Lean and Agile in Software Development

Some time ago, at the company where I am currently working, I suggested that we could create some posters about Lean and Agile so that people would keep it in the back of their heads on their day to day work. I also made myself available to explain, to the IT department, what Lean and Agile are all about, but because of lack of time, that explanation ended up never happening.
Lately, I've found that there are some people around me that have the desire and need to know a bit more about Lean and Agile, about what they stand for, how their practices should impact our day to day work, and what's there to gain from it. So I decided to write this post in an attempt to cover that.
I've talked about Lean and Agile before, in a post titled *What is Lean and Agile about Software Architecture* but it had a very restricted spectrum. In this post, however, I want to take it a bit further.

Why

Businesses need to go fast. Businesses **always** need to go fast! They need to go fast to be ahead of the market, ahead of the competition, ahead of consumers and customers needs. The business survival and the jobs it guarantees can depend on how fast a company adapts to the market.
The company as a whole needs to be focused on being ahead of the market, or at least ahead of the competition. This, however, doesn't mean that everyone in the company should be individually focused on going fast. Developers certainly shouldn't!
Product development, for example, should focus on accurately understand, and even anticipate, consumers and customers needs. This will definitely contribute to keeping the company in the vanguard of the market.
Can you imagine trying to understand and anticipate the market needs, in a hurry?! How could we accurately do that without properly analysing and understanding the market? wouldn't we have a high risk of putting time and effort into creating something the market doesn't need? Wouldn't that be a waste that would hold the company back, and maybe even destroy it?
Can you imagine the risk of creating UX deliverables in a hurry, without properly thinking of the user's journey through the application?
Developers should not be focused on going fast either, they should be focused on doing things right, just like everybody else in the company!
When I think about this subject, it reminds me a lot of Robert C. Martin opinion about professionalism, and what it entails to be a developer. He talks about it extensively in one of his books, titled *The Clean Coder: A Code of Conduct for Professional Programmers*, and one of the lectures he has been doing for years, titled *Expecting Professionalism*. As he explains in his lecture, software is everywhere and a lot of it controls equipment that means the difference between life and death.
One of the examples given by Robert C. Martin is about one of the NASA shuttles that exploded after launching, killing all its passengers. As it seems, the engineers knew that there was a high chance of the shuttle having serious problems. So "why did the launch go through then?!" may you ask. Well, it seems the managers and politicians involved wanted it done fast, they didn't want to spend more time on the problem to make sure nothing bad would happen, so they ignored the engineers. And people died because of it.
Of course, not all of us build software that can have a direct impact on the life or death of others, but my point is:

- Managers know what they need to be built and under what conditions;
- Engineers know how to build it and when it is ready.

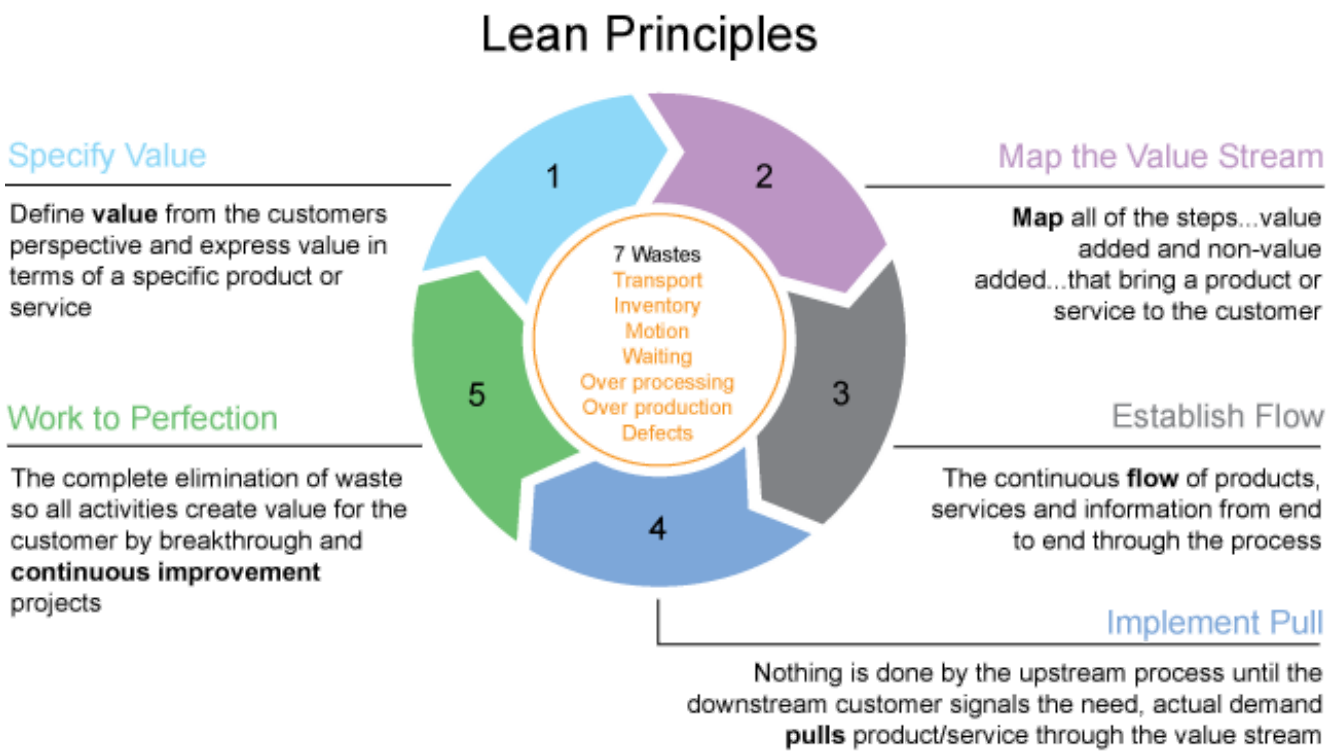
In other words, non-technical managers should refrain themselves from interfering in the development process, including pressuring the developers into "going faster".
Managers role is not to tell engineers how to do their job, their role is to work **with** the developers by handing them clear and accurate requirements and **help** the developers by leaving them free to focus on what they do best: develop software. This means:

- Handling stakeholders expectations;
- Providing the tools the developers need, like cloud repositories, production, acceptance and testing servers, continuous integration servers, etc.;
- Protect the developers from external interferences like interruptions from other colleagues or meetings irrelevant for their work (this is actually one of the main goals of the Scrum Master role).

If the company really needs the developers to "go faster", the developers should work extra hours and the company should pay for those hours accordingly. If the company does not see the need for the developers to work extra hours and get paid for those hours, then I guess the need to "go faster" is not really there.
If I am pressured to "go faster", I will end up cutting corners, doing code that is quick and dirty, working extra hours without getting paid, which will make me unhappy and less productive. This may deliver functionality sooner but it will prone to error, it will not be scalable and it will require more time to be changed later down the line when changes are needed so in the medium/long run it will definitely make us slower. For some reason, this seems to be difficult for managers to understand. In the other hand, maybe they don't really need to understand it, they should just get out of our way and let us do the job we were hired to do.
This all comes down to **doing things right**. Doing things right **helps us deliver features consistently, in a constant and somewhat predictable velocity** because there isn't much technical debt that can give us nasty surprises and slow us down. **Going fast is a result of doing things right.**
However, **what is "doing things right"?** This is where **Lean and Agile** come into play.

Lean

Lean is a production management methodology developed by Toyota during the first half of the 20th century. Later on, it was adapted to the American corporate context and named "Lean".
There are plenty of books explaining it, but the ones I see referenced more often are The Toyota Way (Liker 2004), Lean Thinking (Womack & Jones 2003) and Lean Enterprise (Humble, Molesky, O'Reilly 2015).
The end goal of Lean is to reduce waste, inconsistency and irregular production/development.



Lean has 5 principles and warns us about 7 wastes that we should strive to eliminate. These Ideas were initially thought by Toyota so, at first glance, we might think that it only applies to a factory manufacturing context. However, after some reflexion, we can see that those ideas are pretty generic and can be adapted to probably any production context.
Here's my own humble take on those principles and wastes applied to software development.

Lean Principles

1. Specify value

Define value from the customer's perspective and express value in terms of a specific product or service.
This is our main goal, it represents the actual value for the customer. Everything we produce must have some kind of contribution to this goal, although it can be an indirect contribution.
This can be defined at different levels of granularity. At a corporate level, we can see it as the core goal of the company, defined in a short and simple sentence. For example, the core goal of my current company, Werkspot, is to "***Be the easiest and most reliable way to arrange home services***". At a smaller granularity level, it can be the main goal of a specific project, for example, something like "*Make it easier for users to request a service*".

2. Map the value stream

Map all the steps... value added and non-value added... that bring a product or service to the customer.
We need to establish how we are going to bring a product together, so we need to **define all stages of production from end-to-end**, from the first step of defining what is actually needed by the users to the last step: the delivery.
In a software development project this could be something like:

1. Talk to the targeted users so we can define what they need;
2. Plan what features we will build, in what order, and in what iterations (versions);
3. Build the planned features:
    1. Create the designs;

    2. Build the functionality;
    3. Review what has been built;
    4. Move on the next feature (back to 3.a);
4. Get feedback from the users about the new features;
5. Move on to the next iteration, taking into account the received feedback (back to 3).

If we already have a value stream in place, this exercise can help us identify redundant steps so we can eliminate them, optimising the production process.
This, however, is only about defining the steps, not implementing them.

3. Establish flow

The continuous flow of products, services and information from end to end through the process.
Establishing the flow is about actually implementing the value stream. How are we going to operationalize the value stream? Who do we need in the team? Will we produce in isolated silos or will we have cross-functional teams? Will the team members work at the same location or far away from each other?
In software development, I imagine a production flow where we have a cross-functional team with (for example) a product owner, a designer, a front-end developer and a back-end developer. **The value-creating steps occur in tight sequence, if not at the same time, with close collaboration between the team members, which are located as close to each other as possible. This will increase collaboration, and reduce feedback loops between the different aspects of the product development.** The moments we spend waiting for a response from a colleague will be reduced to the minimum possible.

4. Implement pull

Nothing is done by the upstream process until the downstream customer signals the need, actual demand pulls product/service through the value stream.
By having short production cycles, we can deliver value to the customers in shorter periods of time. This means we don't have to predict what the user will need, we don't need to prepare in advance, we don't need to produce ahead of time. We can **produce only when the customer signals the need for something**.
This means we should only produce features that are actually "*requested*" by the users. Features that we build but were not "*requested*" by the users have a high risk of ending up not being used by the end users and therefore being a waste of time and resources that could otherwise have been used to build, or improve, a feature actually needed by the end-users.
It's good to keep in mind though, that the "*request*" for a feature, or improvement, can be done directly, for example by interviewing end-users, or indirectly, by analysing the end-users behaviour when using the application: do they use workarounds? how many steps does it take for them to accomplish a business process? do they have a post-it in their desktops reminding them about something they need to work with the application?
The partial product components will be delivered *just-in-time*, meaning, only **when they are needed** and only **what is needed** at that moment. In other words, it’s about doing what is needed *now* (which depends on the planning scope, maybe this sprint target or this quarter target) to keep you in the game for the long run, it’s about carefully planning and slicing the design to deliver exactly what is needed in order to support feature development in the long run.

5. Work to perfection

The complete elimination of waste so all activities create value for the customer by breakthrough and continuous improvement projects.
Perfection does not exist. It is relative to a context, and nowadays the context changes quite fast: new business cases surface, people leave the team and other people come in.
This does not mean, however, that we should not have perfection as a goal. We must continuously work towards optimising the development process. We must make the *relentless pursuit for an optimised production flow*, be part of the company culture, part of everyone's attitude. All **employees must be empowered and involved in implementing lean**.

Lean Wastes

1. Transport

The waiting time while moving things around.
In software development, our things are the code and the data so this is, for example, about deploying code: how much time does it take to download and install all dependencies? How much time does it take to run tests before deploying? how much time does it actually take to place all code in the production server, ready to run?
We must strive to reduce the time wasted waiting for these tasks to be completed.

2. Inventory

Keeping products around, hoping they will be needed in the future.
When developing software, inventory is the code we produce that "will be used in the future" and/or keeping/maintaining code that might be reused in the future. The reality is, however, that we can not predict the future: requirements change constantly. So why should we allocate time and resources to produce code that might never be used or maintaining code that might never be reused?
So, basically:

- **Develop *just-in-time*:**
  - **Only build code when it is actually needed***;*
  - Only upgrade servers when it's actually needed;
- **If code stops being used: remove it immediately**, don't leave it laying around in the hopes that it will be needed again in the future.

3. Motion

Repetitive movement.
All the **manual repetitive tasks that we can automate**, should be automated. It will eliminate time wasted repeating them and will remove the risk of human error. For example, instead of manually issuing a shell command to copy production data to an acceptance DB, just create a Cron job to do it every night.

4. Waiting

Wait for reactions from colleagues.
We must eliminate waiting states when collaborating with colleagues. Always prefer the fastest, richest, and most direct communication method. Don't wait for days, hours nor even minutes for someone to reply to an email, if possible just make a phone call or go have a face to face conversation.

5. Over-processing

Trying to produce the perfect product.

Trying to create the perfect code (or perfect design or feature) is an inglorious goal because perfection does not exist: it is subjective to who is chasing it and depends on ever-changing requirements. So it happens that we didn't reach perfection yet and the concept of perfection has already changed because the requirements changed in the mean time.

6. Over-production

Produce more than needed.

Don't create code nor features that are not needed now. Only create code that is actually needed and only create features that have a reasonable expectation of being used by the end-users.

7. Defects

Creating products with defects.

This waste is about **building features that do not work as the user expects** them to work, either because of bugs in the implementation or because of bugs in the requirements (the requirements were not squeezed out properly from the users).

All bugs need to be fixed immediately so they stop impacting the users as soon as possible. However, bugs in requirements also need to be examined carefully so that the requirements gathering process can be fixed and we can avoid similar bugs in the future.

We need to invest time in doing things right. Invest time in getting feature requests actualy needed by the market, in getting correct requirements for the feature requests, in building features right. Not doing so, has a high risk of becoming a waste.

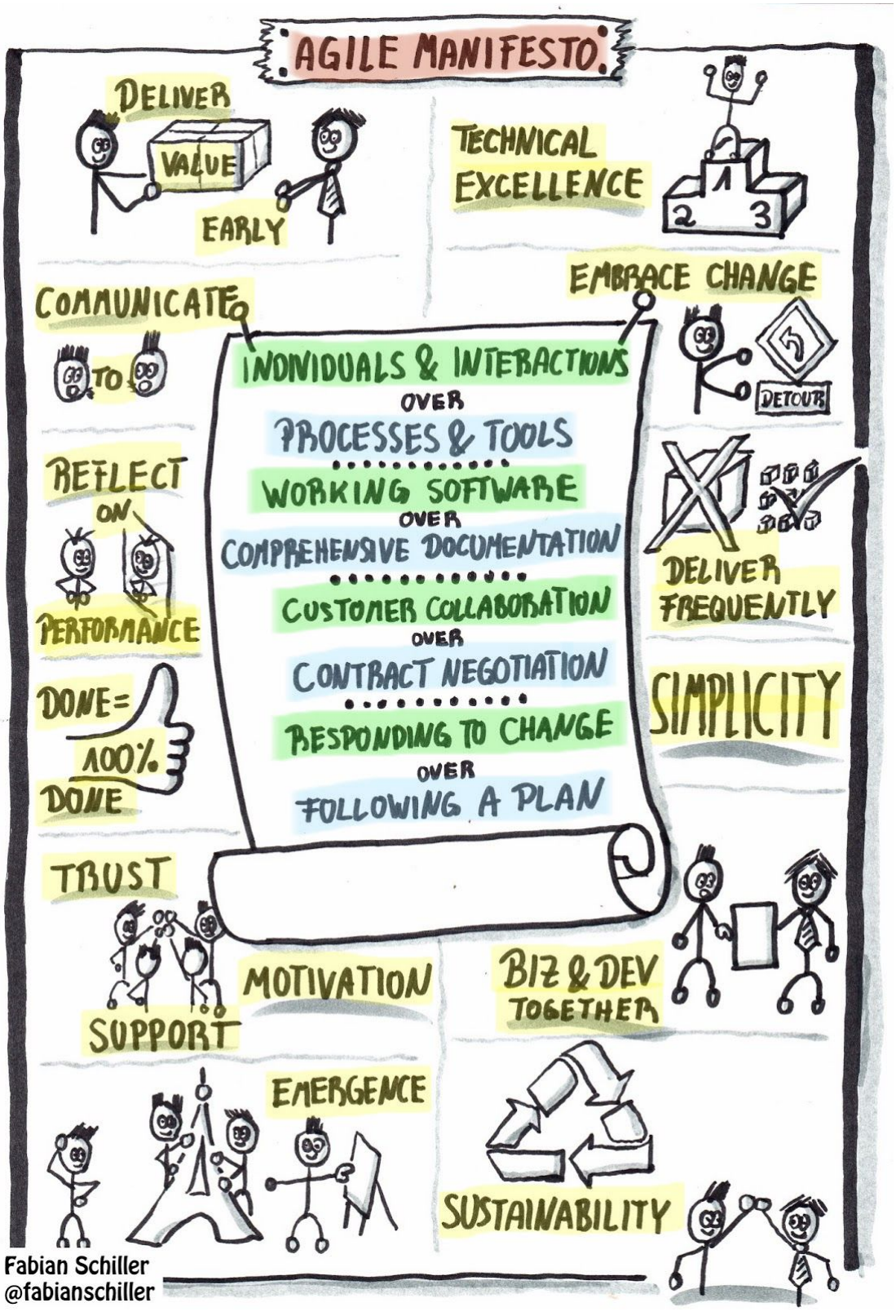
Rework because of changes in business cases is great, it means we are adapting to the market, but rework because of defects is a serious waste!

Agile

Agile has its foundations on the Agile Manifesto, which was formally written in 2001 by a group of renowned Software Developers, as a response to the frustrations of software development during the 1990s.

In an agile methodology, solutions evolve through collaboration between self-organizing, cross-functional teams using the appropriate practices for their context. The goal is to create better software, faster, using end-user engagement, short feedback loops, and eliminating waste.

The Agile manifesto is comprised of four foundational values and 12 supporting principles. There are several agile methodologies, or frameworks, like Scrum or Kanban, and they apply the agile values and principles in different ways according to, for example, the type of tasks or team experience.



I'm not going to talk about the principles because I find them straightforward and it would make this (already long) post unbearably long. But I will give my own explanation of the values.

1. Individuals & Interactions over Processes & Tools

Processes and tools are important because they give us practical and standardised ways of doing things. They make things easier and faster.

However, they do not understand the need for changes, they are not creative elements that can give us solutions to problems: people are! People are the elements that can respond to change.
So, use the tools and processes to produce, but prefer face-to-face communication with people (clients, colleagues, any stakeholder) to clarify, understand and improve: to respond to change.

2. Working Software over Comprehensive Documentation

In the old days (pré 2000s), we would need formal documentation for before, during and after project development. Theoretically, we would have the project extremely well documented. But how much of that documentation was actually needed? How much was actually ever used? How many resources were allocated to produce documentation that was barely used, if ever used at all?!
The Agile Manifesto tells us that we should only create the formal documentation that is actually needed. So:

- Don't document extensively everything that we think we need to build: after some time, what we need to build might change and the time and energy we've put in that documentation will have been a waste. So, do a high-level overview of what we need to build, and then, just-in-time, do small and accurate stories with clear requirements and acceptance criteria.
- When coding, use good naming conventions and choose names that reflect what a code unit is and does (ubiquitous language), as to reduce the need for technical documentation explaining code;
- Structure your code in such a way that there is only one logical place for a code unit to exist and it is encapsulated in the architectural and domain modules they belong to. This will, again, reduce the need for technical documentation that relates the code with the architecture and the domain;
- Design the application UI in such a way that it guides the users through the processes, and helps them not to make mistakes (build dummy proof software). For example, expose the business processes to the users in a language that they can clearly understand and help them pick which one they need. Another example is when deleting a resource, show a confirmation box where it's explained the consequences of deleting that resource. This will help reduce the need to write detailed and/or extensive user documentation.

3. Customer Collaboration over Contract Negotiation

Contracts are important. After all, they are what gives us some guarantee that we will be paid. However, we shouldn't blindly and carelessly build what is specified in the contract.
Nowadays, businesses contexts, challenges and evolution develop much faster than the speed at which we can develop a complete software application. This means that if we limit ourselves to build what is written down in a contract before a project is even started, by the time we deliver the application there is a big chance that the application will already be outdated and not compliant to the customer's current needs.
If we want our project to be successful, the moment of contract negotiation and the moment of delivery are not the only moments the customer should be involved:

- The contracts must be open to accommodate for business requirements changes, as well as the due gratifications;
- The customer (as other stakeholders) must be involved throughout the development process:
    - At the beginning of a sprint, explaining the functionality that he needs to be built;
    - At the end of a sprint, validating the functionality that has been built;

This approach will guarantee that:

1. What we build, will match the customer business needs;

2. The development team will know as soon as business needs change.

4. Responding to Change over Following a Plan

When developing a project, having a plan is essential. However, having the right type of plan is as important. Agile tells us that it doesn't make much sense to create a very detailed plan for the whole project: after all, the initial requirements are likely to change during the project development. Instead, we should define a loosely detailed plan at a macro level, and only have a detailed plan for a short period of time: the current sprint.
This approach will allow us to quickly adapt production to emerging or changing requirements. We are mostly planning as we go, so we are not tied to a rigid long term plan.
Furthermore, **emerging and changing requirements are great! That is what enables us to deliver a product that fits the customer's current needs, as opposed to what the customer needed when the project started!**
In the case of **emerging requirements**, this simply means that, for example, there is a new business case and the software needs to be able to handle it for the customer.
In the case of **changing requirements**, there are two possible reasons:

1. A **business case changed**, for example, it has different restrictions, and the software needs to be able to handle the new restrictions;
2. A business case technical requirements were not properly distilled, in other words, **there is a bug in the requirements**.

Either way, we need to adapt to those changes, we need to rethink and change the code accordingly. The difference is, however, that when there is a bug in the requirements, we must understand what led to this bug and do our best to prevent it from happening again, just like if it was a bug in the code.

Conclusion

The IT market is very competitive. For a company to be successful, it needs to adapt and produce faster than the competition. This, however, doesn't mean that everyone should be running around doing things in a rush. Doing things in a rush will lead to mistakes that can prove fatal.
We need to be smart and do things right. Doing things right will, hopefully, lead us to constant productive flow with few mishaps and in the end will make us go faster than the competition.
Lean and Agile can help us to do things right.
As James Coplien says, **Lean is about thinking** and **Agile is about doing**.
Lean focuses in preparing and optimising our work, it's about thinking how we need our workflow to be, how we should set it up, what work we actually need to do *now* and eliminate everything that is waste.
Agile focuses in getting things done:

- Build software that doesn't need documentation;
- Communicate directly, face to face, to solve problems and misunderstandings quickly and effectively;
- Involve the customer throughout the development process, so we always work on what is needed, and identify changing requirements as soon as possible
- Accept and adapt to changing requirements, so we deliver software that is actually useful.

Do it right, do it Lean & Agile, speed and quality will come as a result.

Sources

2001 - agilemanifesto.org - Manifesto for Agile Software Development
2010 - Coplien & Bjørnvig - Lean Architecture: for Agile Software Development
2016 - Mark Crawford, ASME.org - 5 Lean Principles Every Engineer Should Know
2017* - smartsheet.com - Comprehensive Guide to the Agile Manifesto

*** Seen in**