

# Agile Software Development, lessons learned

by Jerome Kehrli

---

Written on Wednesday Oct 19, 2016

---

After almost two years as Head of R&D in my current company, I believe I succeeded in bringing Agility to Software Development here by mixing what I think makes most sense out of eXtreme Programing, Scrum, Kanban, DevOps practices, Lean Startup practices, etc.

I am strong advocate of Agility at every level and all the related practices as a whole, with a clear understanding of what can be their benefits. Leveraging on the initial practices already in place to transform the development team here into a state of the art Agile team has been - and still is - one of my most important initial objectives.

I gave myself two years initially to bring this transformation to the Software Development here. After 18 months, I believe we're almost at the end of the road and its a good time to take a step back and analyze the situation, trying to clarify what we do, how we do it, and more importantly why we do it.

As a matter of fact, we are working in a **full Agile way** in the Software Development Team here and we are having not only quite a great success with it but also a lot of pleasure.

I want to share here our development methodology, the philosophy and concepts behind it, the practices we have put in place as well as the tools we are using in a detailed and precise way.

I hope and believe our lessons learned can benefit others.

As a sidenote, and to be perfectly honest, while we may not be 100% already there in regards to some of the things I am presenting in this article, at least we have identified the gap and we're moving forward. At the end of the day, this is what matters the most to me.

This article presents all the concepts and practices regarding Agile Software Development that we have put (or are putting) in place in my current company and gives our *secrete recipe* which makes us successful, with both a great productivity / short lead time on one side and great pleasure and efficiency in our every day activities on the other side.

## Table of Contents

Agile Software Development, lessons learned.....	1
1. Agile Software Development.....	2
1.1 Why Agile anyway ?.....	3
1.2 Agile Development Value Proposition.....	4
1.3 Scrum.....	5
1.4 Kanban.....	7
1.5 Prerequisites : XP !.....	7
1.6 Benefits : DevOps, Lean Startup.....	9
2. Scrum roles.....	10
3. From Story Maps to Product Backlog.....	12
3.1 User Stories.....	12
3.2 Story Maps.....	13
3.3 From User stories to Developer Tasks.....	16
4. From User Stories to Releases.....	18
4.1 Composing our releases.....	18
4.2 Composing the sprint.....	19
4.3 Estimations in Story Points.....	22
5. Introducing our sprints.....	24
5.1 Before Sprint.....	24
5.2 During Sprint.....	25
5.3 After Sprint.....	26
6. Release Backlog and Sprint Backlog.....	27
6.1 Different release backlogs, long term backlog, sprint backlog .....	27
6.2 While being Agile.....	29
6.3 Handling customer requests and production concerns.....	30
6.4 Sprint Kanban backlog management.....	30
7. Conclusion.....	32

## 1. Agile Software Development

Agile Software Development and Agile methodologies form both an approach regarding software development and a set of practices for managing and driving software development projects.

Initially really intended solely for Software Development projects, these methodologies can apply to a wide range of engineering fields.

Agile Methodologies have as origin the [Agile Manifesto](#). Written in 2001, this manifesto first used the term of *Agile* to qualify some methods that were used for a long time in various engineering fields.

The Agile Manifesto has been written by seventeen experts in the software development business, mostly those already behind the eXtreme Programming movement such as Kent Beck, Ward Cunningham or Martin Fowler.

## 1.1 Why Agile anyway ?

The experts behind the *Agile Manifesto* concluded long ago that the current *Waterfall* methodologies such as RUP (Rational Unified Process) were not adapted anymore to today's challenges in regards to today's fast evolving organizations.

The problems with the traditional Waterfall approach can be summarized as follows:

- **Incomplete or moving specification** : no matter how smart the business experts you are working with when writing specifications, no matter the time you dedicate to it, your specifications **will** be incomplete, biased and wrong. That comes from a very simple reason : it's impossible to imagine a solution just right the first time.  
Business experts will change their mind once they see what comes first out of their inputs, always.  
They need to see a first version coming from their initial inputs and see it to actually understand, with the help of the architects, what they really need. Finding the actual solution to any business requirement or problem requires iterations : a first, as simple and stupid as possible version is required to help the business understand what they really need. Then that solution needs to be refined through several additional iterations.
- **The tunnel effect** : Think of a several years software development projects. Business experts spend a few months specifying everything and then wait three years before actually seeing it coming (wrong and buggy, needless to say, but that is another story). In three years, business requirements would have changed, evolved. And even if what was specified three years ago was greatly written and well thought out, now it's neither accurate nor relevant anymore. We live in a very fast evolving world.
- **Drop of Quality to meet deadlines** : It's always the same, isn't it ? When the deadline gets closer and the team needs to rush into fixing the issues that arise from the first batch of tests (always much bigger and far more numerous than expected), when the initial feedback from the stakeholders or users come and underlines how far the product is from what is required (which is not what has been specified of course), we all do the same : we drop quality, drop testing, drop design and rush into trying to make it work.
- **Heightened tensions between teams** : so what do you think happens when after several months (years) of development, that first version is finally presented to the stakeholders and they realize that it's most definitely not what they need ? Everything turns ugly. The development team is angry because they implemented the specifications and yet they're told that the software is not good, the stakeholders and business analysts are angry

because they do not understand why the dev team is so stubborn about specification (and ashamed those were screwed), etc.

These problems most of the time lead to these consequences :

- Projects failure - slippage and inadequacy with actual needs make projet abandoned
- Exceed budget and deadline - sometimes up to ten times initial budget
- Lack of reactivity - business requirement change, project is delivered but doesn't help business in the end
- Software inadequacies (functionalities, quality)
- Teams demotivation - try to convince an engineer he has to start all over again once he is done developing something
- User dissatisfaction

### 1.2 Agile Development Value Proposition

The *Manifesto for Agile Software Development* uncovers better ways of developing software by doing it and helping others do it.

It values *individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan.*

#### **Individuals and interactions over processes and tools**

If processes and tools are seen as the way to manage product development and everything associated with it, people and the way they approach the work must conform to the processes and tools. Conformity makes it hard to accommodate new ideas, new requirements, and new thinking. Agile approaches, however, value people over process. This emphasis on individuals and teams puts the focus on people and their energy, innovation, and ability to solve problems.

#### **Working software over comprehensive documentation**

Developers should write documentation if that's the best way to achieve the relevant goals, but that there are often better ways to achieve those goals than writing static documentation.

Too much or comprehensive documentation would usually cause waste, and developers rarely trust detailed documentation because it's usually out of sync with code.

#### **Customer collaboration over contract negotiation**

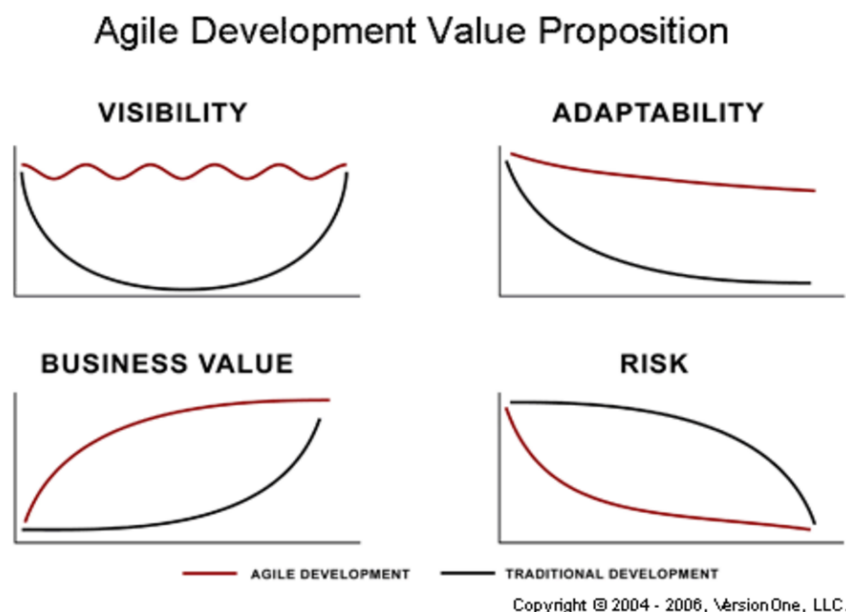
No matter which development method is followed, every team should include a customer representative (product owner in Scrum). This person is agreed by stakeholders to act on their behalf and makes a personal commitment to being available for developers to answer questions throughout the iteration. At the end of each iteration, stakeholders and the customer representative review progress and re-evaluate priorities with a view to optimizing the return on investment (ROI) and ensuring alignment with customer needs and company goals.

### Responding to change over following a plan

Adaptive methods focus on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well.

An adaptive team cannot report exactly what tasks they will do next week, but only which features they plan for next month. When asked about a release six months from now, an adaptive team might be able to report only the mission statement for the release, or a statement of expected value vs. cost.

### Summary

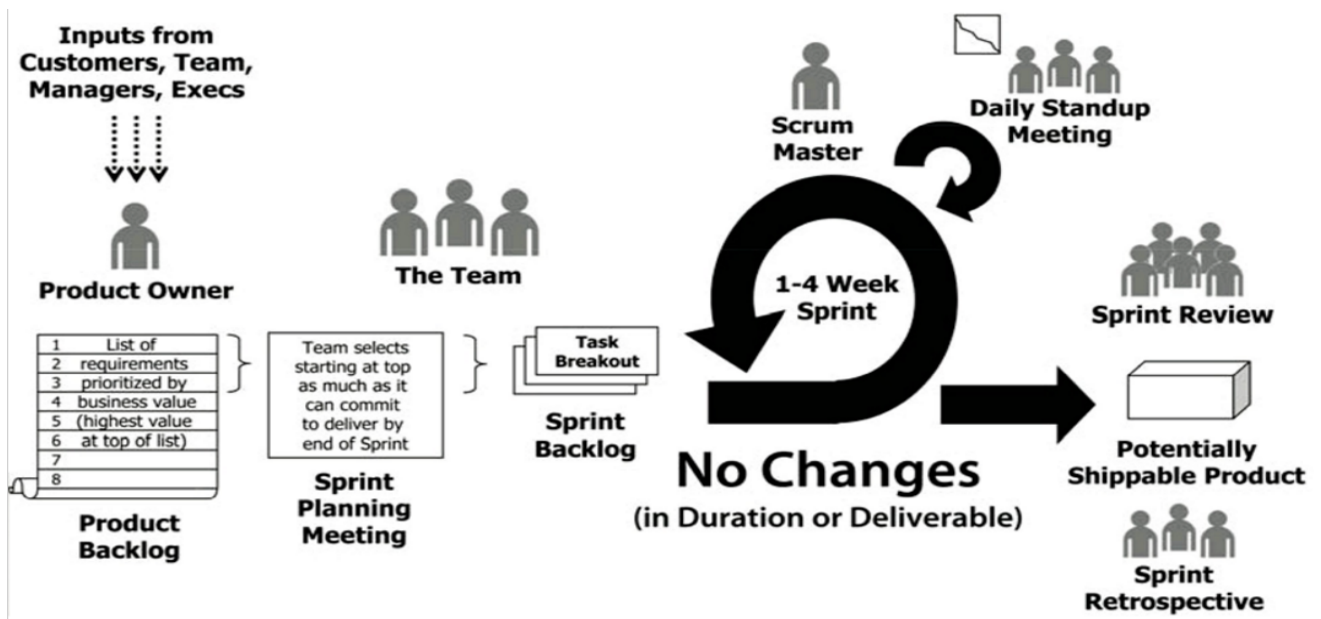


## 1.3 Scrum

### Scrum - A Fundamental Shift

Scrum is a well-defined process framework for structuring your work in an *Agile* way. Scrum consists in working in iterations, build cross-functional teams, appoint a product owner and a Scrum master, as well as introducing regular meetings for iteration planning, daily status updates and sprint reviews. The benefits of the Scrum methodology are well understood: Less superfluous specifications and fewer handovers due to cross-functional teams and more flexibility in roadmap planning

due to short sprints. Switching your organization to use Scrum is a fundamental shift which will shake up old habits and transform them into more effective ones.



Scrum consists in running the development with a tempo of two to four weeks sprints. A sprint starts with a *Sprint planning meeting* where the whole development team picks tasks from the *product backlog* until the *sprint backlog* is filled with enough tasks to fulfill the capacity of the team. A sprint finishes with a *Sprint retrospective meeting* where performance is evaluated and the sprint whereabouts are discussed.

Within the sprint, the development team meets everyday at the daily scrum to discuss everyone's tasks and activities.

At the end of the sprint, the development team delivers a production-ready software that is potentially shippable.

While from a sprint to another, priorities can change completely, the priorities, scope and duration of a sprint can never change !

This is an important aspect of the Scrum framework and ensures serenity of the team.

### Scrum leverages Commitment as Change Agent

The initial introduction of Scrum is not an end in itself. Working with Scrum, one wants to change the teams' habits: Take more responsibility, raise code quality, increase speed. As the teams commit to sprint goals, they are intrinsically motivated to get better and faster in order to deliver what they promised. Scrum leverages team commitment as change agent.

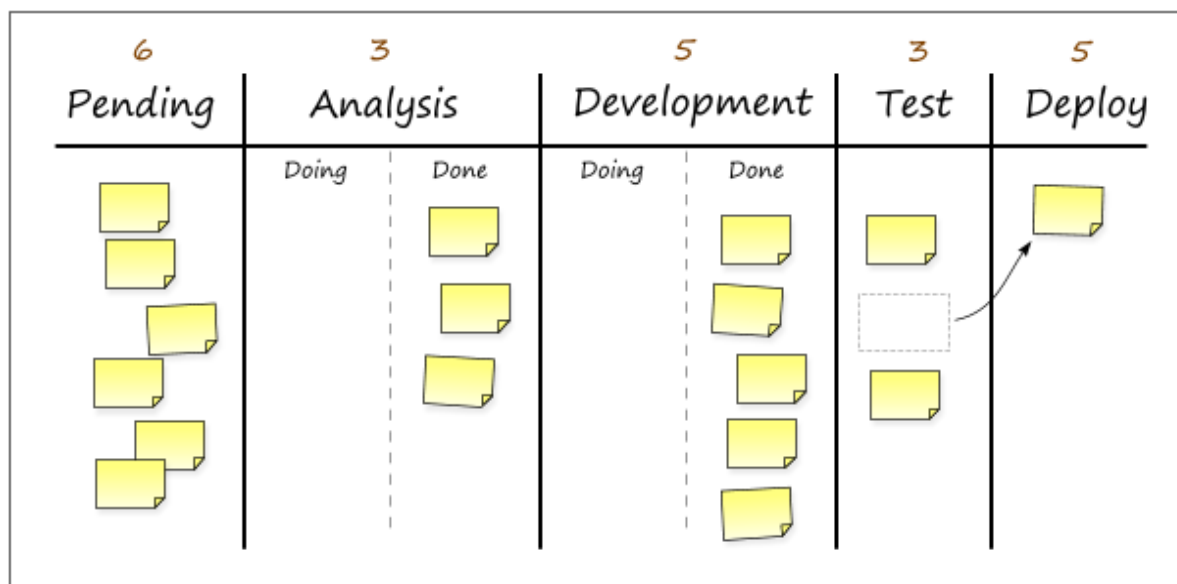
## 1.4 Kanban

### Kanban - Incremental Improvements

The Kanban methodology is way less structured than Scrum. It's no process framework at all, but a model for introducing change through incremental improvements. One can apply Kanban principles to any process one is already running.

In Kanban, one organizes the work on a Kanban board. The board has states as columns, which every work item passes through - from left to right. One pull work items along through the [*in progress*], [*testing*], [*ready for release*], and [*released*] columns (examples). And you may have various swim lanes - horizontal "*pipelines*" for different types of work.

The only management criteria introduced by Kanban is the so called "*Work In Progress*" or WIP. By managing WIP you can optimize flow of work items. Besides visualizing work on a Kanban board and monitoring WIP, nothing else needs to be changed to get started with Kanban.



## 1.5 Prerequisites : XP !

Agile methodologies leverage eXtreme Programming practices. A sound understanding of XP practices and their rigorous application is a mandatory prerequisite of Agile methodologies.

While some practices are applied sometimes a little differently in scrum, really all of them are important. XP Practices are really intended to be respected all together since they have interactions and one cannot benefit from the advantages of XP if

one's picking up only a subset of the practices.

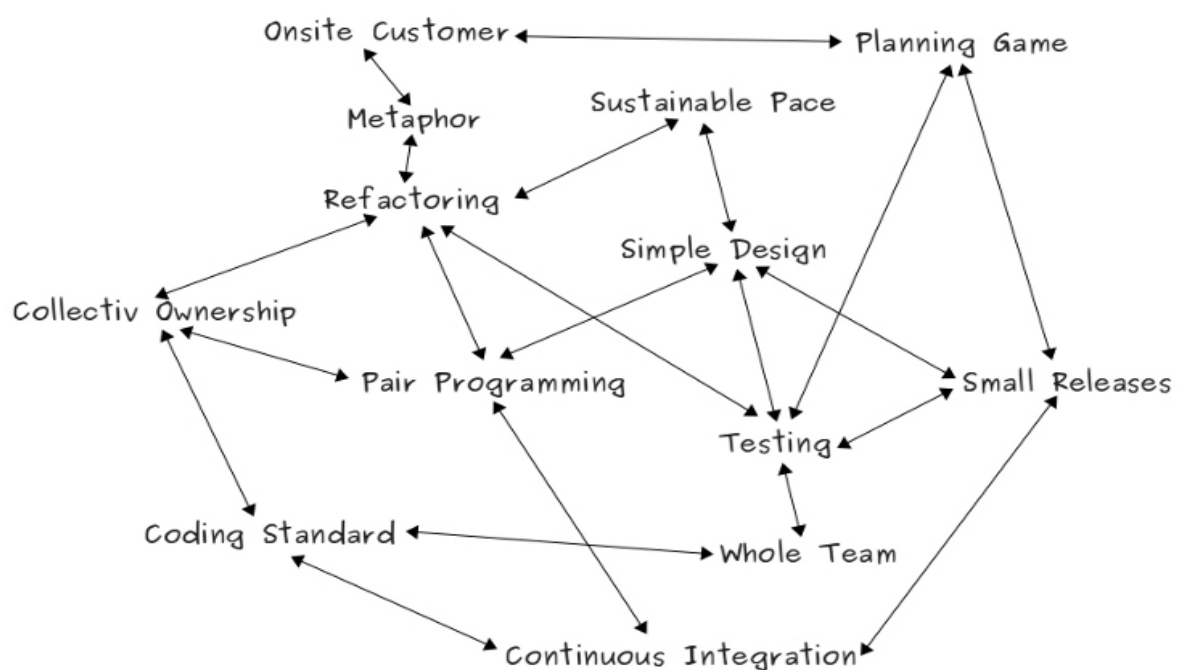
Some XP Practices should really be respected as described and advocated by XP :

- Metaphor
- Refactoring
- Simple Design
- TDD (Testing)
- Coding Standards
- Collective Ownership
- Continuous Integration

While some others take a specific form in Scrum :

- Onsite Customer → Product Owner and his everyday communications with stakeholders
- Sustainable Pace → Immutable and frozen Sprints
- Planning Game → Sprint planning
- Small Releases → Shippable product at the end of every sprint
- Whole Team → Daily Scrum

Interactions between XP practices can be represented this way:





I didn't mention *Pair programming* ... To be honest we do not apply it consistently in my team. While I do certainly encourage pair programming when some specific and complicated algorithm or design needs to be implemented, I do not insist on it and most of the time my developers work on their own.

From there, how can we ensure every single line of code is reviewed by at least a second pair of eyes ?

We do enforce **Code Review** as part of our testing process. I'll get back to that.

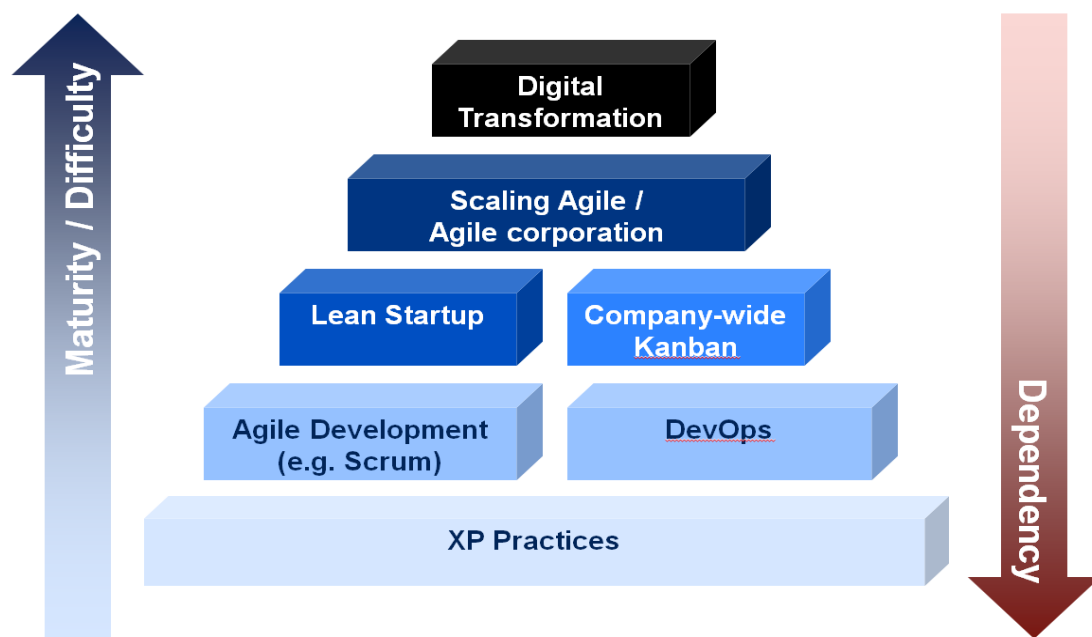
### 1.6 Benefits : DevOps, Lean Startup

Adoption of an *Agile Development Methodology* is the very ground on which many other practices or principles are built, should a Tech Company or an IT Department want to move forward towards more efficiency, shorter lead times, better reactivity and controlled costs.

To make it simple:

- Without a proper understanding and adoption of eXtreme Programming values, principles and practices, moving towards *Agile Software Development* will be difficult.
- Without Agility throughout the IT processes, both on the development side (Agile) and on the Production side (DevOps), trying Lean Startup practices and raising Agility above the IT Department will be difficult.
- Without a sound understanding of the Lean Startup Philosophy and practices and a company-wide Agile process (such as a company wide Kanban), transforming the company to an Agile Corporation will be difficult.
- Finally, only Agile Corporations can really imagine successfully achieving a Digital Transformation

This can be represented as follows:



As shown on the above pyramid, a good adoption of sound DevOps and Lean Startup practices itself is a prerequisite towards further transformations: from enterprise-scale Lean-Agile development to ultimate Digital transformation of the company. Well I guess developing these concepts should be the topic of another blog post.

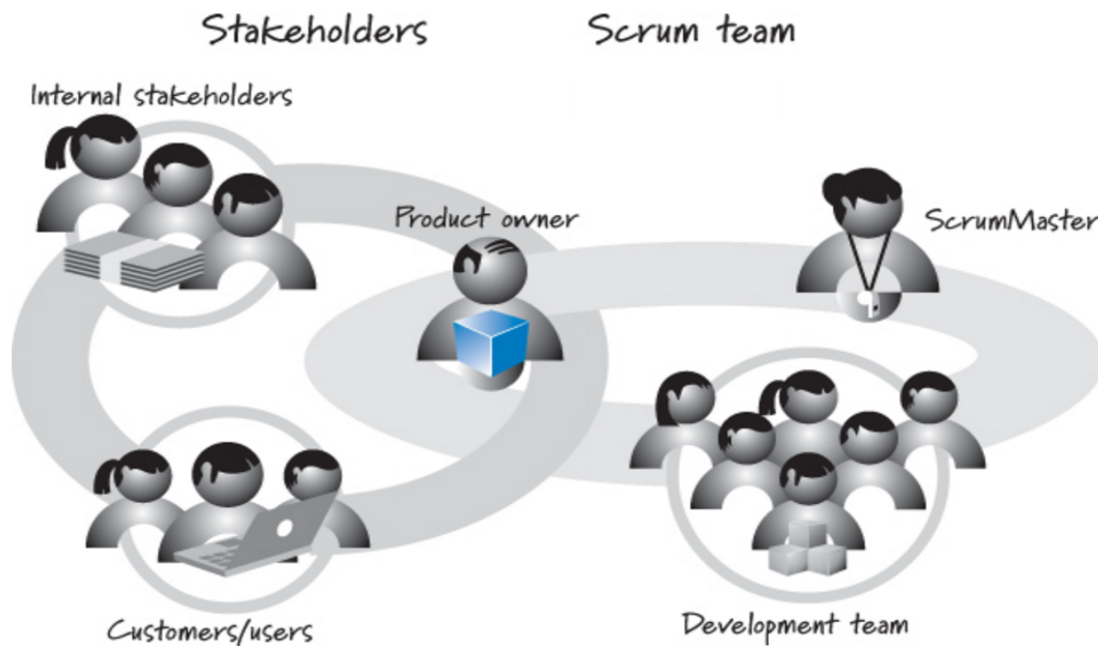
## 2. Scrum roles

Arguably, a very important role involved in Scrum is the **Stakeholder**, as the Stakeholders are the ones who have desires and needs, and are the reason the team is developing the software in the first place.

While the Stakeholders are the most important source of validation for the project, the most important person on the Scrum Team is the **Product Owner** (PO).

The Product Owner works with the Stakeholders, represents their interests to the team, and is the first person held accountable for the team's success. The Product Owner must find a result that will satisfy the Stakeholders' needs and desires.

The Product Owner provides direction and goals for the team, and prioritizes what will be done.



In my current company, the stakeholders are reunited in a formal **Product Management Committee** Meeting once per month and are weighted this way:

- Head of R&D - myself - 30%
- Head of Delivery - 30%
- Company founders and top executives - 20%
- Sales representatives - 20%

I myself, as Head of R&D, ensure the role of Product Owner for what the development team is concerned with. As a matter of fact, I ensure two roles : Product Owner and Lead Architect.

Regarding architecture, I rely on two technical leaders in my team who help my with this duty.

I would strongly recommend the reader takes 10 minutes to watch this Video from Henrik Kniberg presenting pretty clearly and completely the role of Product Owner and the functions of the Scrum Team around him : [Agile Product Ownership in a Nutshell - VOSTFR](#)

The **Scrum Master** is an important role as well. The Scrum Master serves as a facilitator for both the Product Owner and the team. The Scrum Master has no authority within the team (thus couldn't also be the Product Owner!) and may never commit to work on behalf of the team. Likewise, the Scrum Master also is not a coordinator, because (by definition) self-organizing teams should co-ordinate directly with other teams, departments, and other external entities.

The Scrum Master removes any impediments that obstruct a team's pursuit of its sprint goals. If developers don't have a good sense of what each other are doing, the Scrum Master helps them set up a physical taskboard and shows the team how to use it. If developers aren't collocated, the Scrum Master ensures that they have team room. If outsiders interrupt the team, the Scrum Master redirects them to the Product Owner.

In my current company, the development team is a single team spread among two locations. Half of the team is in Switzerland and the other half is a near shore team. I have selected 2 persons, one in each location, to ensure the scrum master role in both locations.

I define their duties this way : they take care of making sure the team as a whole sticks to the scrum process, raise warnings when I myself tend to compromise it, facilitate communication issues between both locations, etc.

Having a dedicated scrum master in both locations is utmost important since communication issues tend to be amplified by remoting.

### 3. From Story Maps to Product Backlog

---

As an Agile team and increasingly an Agile Company, we strongly emphasizes Visual Management Tools.

I think this is one of great strength in my current company: our understanding of *Lean management* practices and the way we inspire our everyday rituals by the *Kaizen* method : we try to learn and improve everyday, learn from our mistakes, leverage on our strengths. We have weekly rituals where we discuss our processes, the issues we encounter and we are agile in every way on the whole line, we adapt the way we work, communicate and collaborate continuously to what we learn.

In my opinion, one of the most important aspects of Lean management is Visual Management.

In this regards, I will focus in this article on two very important tools : **The Story Map** and **The Kanban Board**.

#### 3.1 User Stories

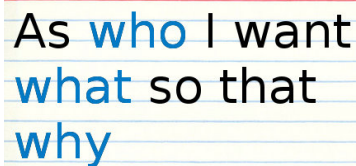
User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

They typically follow a simple template:

*As a <type of user>, I want <some goal> so that <some reason>.*

User stories are often written on sticky notes and arranged on walls or tables to facilitate planning and discussion.

As such, they strongly shift the focus from writing about features to discussing them. In fact, these discussions are more important than whatever text is written.

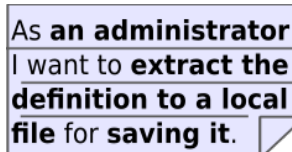


As **who** I want  
**what** so that  
**why**

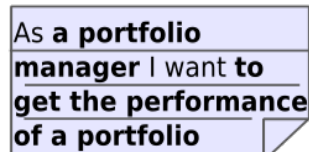
It's the product owner's responsibility to make sure a product backlog of agile user stories exists, but that doesn't mean that the product owner is the one who writes them. Over the course of a good agile project, you should expect to have user story examples written by each team member.

Also, note that who writes a user story is far less important than who is involved in the discussions of it.

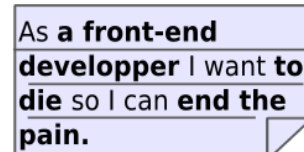
Some example stories for different application contexts:



As **an administrator**  
I want to **extract the**  
**definition to a local**  
**file** for **saving it.**



As **a portfolio**  
**manager** I want to  
**get the performance**  
**of a portfolio**



As **a front-end**  
**developer** I want to  
**die** so I can **end the**  
**pain.**

Agile projects, especially Scrum ones, use a product backlog, which is a prioritized list of the functionality to be developed in a product or service. Although product backlog items can be whatever the team desires, user stories have emerged as the best and most popular form of product backlog items.

## 3.2 Story Maps

User stories, when converted as Developer tasks in the product backlog, should be very finely defined and well documented.

But at the time of designing a product or an evolution, brainstorming around the functionalities requires some abstraction and to remain at a very high functional level.

It makes no sense at this initial stage to design fine and well documented tasks. One should rather focus on identifying high level user stories covering each and every expected functionality of the new software or evolution.

This is the purpose of the **Story Mapping** workshop. This is typically a few days workshop organized as max 4 hours sessions per day where **all the stakeholders** (this is important) take the time to sit in a room together and define the product with the help of User Stories.





## Product Vision

But everything should really start by the definition of **Product Vision**. The first session of a set of Story Mapping workshops should be the *Vision Workshop* where everyone first agrees on a common 2 to 3 years vision of the *Experience Users* will have with the Product (or evolution, new feature, whatever).

Defining and agreeing on a vision is important since the vision:

- drives product decision - The vision should be complete and clear enough to act as a reference one should be able to turn to in case of doubts regarding where to move the product forward.
- provides a destination for the team to stay on course
- gets the entire team on the same page
- *inspires and motivates*
- aligns roadmap and sprint investments with user needs and business goals
- importantly : enables the product team to say "NO!" to features that don't align with the vision.

The result of the product vision workshop should fit on a one page vision map, such as for instance:

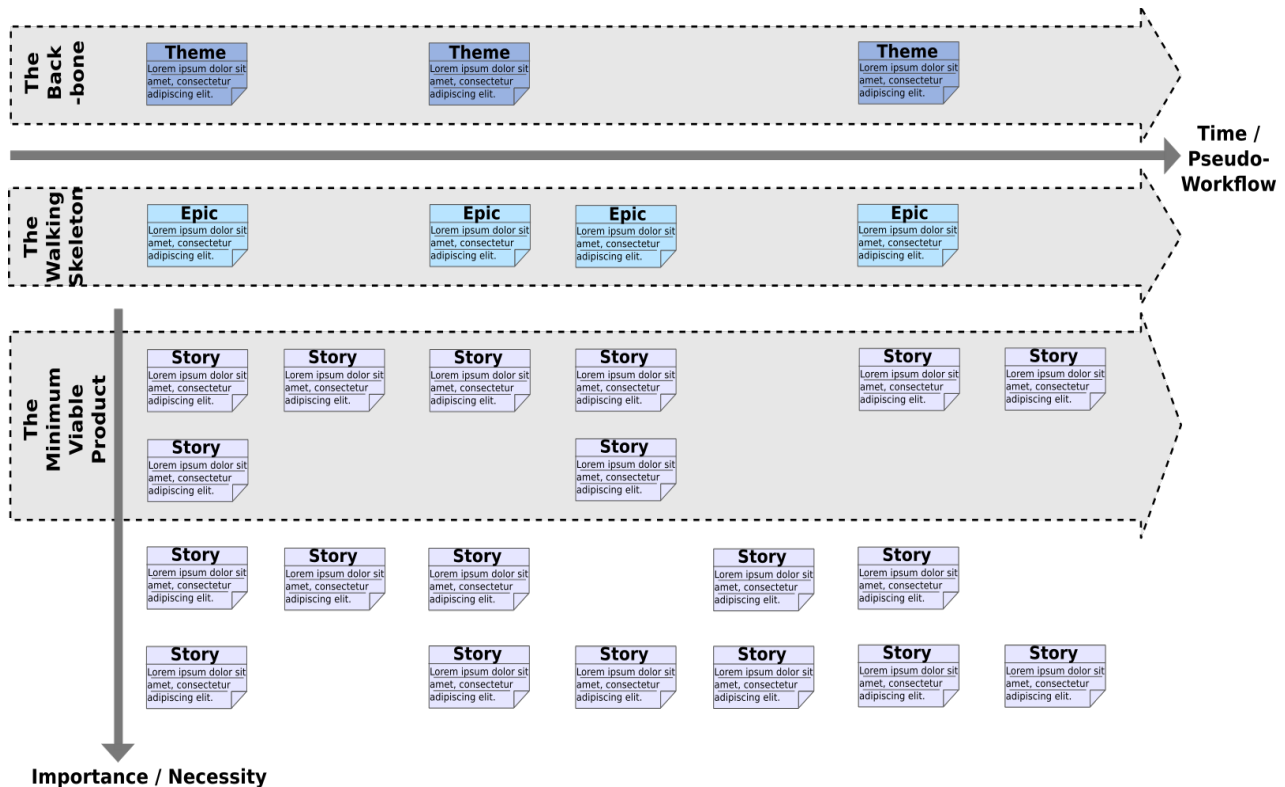
 Vision Statement         Develop a digital product canvas to help teams create great products			
 Target group users: Product managers and product owners customers: Mid-size to large enterprises	 Needs Have an effective tool for creating ux-rich products while taking advantage of Greentopper Leverage the existing investment; minimise the cost of acquiring a new tool	 Product Tablet app; data is held in Greentopper Looks like a physical canvas; intuitive to use Provides guidance and templates	 Value Open up a new revenue stream Develop our brands and reputation

## Story Map

With this first step achieved, and a proper vision define, the real job, defining User Stories and laying them down on the **Story Map** can start. This is usually done in a few session of 4 hours max.

The purpose of the Story Map is that arranging user stories into a helpful shape - a map - is usually deemed as most appropriate.

A small story map might look something like this:



At the top of the map are "big stories." We call them **themes**. A theme is sort of a big thing that people do - something that has lots of steps, and doesn't always have a precise workflow. A theme is a big category containing actual user stories grouped in **Epics**.

Epics are big user stories such as the one mentioned in example above. They usually involve a lot of development and cannot be considered as is in an actual product backlog. For this reason, Epics are split in a sub-set of **stories**, more precise and concrete that are candidate to be put in an actual product backlog.

The big things on the top of the story map look a little like vertebrae. And the cards hanging down look a little like ribs. Those big things on the top are often the essential capabilities the system needs to have.

We refer to them as the "**backbone**" of the software.

**The Walking Skeleton** is composed by the epics of the software. The Walking skeleton is a refinement of the backbone, composed by epics taking the form of user stories, in different with themes that are rather very high level titles or sometimes even simple words.

When it comes time to prioritize stories, we don't prioritize the backbone or the walking skeleton. We do prioritize the ribs - the stories hanging down from the backbone. We place them high to indicate they're absolutely necessary, lower to indicate they're less necessary.

By doing this, we find that all the stories placed high on the story map describe the smallest possible system you could build that would give you end to end functionality. This is what *Lean Startup* calls the **Minimum Viable Product**.

A *Minimum Viable Product* has just those core features sufficient to deploy the product, and no more. Developers typically deploy the product to a subset of possible customers—such as early adopters thought to be more forgiving, more likely to give feedback, and able to grasp a product vision from an early prototype or marketing information.

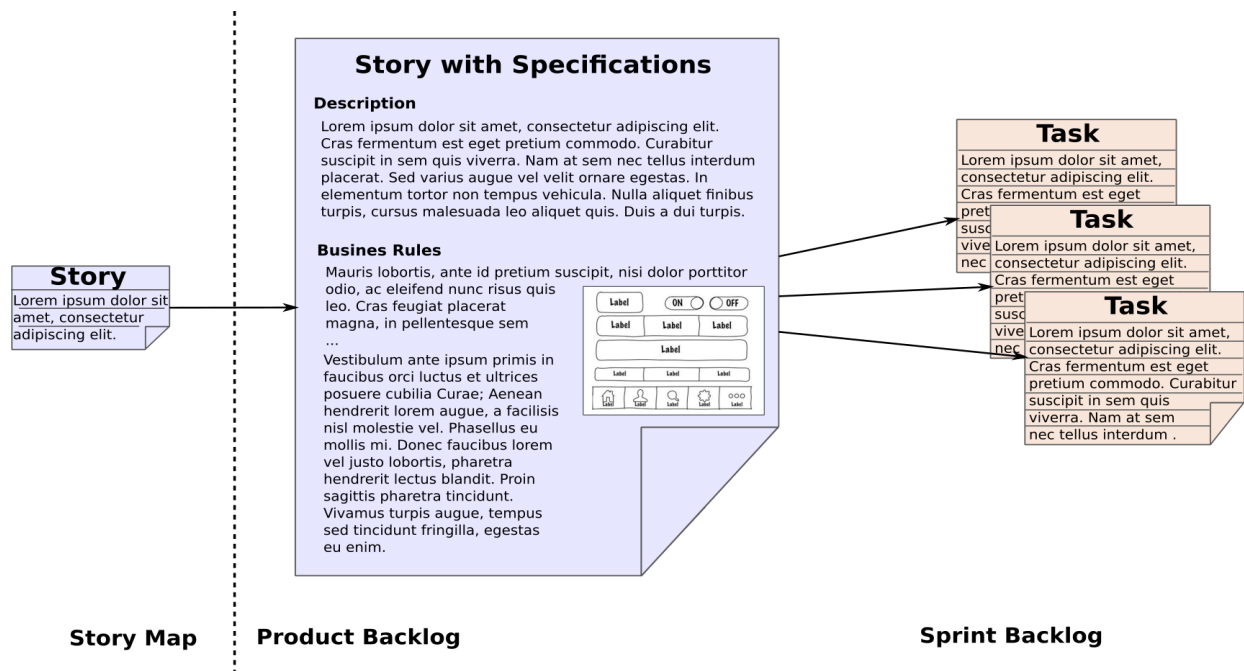
This strategy targets avoiding building products that customers do not want and seeks to maximize information about the customer per dollar spent. **The Minimum Viable Product is that version of a new product a team uses to collect the maximum amount of validated learning about customers with the least effort.**

### 3.3 From User stories to Developer Tasks

While a product backlog can be thought of as a replacement for the requirements document of a traditional project, it is important to remember that the written part of an agile user story ("As a user, I want ...") is incomplete until the discussions about that story occur.

It's often best to think of the written part as a pointer to the real requirement. User stories could point to a diagram depicting a workflow, a spreadsheet showing how to perform a calculation, or any other artifact the product owner or team desires.





The simplest way to state this is as follows :

**User Stories are what users do to reach their goals.**

**Developer tasks are what developers do to implement user stories.**

Transforming a *User Story* to *Story with Specification* has to be done by the Product Owner and the Technical Architect of the platform (so two times myself in our case). The Product Owner may need to get in touch with the stakeholders to get some precisions in case there are doubts in regards to the *User Experience* to be presented to the end users.

The Story with specification should contain, at least, in a non-exhaustive way :

- The initial user story and all that was expressed at that time
- A complete description of the purpose of the feature
- A complete description of the expected behaviour from all perspectives : user, system, etc.
- Mock-ups of screens and front-end behaviours as well as validations to be performed on the front-end
- A list and description of all business rules
- A list and description of the data to be manipulated
- Several examples of source data or actions and expected results
- A complete testing procedure

Then, the *Story with specification* is decomposed in *Developer Tasks* either in advance by the Architect of the platform (well myself again) or at the latest by the whole team during the *Sprint Planing* meeting.

## 4. From User Stories to Releases

---

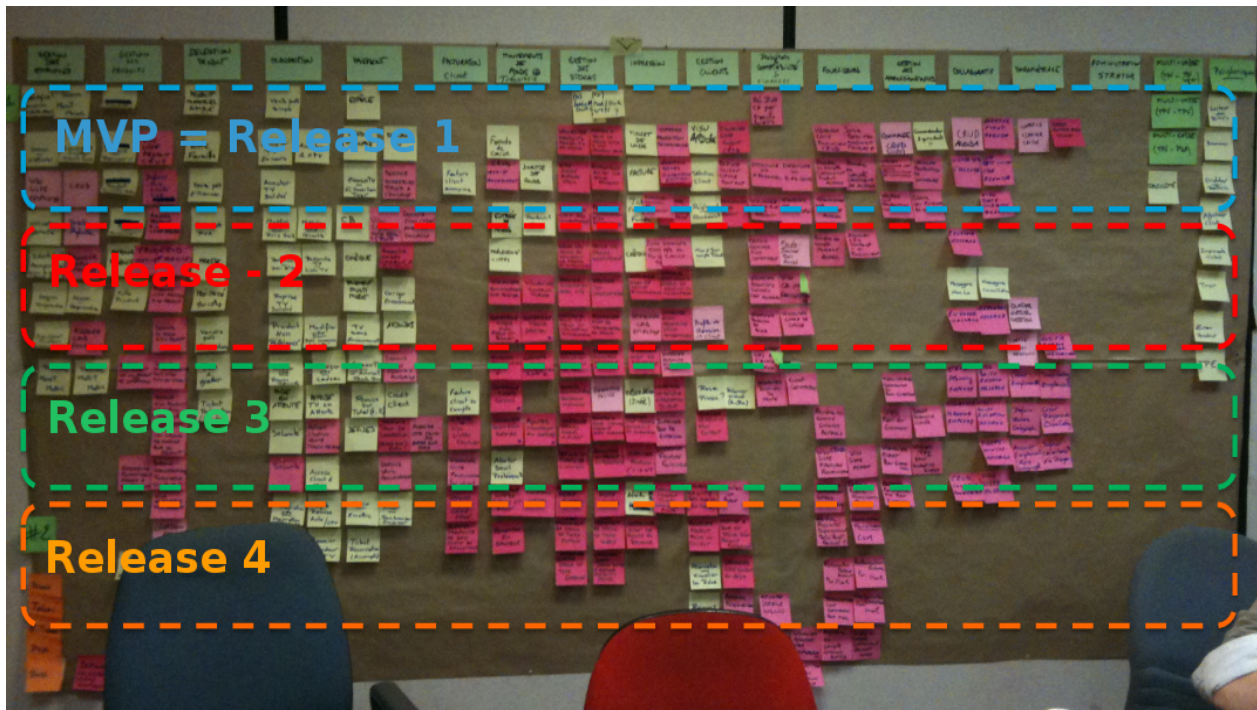
We find a story map hung as an information radiator becomes a constant point of discussion about the product we're building. When the project is running, it becomes our sprint or iteration planning board. We identify or mark off stories to build in the next iteration directly on the map. During the iteration we'll place just the stories we're working on into a task wall to managing their development - but the story map lives on the planning wall reminding us what the big picture is, and how far we've come.

When we're building software incrementally, story by story, we'll choose them from the story map left to right, and top to bottom. We'll slowly move across the backbone, and down through the priorities of each rib. We're slowly building up the system not a feature at a time, but rather by building up all major features a little at a time. That way we never release a car without brakes.

### 4.1 Composing our releases

With the help of the story map and a clear classification of our User Stories in terms of importance and priority. We try to plan for our releases, grouping together features that require to be delivered consistently.

Grouping these feature together is usually done in a another workshop that we call the **roadmap workshop**. When we have identified a set of stories that definitely belong together, we group them horizontally so that we can identify releases by horizontal boxes, for instance as follows:



Among these releases, the *Minimum Viable Product* release is the most important one. A great care should be taken when composing this release to respect the definition of the *Minium Viable Product* indicated above.

One should note, we really use the story map releases as initial plan. In practice, real releases differ a lot from our plans, always. We more or less release when some features we were working on become urgently required for a customer.

Long story short, the release we plan initially give us a long term vision, a direction. But reality differ a lot and we do usually many more releases that what we planned initially.

Once we know what our release will be composed for, we're left with composing our sprints.

Then, we release either because we have reached what we initially intended to be part of the release, but that never happens in practice. In reality, we release more often, simply when a set of features implemented in a sprint are required by a customer.

Since every sprint finishes with a shippable, production-ready product, this is perfectly fine. We'll get back to this at the end of this paper.

## 4.2 Composing the sprint

The priorities coming from the release planning on the story map of its vertical scale are too coarse-grained to be used to prioritize tasks when composing the next sprint. We need a better way to fine tune the task priorities when stories are split to tasks in

the product backlog.

Since we are using redmine to manage our tasks and sprints - along with some redmine plugins for Agile projects - we make use of the redmine notion of **priority** for this concerns. When a task coming from a story is put in the backlog, it inherits from the priority of the user story induced by the position of the story on the Story Map. This is a notion of *initial priority*.

Later, as Product Owner, when I have all my tasks for a given release in the backlog, I change priorities for much finer notions by still respecting the workflow induced by the Story Map.

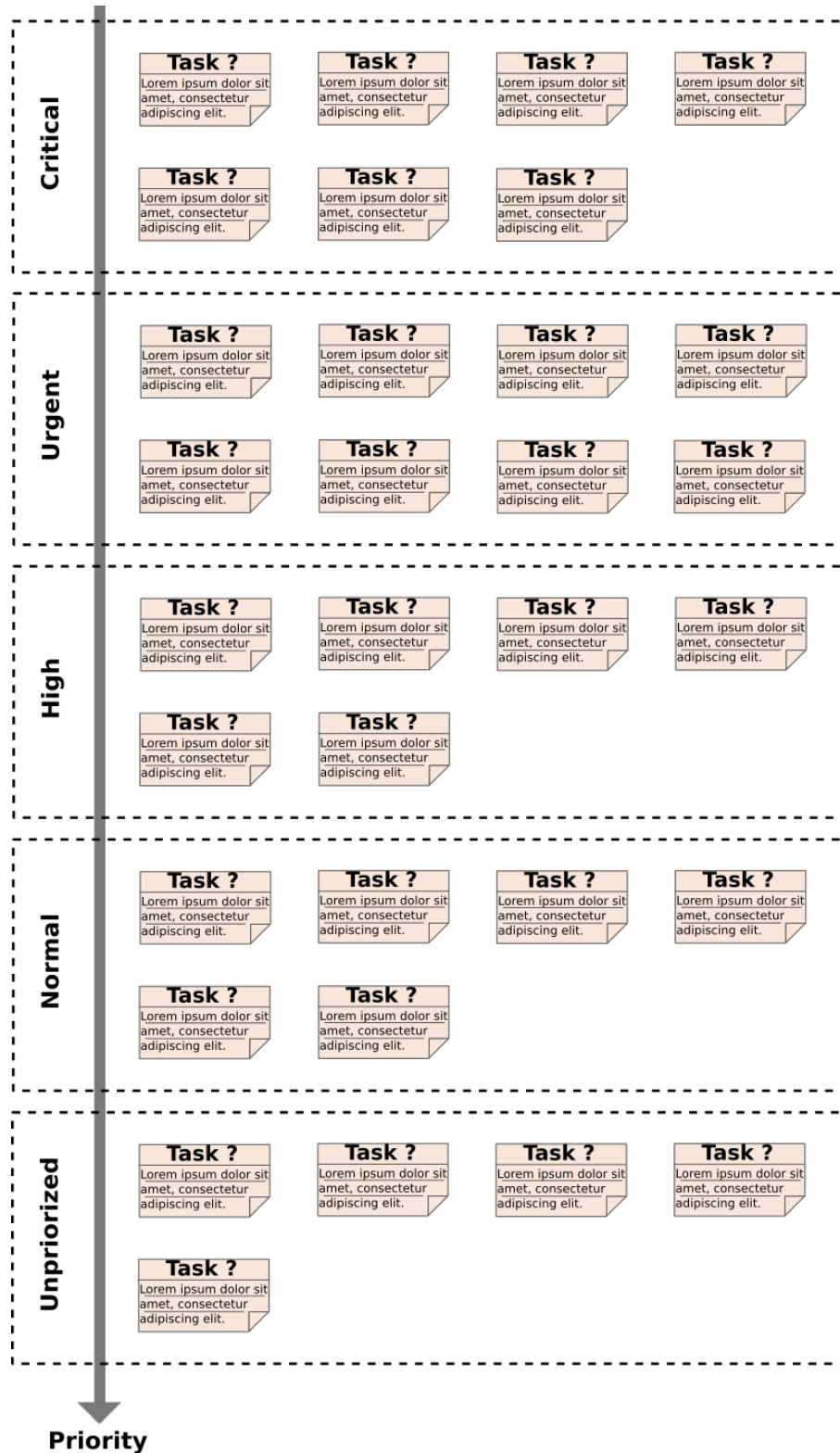
In addition, we use task priorities in a somewhat specific way for Internal R&D Organization to have a way to select task when composing our sprints.

We classify priorities depending on the moment of the release development when we want to implement them - high priority tasks are implemented first - Normal tasks are implemented last, etc. This is a principle. The *Urgent* priority is reserved for R&D for a specific purpose:

- **Urgent** tasks are the candidates to be picked up in the next sprint, and only them
- Whenever we are out of Urgent tasks, an election process is run and tasks in High priority are elected to Urgent. This way they become candidate that can be picked up in next sprint

Backlog priorities between [**high, normal, low, unprioritized**] are set in good understanding with PMC (Product Management Committee)

[**urgent**] priority is reserved for R&D only to define candidates to be taken in the next sprint.



The process is iterative : when we are out of *Urgent* tasks, we re-prioritize the backlog again and elect some new *urgent* tasks from the *high* tasks or even lower

priorities. We keep doing that over and over again until we have no more tasks of lesser priorities and the set of urgent tasks can be finished in 1 or 2 sprints.

### 4.3 Estimations in Story Points

In waterfall, managers determine a team member's workload capacity in terms of time. Managers ask selected developers to estimate how long they anticipate certain tasks will take and then assign work based on that team member's total available time. In waterfall, tests are done after coding by specific job titles rather than written in conjunction with the code.

The downsides of waterfall are well known: work is always late, there are always quality problems, some people are always waiting for other people, and there's always a last minute crunch to meet the deadline. Scrum teams take a radically different approach.

- First of all, entire Scrum teams, rather than individuals, take on the work. The whole team is responsible for each Product Backlog Item. The whole team is responsible for a tested product. There's no "my work" vs. "your work." So we focus on collective effort per Product Backlog Item rather than individual effort per task.
- Second, Scrum teams prefer to compare items to each other, or estimate them in relative units rather than absolute time units. **Ultimately this produces better forecasts.**
- Thirdly, Scrum teams break customer-visible requirements into the smallest possible stories, reducing risk dramatically. When there's too much work for 7 people, we organize into feature teams to eliminate dependencies.

#### Planning Poker

**Planning poker**, also called Scrum poker, is a consensus-based, gamified technique for estimating, mostly used to estimate effort or relative size of development goals in software development.

In planning poker, members of the group make estimates by playing numbered cards face-down to the table, instead of speaking them aloud. The cards are revealed, and the estimates are then discussed. By hiding the figures in this way, the group can avoid the cognitive bias of anchoring, where the first number spoken aloud sets a precedent for subsequent estimates.

The cards in the deck have numbers on them. A typical deck has cards showing the Fibonacci sequence including a zero: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89; other decks use similar progressions.





The reason to use planning poker is to avoid the influence of the other participants. If a number is spoken, it can sound like a suggestion and influence the other participants' sizing. Planning poker should force people to think independently and propose their numbers simultaneously. This is accomplished by requiring that all team members disclose their estimates simultaneously. Individuals show their cards at once, inspiring the term "*planning poker*."

In Scrum these numbers are called **Story Points** - or **SP**.

### What does the process of estimation look like?

The actual estimations in SP, those on which the development team as whole commits and agrees are set during the *Sprint Planing* meeting, all together. This is the only way. No one single person can decide of the estimation in SP of any given task.

However, a way to estimate the whole workload of a release backlog or even a long-term backlog is required.

This is the reason why, the Product Owner and The Architect meet once in a while - not in our case since I ensure both roles - to provide an **initial estimation** in SP on all the tasks and stories (or even epics) in the release and long term backlogs. These are initial estimations having as only purpose the need to estimate workloads of future releases. before the sprint planning meeting occurs, these initial estimations are removed in order not to influence the development team who will have to provide the real estimations.

### Filling the sprint

During the **Sprint Planing** meeting, we take the *developer tasks* with the highest priority from the next release backlog and put them in the sprint backlog (these different backlogs are introduced below).

The whole team gathers in a room and takes all tasks sorted by priority, evaluates them - gives them an estimation in SP - discusses all their aspects, makes sure they're crystal clear to everyone and finally moves them to the sprint backlog.

The question is : when should we stop? When do we have enough tasks in the sprint backlog to form the next sprint?

And the answer is simple : we stop when the set of tasks in the sprint backlog have a sum of SP that match the **Team Capacity**, also expressed in SP.

The *Team Capacity* is computed by taking the average sum of SP implemented in a set of first sprints, when the team is constituted or when new engineers join it. It takes a few sprints to be able to compute a relevant average. As a sidenote, funnily enough, in my current company, the *capacity* of the development team is precisely (i mean precisely !) in its current form.

During these first sprints, when the team capacity is unknown, I believe the simplest way is to start with an empty sprint backlog and simply let developers take tasks from the release backlog, moving every task to the sprint backlog before working on them. But others have other ideas ...

## 5. Introducing our sprints

---

We run sprints of two weeks in my current company. Two weeks is really what works the best in our setup. One week would obviously be too short and three weeks would make it impossible to close all the tasks in one single *Testing Friday* (see below). In addition, in a continuous delivery approach (or I should rather say, as close to Continuous Delivery as we can get), more than two weeks between deliveries would be too much.

We do not necessarily stick 100% to the scrum process in the sense that we accept urgent tasks in the sprint even after it has been started. The only reason for that is urgent fixes required in production cannot wait more than a few hours. Urgent production issues are the single and only reason we accept to change a little the scope of our sprints even after they have started.

This requires some arbitration : a production issue at a customer needs to be qualified as urgent to make it to the current sprint, otherwise it goes in the next sprint.

We have formal rituals at the beginning of the sprint and at the end of the sprint.

### 5.1 Before Sprint

Before the sprint, The Head of R&D, a.k.a myself, estimates new tasks by himself. These are so called *initial estimations* as indicated above.

In addition, the monday of a new sprint I take care of all the technical concerns around the sprint (sprint creation on redmine, closing tasks, closing former sprint, etc.).

#### Sprint Planning Meeting

During the Sprint Planning meeting :

- We re-estimate the tasks and come up with **actual estimations**



- We feed the sprint backlog with urgent tasks that have an estimation set in terms of SP. Only these tasks are valid candidates to be put in the sprint backlog.
- We feed the backlog in order to have a total amount of SP corresponding to our average capacity

## 5.2 During Sprint

Every developer is free to pick up any task he wants from the Sprint backlog when he is done with his former task. I myself try to intervene as little as possible in regards to who works on what. This works only if developers are responsible and autonomous. Another team may require more involvement from the team leader in regards to tasks assignment.

Again, we try as much as possible to avoid changing the sprint scope during the sprint. However this might happen. In this case, whenever we have to add some very urgent tasks in the backlog during the sprint to work on it automatically, we respect the following principle :

- We first estimate this task all together after daily scrum
- We put the task in the backlog and remove as many other tasks it is required to remove to leave the total amount in terms of SP of the sprint unchanged

Should it happen that a developer himself takes a task from the release backlog and puts it in the sprint, he needs to remove the initial estimations put by head of R&D on tasks in release backlog so that we remember estimating it at next daily sprint.

### At the end of the sprint : Testing Friday

Every last Friday of each sprint is **Testing Friday**.

At testing Friday, every developer at R&D turns into a tester. Everyone in R&D tests former sprint(s) tasks and closed the subtasks. Tasks themselves are closed by Product Owner, a.k.a Head of R&D, a.k.a Myself (or delegate), not by R&D engineers directly.

Having developers turning into testers that day and testing each other tasks themselves, as opposed to having a dedicated team of testers, actually has a purpose.

I cannot stress enough how much I believe this is important. Having developers tuning to testers a day per sprint and testing each-other's tasks really helps them get a sense of responsibility. Being pissed off when write didn't test his task before moving it to *Done* (or *Testing Ready* in our case) makes one test thoroughly his own tasks before passing them forward. Struggling to understand how to test something makes one document in details the test procedure on his own tasks. Etc.

Testing a task means :

- Running the documented testing procedure and ensuring everything works as expected
- Reading the code and ensuring it matches the code and quality standards
- Testing the feature in other than expected conditions
- Assessing non-functional behaviours such as performance and robustness

### 5.3 After Sprint

After the sprint, we have two important sprint rituals : The **Sprint retrospective** meeting and the **Sprint Demo** Meeting.

In addition, we take care of releasing the Demo VM of our platform. The Demo VM is a virtual appliance that integrates each and every individual software component of our platform and configures them with some default configuration plus feeds them with test data.

The Demo VM building scripts leave us with a Demo platform ready to be used, just as if it was integrated at a customer by our teams of consultants.

The Demo VM building is completely automated and its a step towards **continuous delivery** for us in the form of a **continuous deployment**. At the end of every successful integration build, it is automagically built and deployed on a test virtual server, enabling immediate feedback form our stakeholders.

At the end of a sprint, we release the latest built Demo VM for wide usage by our sales representatives, consultants, etc.

At the end of the sprint, we also take great care of ensuring before building the last Demo VM that all tests are passing:

- Unit tests (Commit Build)
- Integration tests (Nightly build)
- End-to-end UI tests (Selenium tests on Demo VM building)

The closer we get to the end of the sprint, the more carefully we monitor these builds to ensure we don't have any issue building the Demo VM at the end of the sprint.

#### **Sprint retrospective meeting**

We mostly do these things during the *Sprint Retrospective*:

- We review the few tasks that may not have been completely implemented and that need to be partially postponed to the next sprint
- We compute the amount of SP done

- We discuss issues and things to be changed and create tasks for them (refactorings, new unit / integration tests to be written, etc.)
- We discuss about issues encountered in the way the sprint was managed and search for opportunities to improve

### **Sprint Demo Meeting**

We mostly do 2 things during the *Sprint Demo*:

- We demonstrate new functionalities to our internal user representatives
- We take minutes of the meeting in the form of new tasks added to the backlog or updates to existing tasks

## **6. Release Backlog and Sprint Backlog**

---

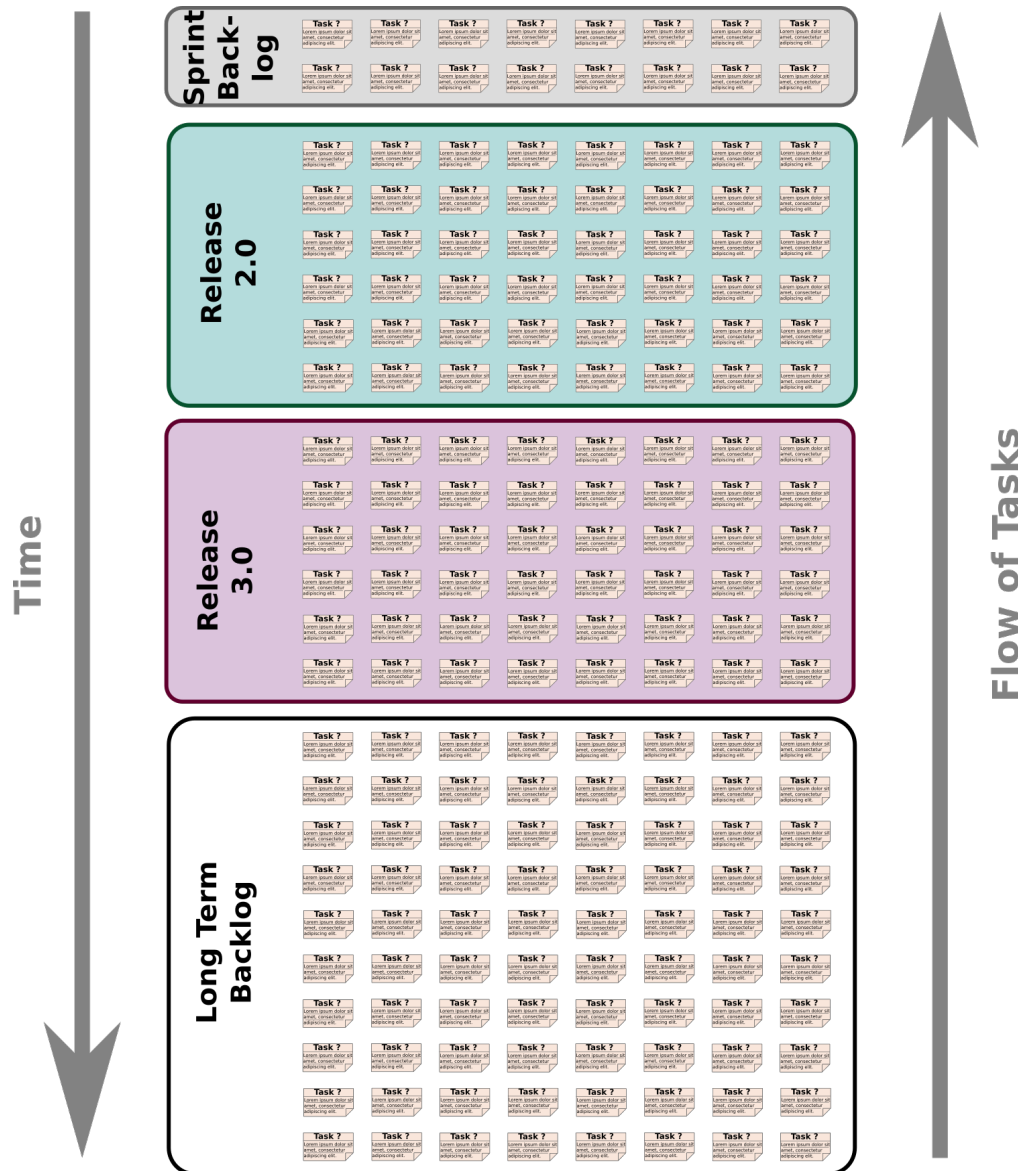
Using redmine, we attach our tasks to releases. We define as many releases as we have planned on our roadmap. Redmine then presents us the tasks grouped by releases first, and then all tasks that have no releases defined, we call this last backlog the long term backlog.

In addition, tasks attached to a version - which we use to identify sprints - are also displayed in a specific backlog.

In the end, it's really as if redmine presents us with different backlogs.

### **6.1 Different release backlogs, long term backlog, sprint backlog ...**

We use all this different backlogs as follows.



## Sprint Backlog

The **sprint backlog** identifies the tasks the development team are going to work on / are working on in the current (or past) sprints. The sprint backlog is "*locked*" at the end of the sprint and "*closed*" whenever each and every of its tasks are closed.

The Sprint backlog is the **Immediate-term backlog**, i.e. things we'll close in the 2 coming weeks.

## Release backlogs

We have as many **release backlogs** as released planned. All the tasks that are not assigned to a specific planned release are part of the **Long Term backlog**. Such tasks are typically assigned a release when the former releases are done and closed and we plan the next releases.

The release backlogs are the **Short-term backlogs**, i.e things we'll close in the coming months.

### Long-term backlog

The long term backlog is composed by tasks of a lesser priority. Those that make sense and we definitely believe we should work on them, those that corresponds to stories of the StoryMap or elements of the Roadmap, but that are not planned for any nearby delivery or for which we haven't identified any customer requirement so far.

## 6.2 While being Agile

Now all of the above form a plan, gives us an objective and a direction. Having a plan, keeping a direction is important. It enables the whole company to agree and commit on a vision and business directions.

Yet we are agile, we stick to our vision, but we adapt the plan continuously. The composition of our releases, our ideas, the priorities of the tasks and the way we intend to group them in releases change all the time, almost every week to be honest.

We already know what we are going to release and when we will release it for the coming 18 months. But in one year, it is absolutely certain that we will have done it completely differently.

Because we adapt to market events and feedback and to customer requirements.

At the end of the day this is no big deal and has really only little importance. **We are Agile**, meaning **every sprint is closed by a production ready and shippable version of our platform**.

Taking the decision to assign a version number to these releases and roll them out in production at some customer is a Product Management Decision, it's almost not anymore a development concern.

At the end of the day, we have really only one single constraint to make it possible : split big refactorings (technical epics) or large business epics in smaller tasks that have to fit in a single sprint, whatever happens.

These tasks have to be completed by the end of the sprint, meaning being properly closed, tested and 100% working.

For instance:

- Imagine we have to realize an important refactoring that would need weeks of development to be completed. That refactoring is split in small tasks such as [1. Put in place the framework], [2. Implement it in this package], [3. Implement it in this other package, etc].

At the end of one sprint, it may well happen that the refactoring is only

partially realized. In this case we make it so that the platform works perfectly even if half of the code is running on the former approach and only the other half of it has been migrated to the new way.

- As another example, imagine a brand new feature requires several dozens of new screens which would take weeks to implement. Thanks to **Feature Flipping**, we simply disable this feature in production while it is still in development. The day we finally finish to implement it in a sprint, that end-of-sprint release will finally have the feature enabled and make it available to our customer.

Long story short, we make it so that partially implemented features are either working 100% from a functional perspective or properly hidden, in order not to compromise User Experience on the platform

Again, all of that is possible because we have embraced eXtreme programming, Agile as well as some DevOps and Lean Startup practices as our core set of practices.

### 6.3 Handling customer requests and production concerns

Now all the above works great on the paper or when we develop a brand new product, a completely new feature or technological evolution.

But it doesn't handle urgent customer requests or production concerns such as bug fixes or other urgent and new requirements coming from our consultants or customers.

We have a special way of handling such new features or bug fixes which takes the form of a special tracker named **wish**.

These wishes are put in a special backlog, the wish backlog which is reviewed every month by the Head of R&D (myself) and the Head of Delivery. Together, we discuss these *wishes* define priorities and transform them into actual development tasks put in one of the backlog above, sometimes as close as the next *Sprint backlog*.

### 6.4 Sprint Kanban backlog management

While the **Burndown chart** is interesting to track the performance in comparison to theory as well as delays or advance, I don't find it so useful in the end and really seldomly use it.

I mean, it's interesting to have a clear visual indication of how the sprint is going and where we stand in comparison with expected status of course, but it's too general. While I can look at it once in a while to confirm a feeling of being late or in advance I might have, it's really not the tool I'm using.

As a manager with a good understanding of what we need to do in every sprint as well as the dependencies between tasks and the overall context of the sprint whereabouts, I need a tool providing me a fleeting glimpse on every task's status

and the overall situation of the Sprint.

In this regards, a Kanban board is to me the "*one ring to rule them all*" tool.

We use redmine and the Agile plugin of redmine to manage our **Kanban boards**.

That plugin works is a pretty specific way : the Kanban board shows tasks (and other items) on the left and sub-tasks (in the redmine terminology) are the elements passed between states. This is illustrated in the example below.

In our case, we manage *stories* and *tasks* in the backlog. **As task never ever has any subtask other than one entitled "*Implementation of #1234*"** where 1234 is the ID of the parent task it belongs to.

That only "*Implementation*" subtask is the visual artifice we use to track the status of our tasks on the Kanban board:

Sprint Tasks	New	Analysis	Implement- tation	Testing Ready	Testing	Closed
<b>Task 1</b> Lorem ipsum dolor sit amet, consectetur adipiscing elit.		Implementation Sub-Task 1				
<b>Task 2</b> Lorem ipsum dolor sit amet, consectetur adipiscing elit.	Implementation Sub-Task 2					
<b>Task 3</b> Lorem ipsum dolor sit amet, consectetur adipiscing elit.			Implementation Sub-Task 3			
<b>Task 4</b> Lorem ipsum dolor sit amet, consectetur adipiscing elit.			Implementation Sub-Task 4			
<b>Task 5</b> Lorem ipsum dolor sit amet, consectetur adipiscing elit.						Implementation Sub-Task 5
<b>Task 6</b> Lorem ipsum dolor sit amet, consectetur adipiscing elit.				Implementation Sub-Task 6		
<b>Task 7</b> Lorem ipsum dolor sit amet, consectetur adipiscing elit.					Implementation Sub-Task 7	

Tasks have a *release* assigned when they are in release backlogs (not those in long-term backlog). When a task is picked up during *Sprint planning meeting*, it gets in addition a *Target Version* assigned. We use redmine's notion of *Target version* to materialize our sprints.

### Tasks assignment

A task is always assigned to the developer who did most work on it. It then stays assigned to that developer forever. Should another developer take the task on, and both agree that that second developer ended up doing more work, he can assign the task to himself.

The subtasks (Implementation of ...) can be assigned to different developers when it is passed from a developer to another one.

A In this case, the assignee of the main task (the parent) remains the developer who did most work on it.

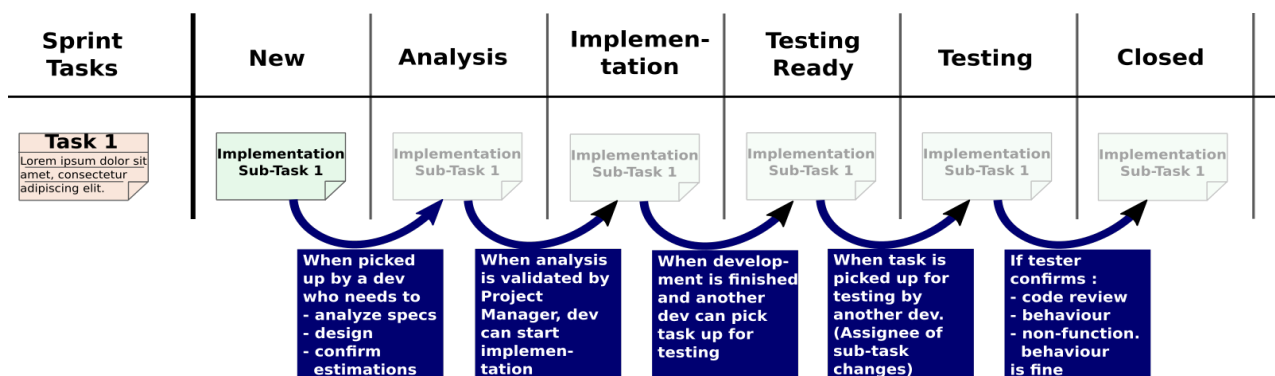
In addition, when a tester takes a subtask from "**Testing Ready**" and puts in in "**Testing**", he needs to assign the subtask to himself.

## Tasks Status

The rules are simple:

- The main task can only have 2 states : "**New**" and "**Closed**".
  - We never move the mast tasks from "**New**" to "**Implementation**", "**Implementation**" to "**Testing Ready**", etc. We never touch the main task.
  - Only Head of R&D (myself) closes main tasks. "*Implementation*" subtasks are however closed by the tester.
  - Main tasks are assigned as explained above to the main developer working on the topic
- The subtask "*Implementation*" is used to actually track the status
  - The subtask is moved in the different status according the to state of the job. It can also be re-assigned.

We use following statuses and rules:



## 7. Conclusion

Adopting the set of practices described in this article really helped us not only to adopt agility within the development team but also in all activities surrounding it. The whole company got used to our ways and takes part in the process either by taking an active part in the **Product Management Committee** to help define the user stories or simply by downloading the latest Demo VM at the end of every sprint.

In addition, as stated above, adopting Agile principles and practices are a ground requirement towards adopting some DevOps or Lean Startup practices that really help not only the efficiency of the development team but really the company as a



whole by improving quality of the software and simply all our interactions with the other teams or even the customers. In addition, they have been key to make us shorten the lead time from ideas to production rollout of new features and our responsivity as a whole company.

While I can imagine that there can be situations where a standard waterfall approach may make more sense, I haven't encountered any in my career. Project size is not an argument there since Agility can be transposed to multiple team projects up to several dozens of thousands of man days projects. This is called **Scaling Agile** with dedicated framework such as [SAFe - Scaled Agile Framework](#). I hope I'll have some experience to share in this regards in another life.