Concentrate them all in one time the day. Preferably, I set them for the last thing before the end of the day.

□ Request the meeting well ahead of time so they keep that hour free for this purpose. That way they'll be able to properly organise the rest of their tasks for the day.

□ Make interruptions as short as possible. Be concise when explaining the issue. Prepare everything needed to resolve the issue before holding the meeting itself.

Keep it fun

This is the hardest part. We are all under pressure and have our problems. But if we try to keep stress levels to a minimum, if we give the team peace of mind and remember that managing means making things easy —not just monitoring— then we will make our work not only less wearisome, but maybe even fun.

Certifications

Nothing will teach you more about managing a project than managing one, then another one, and then another. Of course, it's good to know the theory that so many project managers have written in so many books and guides, such as PMBOK (Project Management Body of Knowledge). Unfortunately, in order to get the chance to manage a project, having read a lot is not always enough. Sometimes it helps to back up our knowledge with a recognised certificate.

Right now, the three most important ones are: Professional Scrum Master (PSM) by Scrum.org, Certified Scrum Master (CSM) by Scrum Alliance and Agile Certified Professional (PMI-ACP) by PMI (the same people who provide the PMP certificate).

The first two have their origin in the same person, Ken Schwaber. Ken is one of the creators of Scrum, who, together with Jeff Sutherland, defined the initial versions of Scrum that they would formally present at the OOPSLA (Object-Oriented Programming, Systems, Languages & Applications conference) in 1995. Together they also created the Scrum Alliance, an organisation that provides Scrum professionals with the CSM certification mentioned above.

In 2010, Ken decided to leave the Scrum Alliance and created the scrum.org institute, to focus more on the dissemination of Scrum. This new institute (scrum.org) created the PSM certification (also mentioned above). Since 2012 another competitor has appeared in the arena, the PMI-ACP certification, which is generating a lot of buzz.

Comparison of Agile certifications

PSM I. The test is online only. It costs 150 dollars. There are 80 questions in English, which you must solve in 60 minutes. You need to get at least 85% right to pass. The certificate does not expire.

CSM. Before the exam you must take a two-day course, which can cost from 1,000 to 2,000 dollars. Then you take a 35-question test, in person (this was in 2012) of which you'll have to get at least 24 answers right. Apparently it is not tough. You must renew your certificate every two years, which costs 100 dollars. From 2015 you need a certain number of SEUs (which are somehow equivalent to the PDUs in PMI).

PMI-ACP. This certification requires 1,500 hours of prior Agile experience and 21 hours of an Agile training course from any institution. It consists of a 120-question test that should be taken in a Prometric centre. The certificate must be renewed every three years. Renewal requires 30 new PDUs in project management (1 PDU equals roughly one hour of training or professional activity).

Even though CSM might get more recognition, I personally chose PSM I from Scrum.org for the following reasons:

□ The certification and its materials have been created by Ken Schwaber, one of the founders of Scrum. He did this after leaving the Scrum Alliance where he shared projects with Jeff Sutherland, who is another one of the founders of this methodology.

□ The PSM certificate does not have maintenance costs, unlike the CSM.

□ Before 2012, it wasn't necessary to pass an exam, only to attend some courses by authorised professionals.

□ PSM I has many questions like "How would you respond in this situation?" This means that to pass you need to have an ample knowledge of Scrum.

□ Lastly, the PSM I certificate can be obtained online without attending any courses in person in any of the big European cities. This is important for me as I live on a faraway island in the middle of the Atlantic Ocean.

The test consists of 80 questions (true/false or multiple choice) that you need to solve in 60 minutes. So, even if you have your materials with you while you're doing your test, you won't have time to cheat if you don't know the answer. To pass the test you need to get 85% of the questions right.

How do I pass the certification exam?

I think you'll find these tips useful:

□ Carefully read the Scrum guide available on scrum.org. It's quite dense, although it's not long. In one single sentence in this guide, you may find the answers to several exam questions.

□ Read the Do Better Scrum guide. It's based on the Scrum they teach at the Scrum Alliance. Although it's not included in the official exam materials, it will give you different perspectives which are not clear in the Scrum.org guide.

□ If you're totally new to Scrum these guides are too condensed. You will need ample training that sets the ground rules and the artifacts of practical case studies. For this purpose I would recommend Henrik Kniberg's book Scrum and XP from the Trenches.

□ Take the free online test which appears in the official site, Scrum Open (take it several times if necessary).

□ Practice with the PSM I online preparation test you can find at www.antoniomartel.com. It is a page with practice tests that I created in 2014 using Ruby on Rails and deployed in the Heroku cloud. It allows you to test your Scrum knowledge with an exam of 10 random questions in Spanish. These are quite similar to the ones you can find in an official PSM I exam. It's free for the readers of this book and you can repeat the test as many times as you want (the questions will be different each time).

□ The exam is in English. Make sure to read each question carefully. Words like "*potentially"* or "*required"* can completely change the answer that you should give.

Lastly, the test costs 150 dollars, not refunded if you don't pass the exam. Good luck with it!

Yes, I'm doing Scrum, but…

The Scrum guide at scrum.org defines this framework as including "*Scrum teams and their roles, events, artifacts and rules. Each component has a specific purpose and is essential for Scrum's success.*" However, in real life we do not always execute exactly what this guide defines. We do not have time to

meet all those time limits, we change the deadline on a sprint…we adapt Scrum to the snags that keep cropping up. These excuses that we normally come up with to justify tinkering with the methodology have a name: the "*ScrumButs. Yes, I'm doing Scrum, but…*" A few years ago Nokia defined a small test —Jeff Sutherland would later modify it— to measure the degree to which their teams complied with Scrum: The ScrumBut… test.

Each Scrum practice has been designed to solve common, difficult to correct dysfunctional ties in many work teams. After all, the guide itself says that Scrum is "*easy to understand, but really hard to master.*"

Scrumdamentalism

Scrum and other Agile techniques have been popular in Europe for a few years. Yet people are already talking about *Post-Agile* and the things that will come after this "*Agile fad.*" As if to say, "*we haven't had time to learn about Scrum and already they're replacing it!*" The *Post-Agile* movement does not intend to replace Scrum or Kanban or any other similar techniques. Rather it tries to preserve the philosophy behind them.

I've observed many professionals worried about following each and every rule of Scrum. They use pompous names for their meetings and lots of cards of every colour for every type of task. They're too worried about not getting a high score on the "*Scrum But… Test"*. They say things like "*if you don't update your burndown chart daily you're not doing Scrum!*" or "*You don't convene a Retrospective Meeting after each demo? That's not Scrum.*"

There is a sort of Scrum fundamentalism that plagues IT professionals. It turns us into a tiny ITaliban group armed with our favourite programming language or the fashionable methodology of the moment. In fact, it's also known as *Scrumdamentalism* and it's defined as the fear of doing Scrum wrong. In other cases it is a kind of disease, *Scrumbutophobia*, as Henrik Kniberg calls it.

There's a far-reaching concept coming from martial arts, which calls learning phases *Shu Ha Ri*. This concept explains that, when you're learning a martial art —and this can be applied in many other fields— you go through these three stages. During the first stage, *Shu*, trainees only follow the rules they have been taught. During the second stage, *Ha*, they learn to adapt them so they work well under specific conditions. The last phase, *Ri*, is reached when every technique is known and mastered, and rules are simply "*ignored*". Those people, or those of us who have suffered that phobia of doing Scrum wrong, have suffered that symptom. We've been stuck in *Shu*, the first stage of learning.

"Screw the rules! Rules are a good start, then break them when needed (Henrik Kniberg)"

If your goal is to deliver quality products which are really useful for the client, instead of checking the contract constantly to see if you're supposed to carry out a certain task… If you forget a bit about percentages, complex processes and a myriad of tools, and instead you focus on whether you're regularly delivering demos that work (*maybe with an error or two*) instead of Power Points and progress bars… If you're doing all this, then I think you're headed in the right direction. Don't worry if you don't comply with each and every rule of Scrum.

The world speeds at blinding speeds (*"Las ciencias avanzan que es una barbaridad*")

This paraphrases a line from Zarzuela, the Spanish theatrical genre music. In the world of IT, more things go on every day, and it all happens faster and faster. For us IT professionals it takes a great deal energy to keep abreast of what's going on in every one of the technological niches.
In software development, for example, if a few years ago you worked for a bank or in accounting software, you could work in only one language. On your first day at the job they let you borrow a very thick book called "*The Bible of (insert here your favourite programming language).*" It was all there: a list with every instruction, the parameters, and examples for using them. It even had an appendix with the list of every possible bungle. Nothing could go haywire. After three years on the job, you knew the whole list of errors by heart and there wasn't a single instruction that baffled you.

Regrettably someone started placing servers farther and farther away from the user. First the database server was hidden at the back of the office, so we had to learn SQL. Then came the Internet and next to the database server we had an application server. We had to learn how to use Tomcat and how to program using Java, PHP or .NET.

When there was no more room for more servers at the back of the office, they looked for a datacentre out of town, with lots of servers from different companies. But that wasn't enough. A few years later, Google

took servers and data even farther away. They were tucked away in Arkansas or Arizona, or both places at the same time, no one really knows where exactly. Then came cloud computing and now it's starting to be a requirement to be able to program using NoSQL, MongoDB or Cassandra, but also for Android and iPhone, and….

A university degree is no longer adequate. In less than the blink of an eye, two new frameworks have come up. It's also not enough to accumulate short subsidised, badly translated courses. The kind where you get the same information sort of reshuffled from one to the next. Collecting diplomas as if they were stamps may give you a false sense of security. Actually it doesn't help much to keep yourself up to date.

Luckily, in the same way as technology makes keeping ourselves up to date elusive, it can also provide high-quality courses. You might find some of those in Miríada X, edX or Coursera. You also can get tonnes of books like this from Amazon or completely free on the Internet. Keeping updated with the latest developments in our field is a responsibility for every professional, but also a must for the project manager.

There is some debate around the issue of how much the project manager should know about the project in question. Should the project manager have full knowledge of whatever's going on in the project, or is it enough to have good skills in negotiation, planning, and resource management? Should the manager have been a good technician in the field, or is it enough to have a general idea of the work and follow the team's advice?

It's not very common for the same person to have all those abilities. Abilities such as gaining a thorough knowledge of the techniques of the field, and at the same time developing a knack for dealing with clients and the work team, as well as being canny at reaching agreements even under the hardest circumstances.

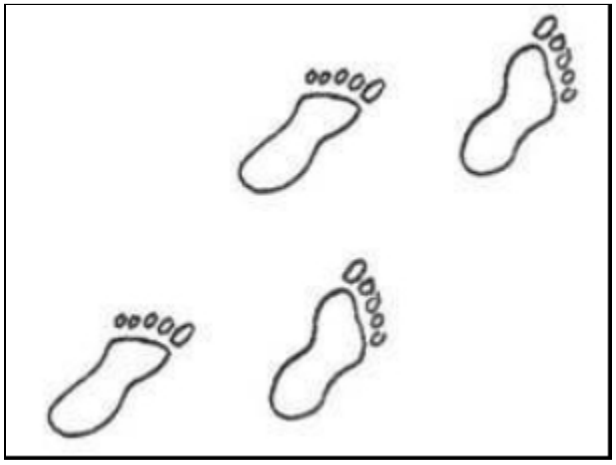
I don't wish to downplay the set of *soft-skills* necessary to become a good negotiator. However, I personally would go for a project manager that has previously been a good technician. Someone able to understand what the technicians are explaining. Otherwise, it will hardly be possible to make the right choices about what should be done or what road to follow.

Also, it's not enough for the boss to require timelines and deadlines to be scrupulously complied with. That manager should also be able to teach the team how it's done. Being able to guide and specify the tasks at hand will help a lot towards a successful completion. As Gabriel Ginebra said: "*Those who specify more, achieve more; and those who review what has been specified, achieve even more.*"

Thomas J. Watson, the cash register salesman that ended up founding IBM, told this anecdote about a conversation with his boss after his (terrible) first week in sales. He spent a whole week going from farm to farm, on a horse loaded with cash registers, without selling a single one. Instead of reprimanding or firing him, his boss explained what was lacking in his sales pitch. He told him how to do it better and went out with him on the road on each one of the sales visits to show him, in person, how it was done. Watson couldn't believe his eyes: each of the visits his boss took part in ended up in a sale agreement.

Thanks to his boss's mentoring, Watson eventually became the top salesman in his area. It's worth it to have a boss who knows how to do the job right.

EPILOGUE



"Some mistakes are too much fun to make only once."

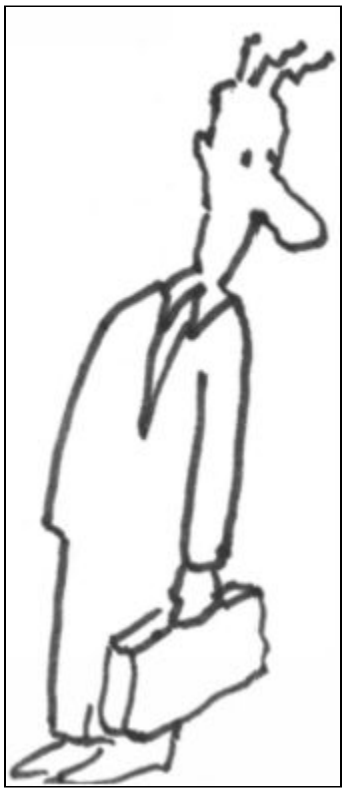
Anonymous, read at the blog *Think like a Project Manager*

So, all this you have read is nearly all I can tell about my experience with Scrum. I hope it has been useful to you, or at least it has given you a viewpoint you can add to your repertoire of ideas.

Please do not forget that in project management, and in practically every aspect of life, there is no one (much less me) who knows everything about this subject. The things that have worked for me might not be useful for you, and those that weren't useful might be just the thing you're after to make your project work. So take them all with a grain of salt.

I'm sure many of the readers have experience with Scrum as well, or with project management in general, and have lots to contribute. I would truly be grateful for any opinion or suggestion about the book that might help me improve it. Anything you can tell me will be appreciated. Feel free to contact me at my email: antonio@antoniomartel.com.

ACKNOWLEDGEMENTS



"God gave you a gift of 86,400 seconds today. Have you used one to say 'thank you?'"

William Arthur Ward

I would like to thank you for having purchased this book, but most of all, for having gotten this far. It means you have devoted part of your time to reading it and that's what I find most valuable of all. If you've enjoyed *Agile 101, Practical project management* and you've found it useful, I would be really grateful if you rated it on Amazon. Readers' reviews are important for me. They will be helpful to continue writing books about Scrum or Agile project management, and to improve, step by step, the contents of this book.

You can leave your review on this link: Reviews on Amazon.com.

I would also like to thank my brother and sister, Carmelo and María, for the many hours they spent reviewing this book and the countless corrections they suggested for the text, images, and cover. I would also like to thank them for their sincere and direct suggestions for improvement (sometimes too sincere). Without them, this book would have needed many more months to see the light of day.

There are many others I should also thank:

Begoña and David, translator and translation editor and proofreader, for their great work and care while working in this book. The translation of these last paragraphs didn't come from them, so please don't blame them for this.

Alex Caja for his great cover design, really fast response and eager collaboration whenever anything was asked to him.

I cannot forget (but cannot count all of them neither, it is half world) to everyone that gave my suggestions and ideas but also supported, encouraged and gave me time and peace of mind to write it, review it and work on it. Many, many thanks to all of them, specially to the one behind all this.

ABOUT THE AUTHOR

"It is always easier to talk about change than to make it."

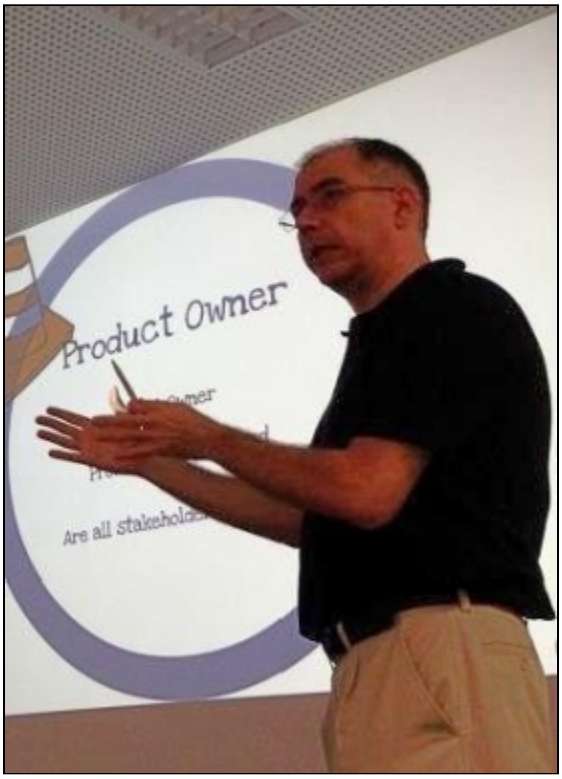
Alvin Toffler

Antonio Martel, in his role as software project manager, works in the fields of security (police), environmental management, and business intelligence in a development company based in the Canary Islands (yes, those small islands out in the Atlantic Ocean, the ones with the singing birds).

Antonio is project manager and specialises in Java and Business Intelligence solutions using Agile methodologies. He is a certified Scrum Master (PSM I) with an ample portfolio of international projects for local and regional public administrations in Spain, but also for German and Scandinavian companies.

He is the author of a number of Amazon bestsellers, like this book you are reading now, which peaked at #1 in ebook sales in Amazon Spain on August 2015 and stayed in the top 100 for 5 days in a row. It also reached #3 in the Project Management category at Amazon.com (for every language and every type of book, not just for eBooks).

He is also the author of the blog www.antoniomartel.com which focuses on improving productivity in software-development departments through the use of Agile methodologies. He has written the book Certificación Professional Scrum Master: PSM I, available on Amazon since August 2015. You can find more information in his professional profile on LinkedIn, in his Prezi about hobbies and interests.



By the same author:

If you have enjoyed this book, you can get more information on Scrum and Agile, not only on my blog www.antoniomartel.com, but also in some other publications by me:

- Scrum Pros & Cons (Prezi)

□ Practical Scrum: How we used it (Prezi)

□ Surviving the adoption of Scrum (Prezi)

□ Yes, you do Scrum but... (Prezi)

□ Post in Hackwriters.com: Google Search Tips