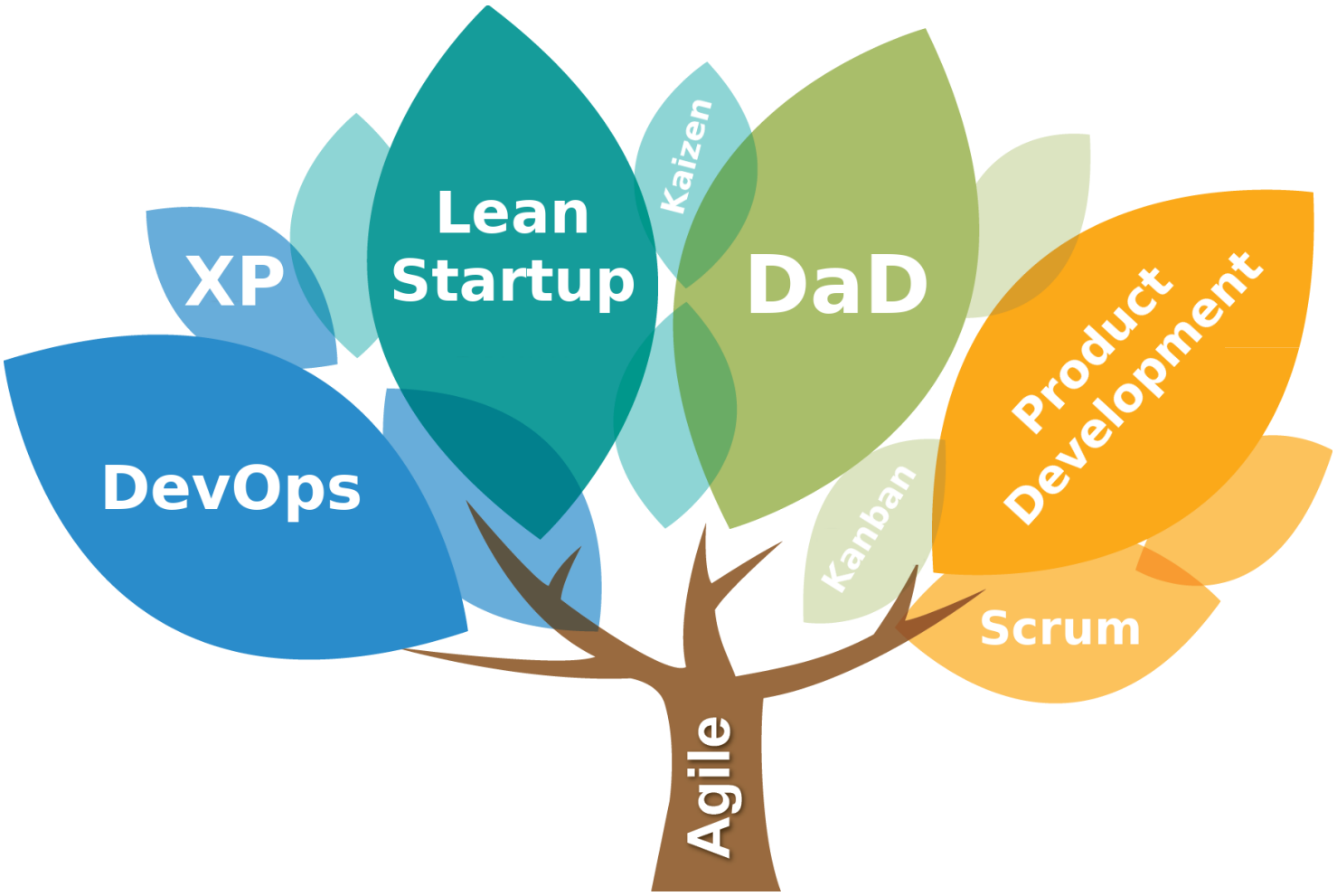
The Agile Method Collection

A collection of articles and writings on today's most indispensable Agile Methods.

Jerome Kehrli



Agility in Software Development is a lot of things, a collection of so many different methods. In a recent article I presented the Agile Landscape V3 from Christopher Webb which does a great job in listing these methods and underlying how much Agility is much more than some scrum practices on top of some XP principles.

I really like this infographic since I can recover most-if-not-all of the principles and practices from the methods I am following.

Recently I figured that I have written on this very blog quite a number of articles related to these very Agile Methods and after so much writing I thought I should assemble these articles in a book.

So here it is, the The Agile Methods Collection book.

The book is simply a somewhat reformatted version of all the following articles:

- Agile Landscape from Deloitte
- Agile Software Development, lessons learned
- Agile Planning : tools and processes
- DevOps explained
- The Lean Startup - A focus on Practices
- Periodic Table of Agile Principles and Practices

So if you already read all these articles, don't download this book.

If you didn't so far or want to have a kind of reference on all the methods from the collection illustrated above, you might find this book useful.

I hope you'll have as much pleasure reading it than I had writing all these articles.

Table of Contents

1. Agile Landscape from Deloitte....7
   1.1 Agile Design....8
   1.2 Agile Development....9
   1.3 Agile Operation....9
   1.4 Agile Management....9
   1.5 Conclusion....11
2. Agile Software Development, lessons learned....12
   2.1 Agile Software Development....12
      2.1.1 Why Agile anyway ?....13
      2.1.2 Agile Development Value Proposition....14
      2.1.3 Scrum....16
      2.1.4 Kanban....17
      2.1.5 Prerequisites : XP !....18
      2.1.6 Benefits : DevOps, Lean Startup....19
   2.2 Scrum roles....20
   2.3 From Story Maps to Product Backlog....22
      2.3.1 User Stories....22
      2.3.2 Story Maps....23
      2.3.3 From User stories to Developer Tasks....26
   2.4. From User Stories to Releases....27
      2.4.1 Composing our releases....28
      2.4.2 Composing the sprint....29
      2.4.3 Estimations in Story Points....31
   2.5. Introducing our sprints....33
      2.5.1 Before Sprint....33
      2.5.2 During Sprint....34
      2.5.3 After Sprint....35
   2.6 Release Backlog and Sprint Backlog....36
      2.6.1 Different release backlogs, long term backlog, sprint backlog 36
      2.6.2 While being Agile....38
      2.6.3 Handling customer requests and production concerns....39
      2.6.4 Sprint Kanban backlog management....39
   2.7 Conclusion....41
3. Agile Planning : tools and processes....43
   3.1 Introduction....45
   3.2 The Fundamentals....47
      3.2.1 eXtreme Programming....47

3.2.2 Scrum....50
3.2.3 DevOps....52
3.2.4 Lean Startup....54
3.2.5 Visual Management and Kanban....56
3.2.5.1 Story Map....57
3.2.5.2 Product Backlog....60
3.2.5.3 Kanban Board....61
3.2.5.4 User Stories....62
3.3. Principles....63
3.3.1 The tools....64
3.3.2 The Organization....64
3.3.2.1 Required roles....65
3.3.2.2 Required Committees and teams....66
3.3.3 The Processes....68
3.3.3.1 Design Process....68
3.3.3.2 Estimation Process....70
3.3.3.3 Product Kanban Board Maintenance Process....74
3.3.3.4 Story Map and Backlog synchronization Process....78
3.3.3.5 Forecasting....80
3.3.3.6 Development process: Scrum....83
3.3.4 The Rituals....85
3.3.4.1 Product Management Committee....85
3.3.4.2 Architecture Committee....86
3.3.4.3 Sprint Management Committee....87
3.3.4.4 Development Team - Daily Scrum....88
3.3.5 The Values....88
3.4 Overview of the whole process....89
3.5 Return on Practices....90
3.6. Conclusion....91
4. DevOps explained....93
4.1 Introduction....94
4.1.1 The management credo....95
4.1.2 a typical IT organization....95
4.1.3 Ops frustration....97
4.1.4 Infrastructure automation....98
4.1.5 DevOps : For once, a magic silver bullet....100
4.2 Infrastructure as Code....101
4.2.1 Overview....102
4.2.2 DevOps Toolchains....102
4.2.3 Benefits....103
4.3 Continuous Delivery....104
4.3.1 Learn from the field....106
4.3.2 Automation....107
4.3.3 Deploy more often....107

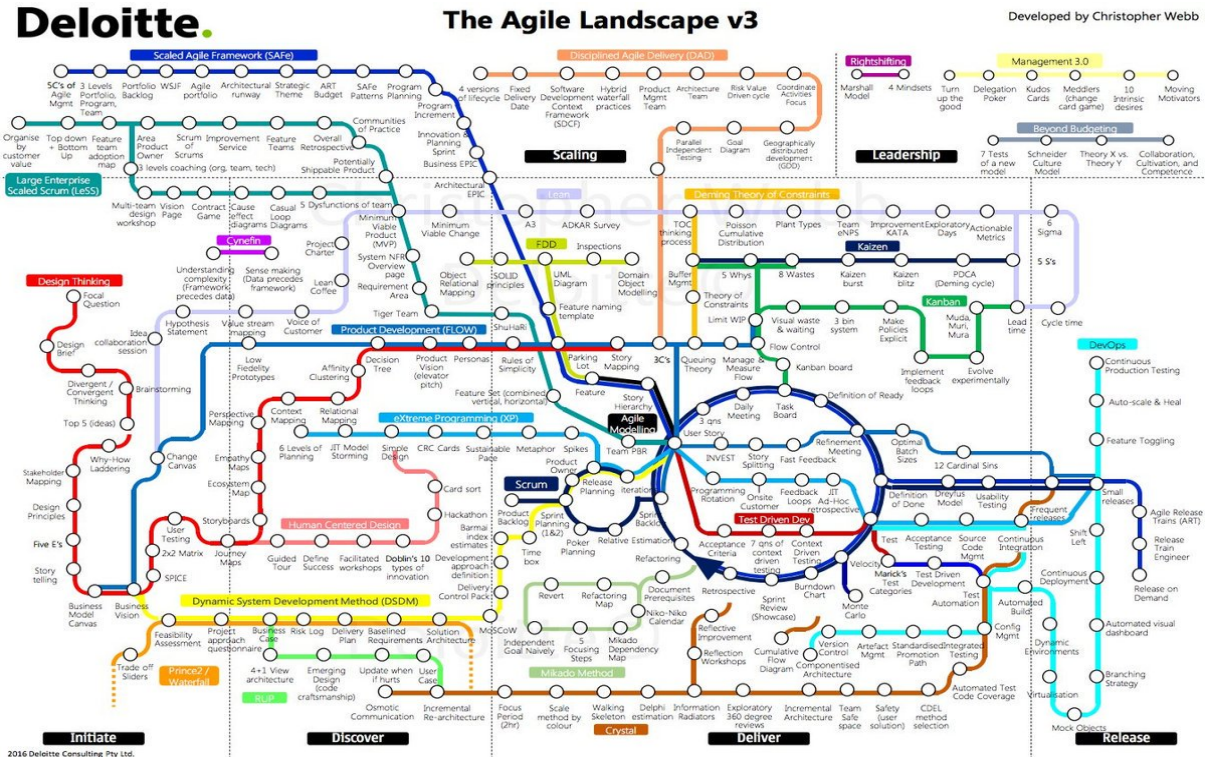
4.3.4 Continuous Delivery requirements....109
4.3.5 Zero Downtime Deployments....109
4.4 Collaboration....112
4.4.1 The wall of confusion....113
4.4.2 Software Development Process....115
4.4.3 Share the Tools....116
4.4.4 Work Together....117
4.5 Conclusion....118
5. The Lean Startup - A focus on Practices....120
5.1. The Lean Startup....121
5.1.1 Origins....122
5.1.2 The movement....123
5.1.3 Principles....123
5.1.4 The Feedback Loop....125
5.1.5 Business Model Canvas and Lean Canvas....126
5.1.6 Customer Development....129
5.2 The four steps to the Epiphany....129
5.2.1 Overview....130
5.2.2 A 4 steps process....130
5.3 Lean startup practices....131
5.3.1 Customer Discovery....132
5.3.1.1 Get out of the building....133
5.3.1.2 Problem interview....136
5.3.1.3 Solution interview....137
5.3.2 Customer Validation....138
5.3.2.1 MVP....139
5.3.2.2 Fail Fast....141
5.3.3 Re-adapt the product....143
5.3.3.1 Metrics Obsession....144
5.3.3.2 Pivot....145
5.3.4 Get new customers....147
5.3.4.1 Pizza Teams....147
5.3.4.2 Feature Teams....148
5.3.4.3 Build vs. Buy....151
5.3.4.4 A/B Testing....152
5.3.4.5 Scaling Agile....153
5.3.5 Company creation....154
5.4. Conclusions....155
6. Periodic Table of Agile Principles and Practices....156
6.1 The Periodic Table of Agile Principles and Practices....156
6.2. Layout Principle....157
6.3. Remarks....158
6.4. Principles and Practices....158

6.4.1 XP.....158
6.4.2 Scrum.....162
6.4.3 Product Development.....166
6.4.4 DevOps.....168
6.4.5 Lean Startup.....172
6.4.6 Kanban.....174
6.4.7 Kaizen.....174
6.4.8 FDD (Feature Driven Development).....175
6.4.9 DAD.....176

1. Agile Landscape from Deloitte

I've seen this infographic from Christopher Webb at Deloitte (at the time) some months ago.
This is the most brilliant infographic I've seen for years.

Christopher Webb presents here a pretty extended set of *Agile Practices* associated to their respective *frameworks*. The practices presented are a collection of all Agile practices down the line, related to engineering but also management, product identification, design, operation, etc.



(Source : Christopher Webb - LAST Conference 2016 Agile Landscape - https://www.slideshare.net/ChrisWebb6/last-conference-2016-agile-landscape-presentation-v1)

I find this infographic brilliant since its the first time I see a "*one ring to rule them all*" view of what I consider should be the practices towards scaling Agility at the level of the whole IT Organization.

Very often, when we think of Agility, we limit our consideration to solely the Software Build Process.
But Agility is more than that. And I believe an Agile corporation should embrace also **Agile Design**, **Agile Operations** and **Agile Management**.

This infographic does a great job in presenting how these frameworks enrich and complements each others towards scaling Agility at the level of the whole IT Organization.

To be honest there are even many more frameworks that those indicated on this infographic and Chris Webb is presenting some additional - reaching 43 in total - in his presentation.
But I believe he did a great job in presenting the most essential ones and presenting how these practices, principles and framework work together to achieve the ultimate goal of every corporation: skyrocketing employee productivity and happiness, maximizing customer satisfaction and blowing operational efficiency up.

Now I would want to present why I think considering Agility down the line in each and every aspect around the engineering team and how these frameworks completing each other are important.

1.1 Agile Design

Normally I am a little sensitive with formal meaning of the word *design* in software engineering.

But for once I'll make an exception.
So for once, by *design* here, I mean the largest possible definition of the term, encompassing as much the discovery of the key features as well as the architecture of the system to be implemented.

Agility in identifying beforehand the product to be implemented and its key features is a must.
Later when the rough form of the product is identified, the process consists in having a Vision workshop to align the stakeholders on the product vision, then Story Mapping workshops, all of these emphasizing Agility, Adaptation and lightweight processes in comparison to the tons of documents produced by more traditional methods.

This is pretty well covered in the infographic above and Design thinking covers all the practices that seem key to me such from the light *Business Model Canvas* to *Product Vision* definition workshops and *Story Mapping* workshops.

At the end of the day, Agility is mostly about the capacity to *adapt* and *react* to changing requirements and changing priorities. Enforcing thorough product identification and feature design phases before actually initiating the development of an **MVP** aimed at validating (or contradicting) the hypothesis makes little sense in my opinion.
One important framework to consider here is the *Lean* approach and the Lean startup Practices discussed in chapter 5. The Lean Startup - A focus on Practices.

At the end of the day, *Agile Software Development* methodologies cannot deploy their full potential if the company itself is not Agile.

1.2 Agile Development

At the root of everything there is XP. *eXtreme Programming* was mostly initiated by Kent Beck, strong from his experience on the C3 project. Kent Beck hardly invented a lot of things but rather took some practices more or less used previously in the industry and took them to *extreme* levels.

Agile Software Development is really built on top of XP genes. Today XP is considered just another Agile Software Development Framework, but I don't share that view. To me, XP and the related practices form the most fundamental core of Agile Software Development Methodologies.
XP Practices take a form or another in the various Agile Frameworks such as RDD, Scrum, Kanban, Scrumban, etc. In some of them some core XP practices are not mentioned; not because they should not be applied, but really because they're nowadays considered so natural that they're assumed. Think for instance of TDD (Unit Tests first), Continuous Integration, Simple Metaphor (Meaningful Naming, Domain Driven Design, Design patterns), etc.

I discussed in this chapter : 2. Agile Software Development, lessons learned the software development methodology we are using in my current company and interestingly all of our practices are pretty well identified on the infographic above.

1.3 Agile Operation

Agile operation is really about **DevOps**.

I developed in length in a dedicated article on this very blog what DevOps is and why it's important so I let the reader refer to this chapter : 4. DevOps explained.

Let's just mention that here as well it is hard for the development team to leverage its Agile practices if the other departments of the corporation - and out of those the operation is crucial - have not embraced Agility.

1.4 Agile Management

Agile management is about Leadership and Leadership pursues the goal of growing and transforming organizations into great places to work for, where people are engaged, the work is improved and customers are simply delighted.

Agile Management is a lot about Management 3.0.

Management 1.0 was about doing the wrong thing, by treating people like cogs in a system.

Management 2.0 was about doing the right thing wrong, with understanding the goals and having good intentions, but using old-fashioned top-down initiatives.

Management 3.0 is about doing the right thing for the team, involving everyone in improving the system and fostering innovation.

Agile Management is about making the components of the Agile corporation collaborate together towards anticipating changes and adapt smoothly and flawlessly.
There are three most essential vectors:

- **Collective Intelligence**: which is key to address and control the increasing complexity of organizations and businesses and based on having everyone in the company taking part in the continuous improvement processes
- **Optimal use of Technology** : Technology is an amazing vector of efficiency in regards to tools supporting the organization
- **A sound adoption of Continuous Improvement Processes** : making the organization identify and build on its strength while continuously addressing its weaknesses to adapt itself continuously.

Agile Management values individual and interactions over formal processes and hierarchy. It really consists in empowering people and making the organization a place where they can develop themselves with passion and energy, leveraging their capacity for both action and innovation.
Now of course this needs to be driven and Agile Management encourages continuous feedback in the form, for instance, of O3s - *One-On-One* - on a regular basis where both the employee and the manager can provide feedback on the organization, respectively the performance of the employee.

Managing performance in this sense is identifying the strengths of the employee, which we should leverage, and the weaknesses, which we should address and improve.

Empowering people is a key practice since, at the end of the day, Management is too important to be left to Managers ;-)
Agility is about adaptation but also about efficiency and quality (think XP practices here) and Agile Management is about putting practices in place aimed at making engineers give the best they can and participate at every level in the success of the company.

I would conclude this section by giving my favorite definition of management:

"*Hire great people, and then get the hell out of their way.*"

1.5 Conclusion

This infographic is an awesome view of what we have achieved over the last 10 to 15 years in regards to understanding of how to design, engineer, build and manage better.
I believe finding better ways of working should be an everyday concern for organizations, from startups to international corporations.

Quoting Jack Welsh:

> "*If the rate of change on the outside exceeds the rate of change on the inside, the end is near.*"

My personal pick-up is:

- Lean (Startup) – See 5. The Lean Startup - A focus on Practices
- DevOps – See 3.2.3 DevOps
- XP, Scrum and Kanban (Agile Development) – See 2. Agile Software Development, lessons learned and 3. Agile Planning : tools and processes
- Management 3.0 - Empowering and energizing people, Developing competences, Aligning teams, Continuous Improvement
- Kaizen (of course)

I have no experience on *Scaling Agile* frameworks for now. It's becoming a pretty hot topic in my current company though and I'll come back with a new article when I have some.
My preference would go to LeSS I think, since it seems more natural to me. But that is just a pretty initial opinion, and it may change ...

2. Agile Software Development, lessons learned

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.

2.1 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 e**X**treme **P**rogramming movement such as Kent Beck, Ward Cunningham or Martin Fowler.

2.1.1 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 project 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

2.1.2 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.

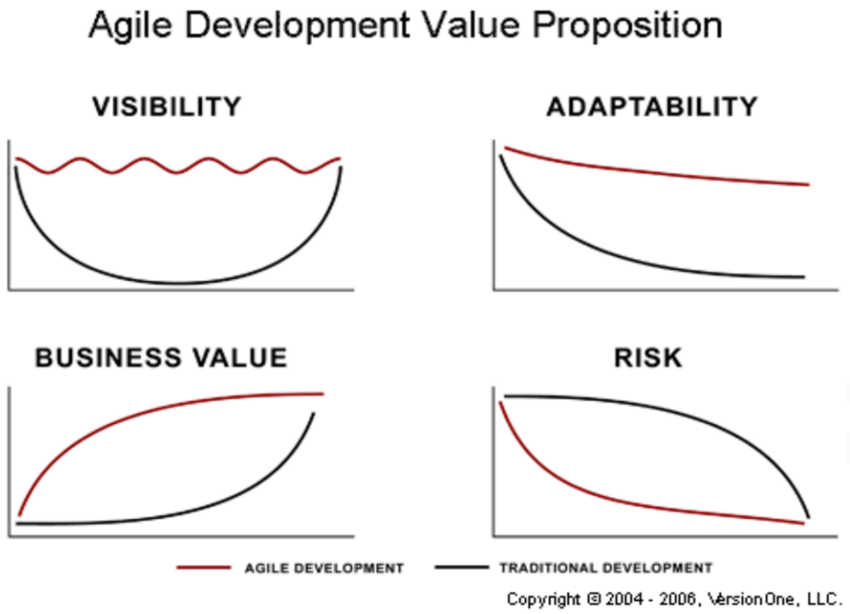
Customer collaboration over contract negotiation

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.

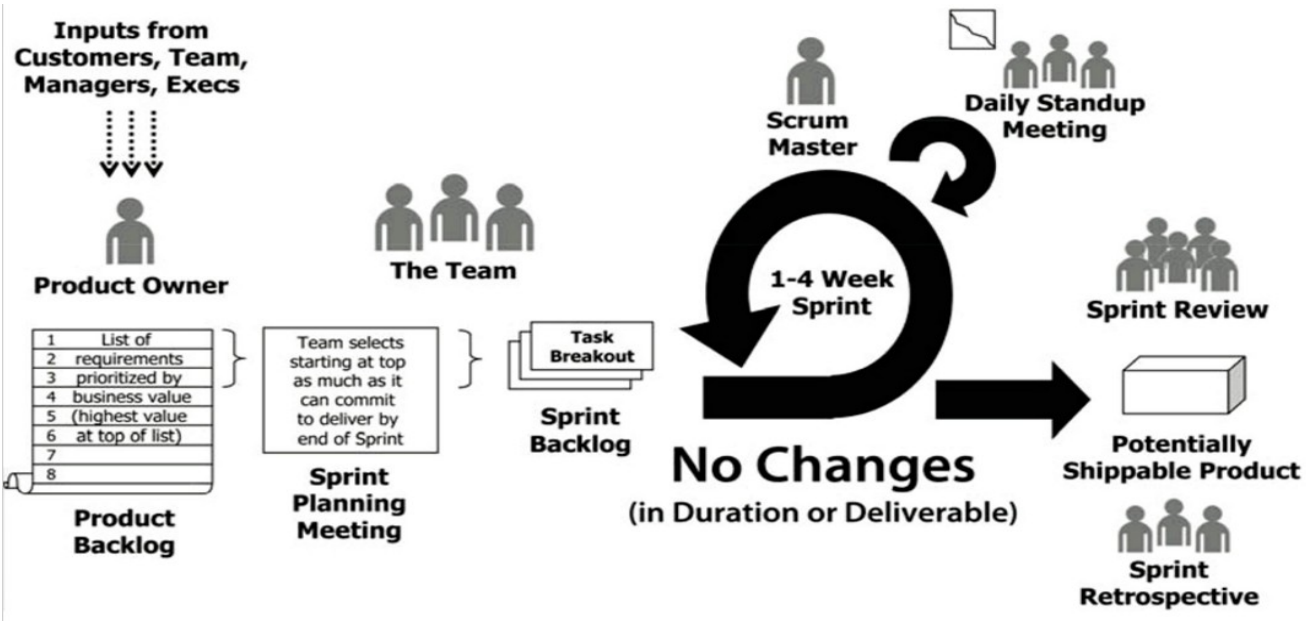
Summary



2.1.3 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.

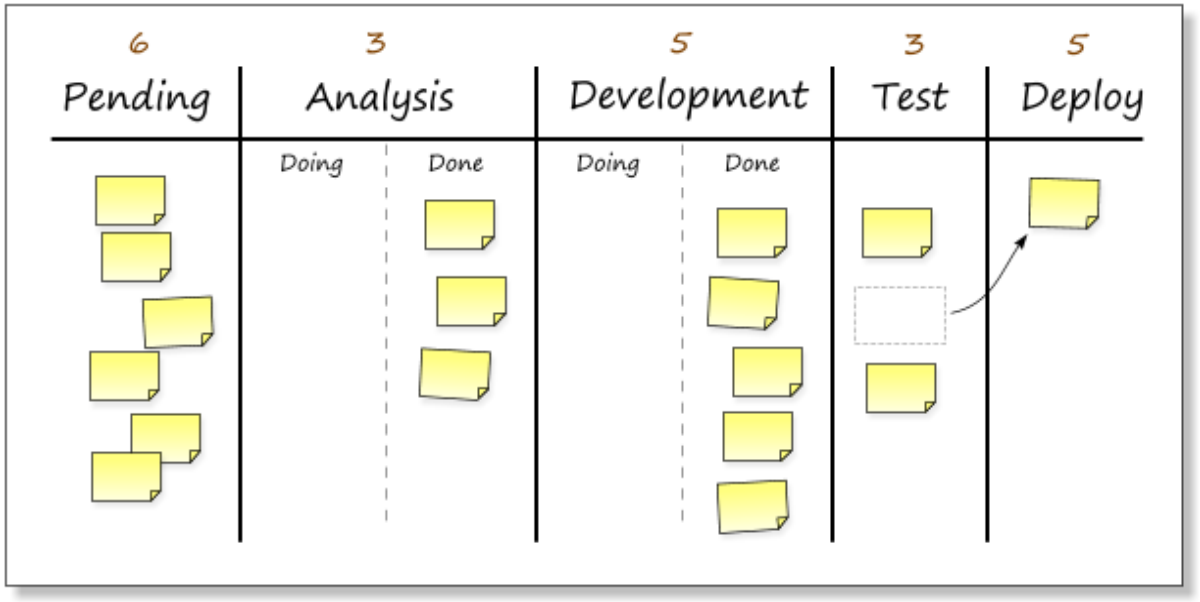
Scrum leverages Commitment as Change Agent

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.

2.1.4 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.



2.1.5 Prerequisites : XP !

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

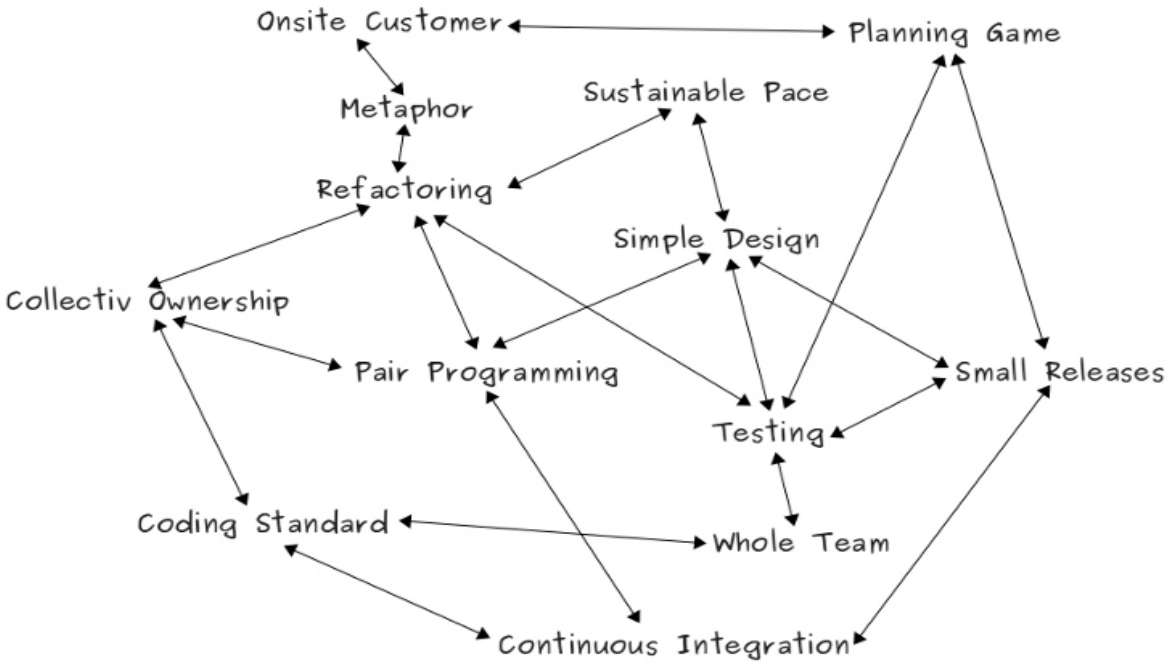
While some practices are applied sometimes a little differently in scrum, really all of them are important. XP Practices are really intended to be respected all together since they have interactions and one cannot benefit from the advantages of XP if one's picking up only a subset of the practices.
Some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

2.1.6 Benefits : DevOps, Lean Startup

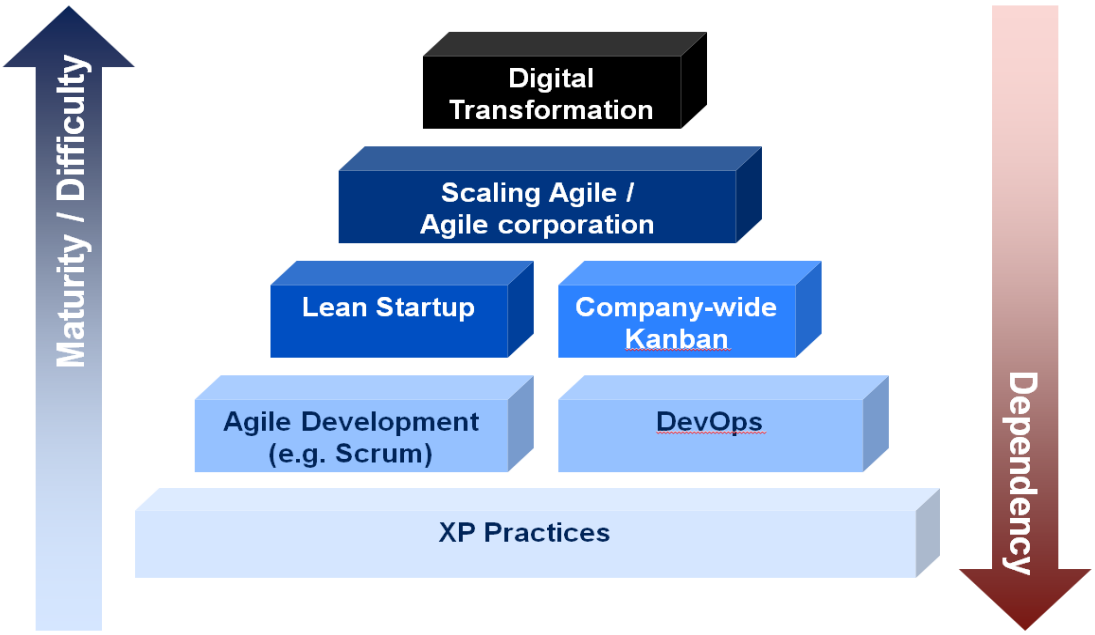
Adoption of an *Agile Development Methodology* is the very ground on which many other practices or principles are built, should a Tech Company or an IT Department want to move forward towards more efficiency, shorter lead times, better reactivity and controlled costs.

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:



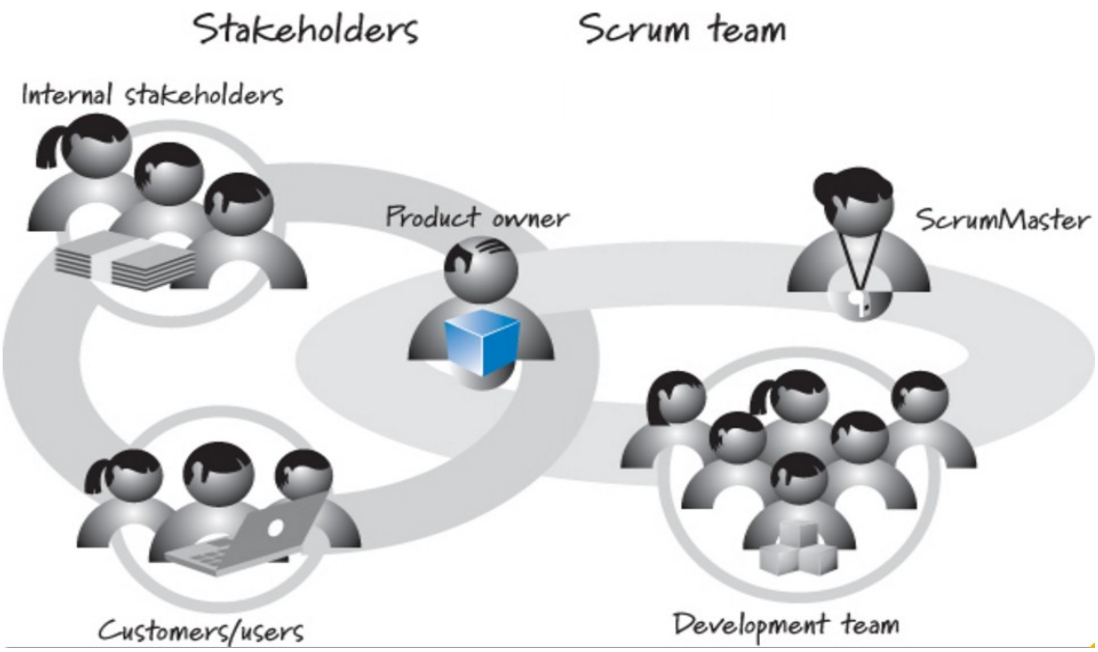
As shown on the above pyramid, a good adoption of sound DevOps and Lean Startup practices itself is a prerequisite towards further transformations: from enterprise-scale Lean-Agile development to ultimate Digital transformation of the company. Well I guess developing these concepts should be the topic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

2.2 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 end 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.

2.3 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**.

2.3.1 User Stories

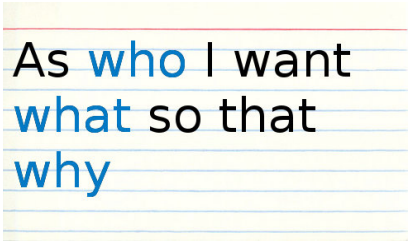
User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

They typically follow a simple template:

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

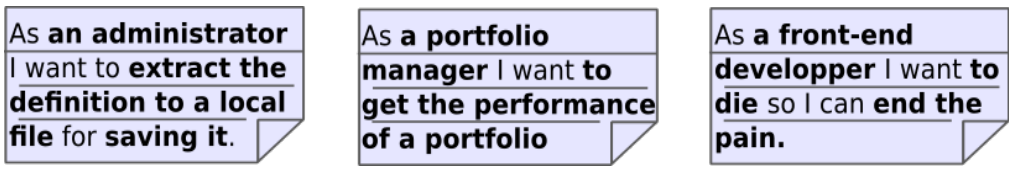
User stories are often written on sticky notes and arranged on walls or tables to facilitate planning and discussion.

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.



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:



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.

2.3.2 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.

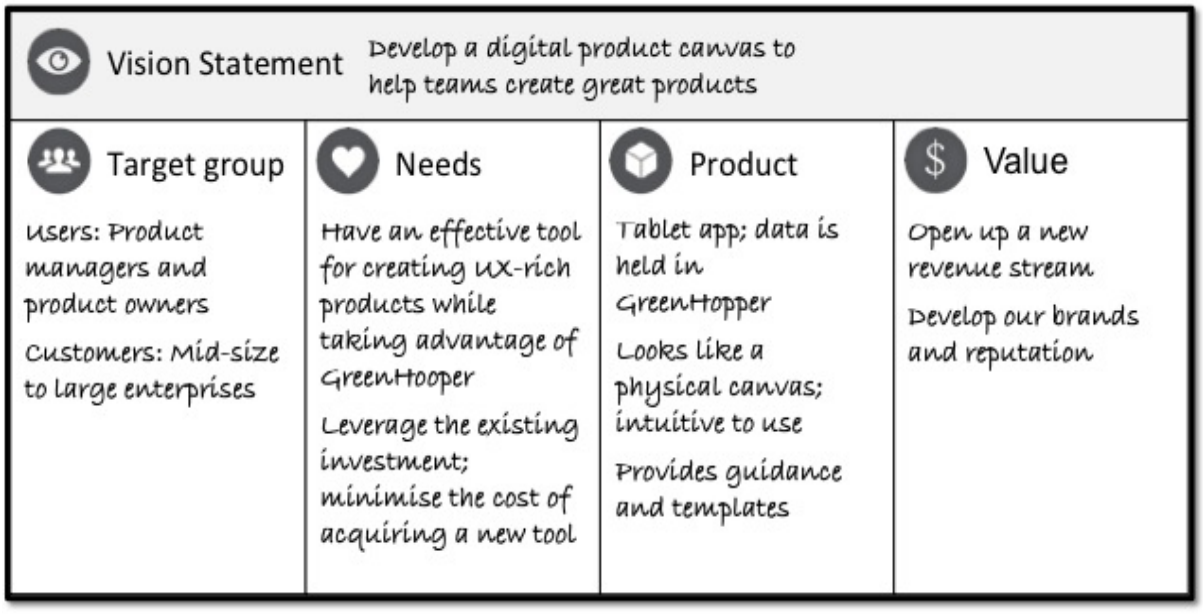
This is the purpose of the **Story Mapping** workshop. This is typically a few days workshop organized as max 4 hours sessions per day where **all the stakeholders** (this is important) take the time to sit in a room together and define the product with the help of User Stories.

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:

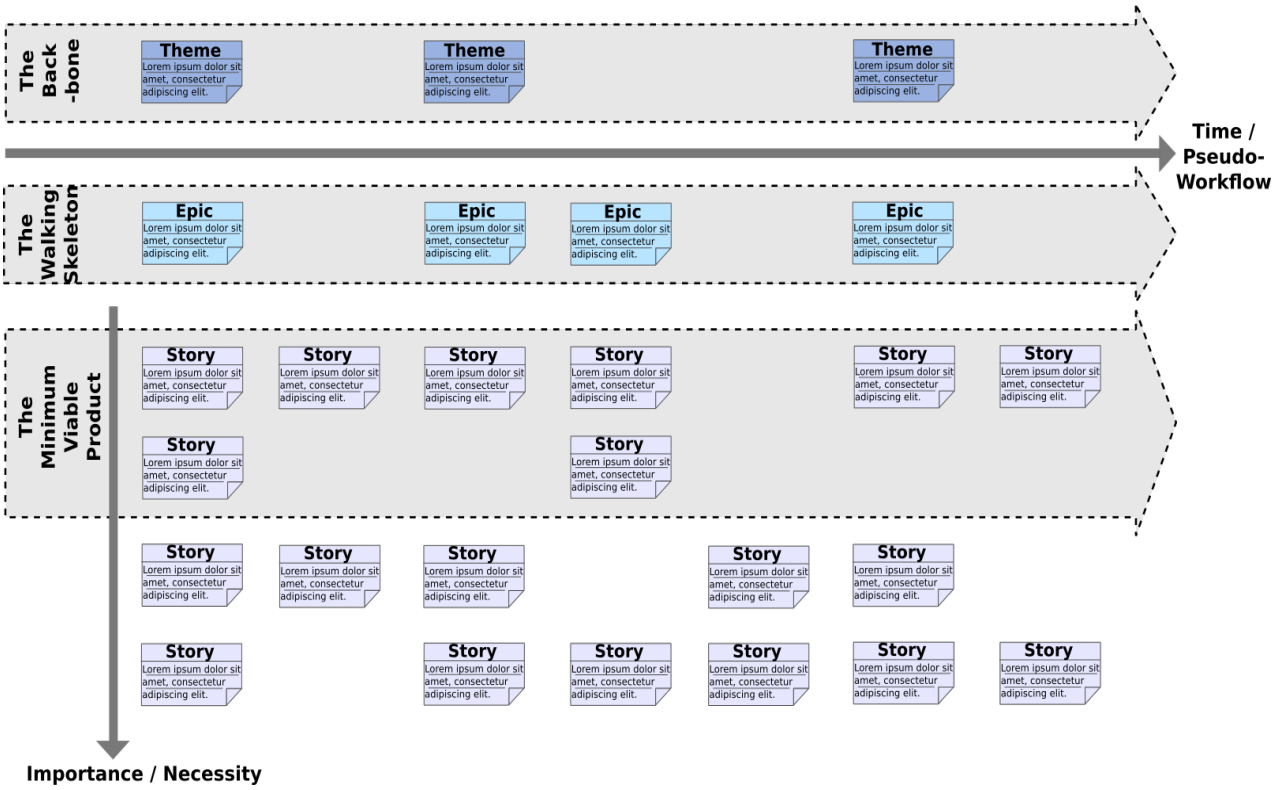


Story Map

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.

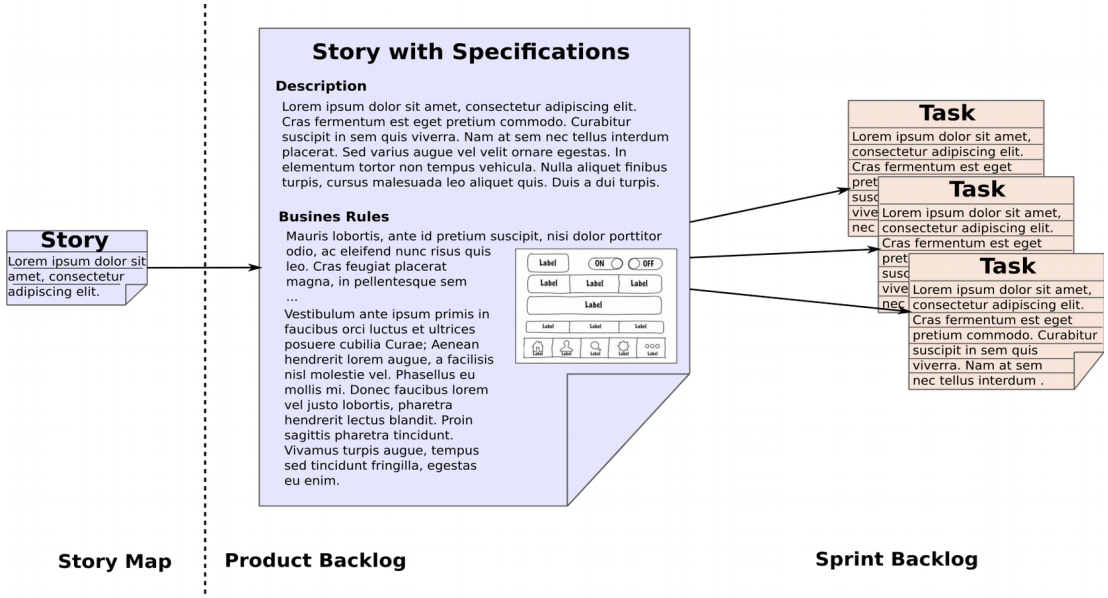
When it comes time to prioritize stories, we don't prioritize the backbone or the walking skeleton. We do prioritize the ribs - the stories hanging down from the backbone. We place them high to indicate they're absolutely necessary, lower to indicate they're less necessary.
By doing this, we find that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

2.3.3 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.

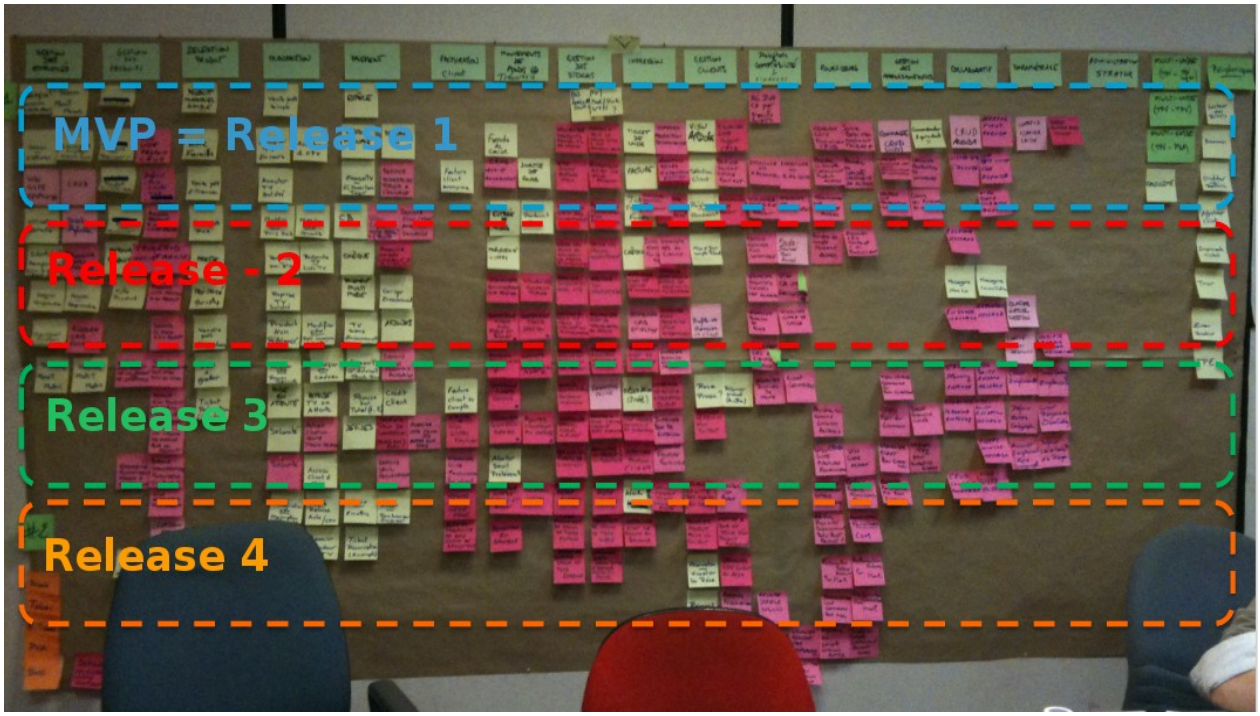
2.4. From User Stories to Releases

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.

2.4.1 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.

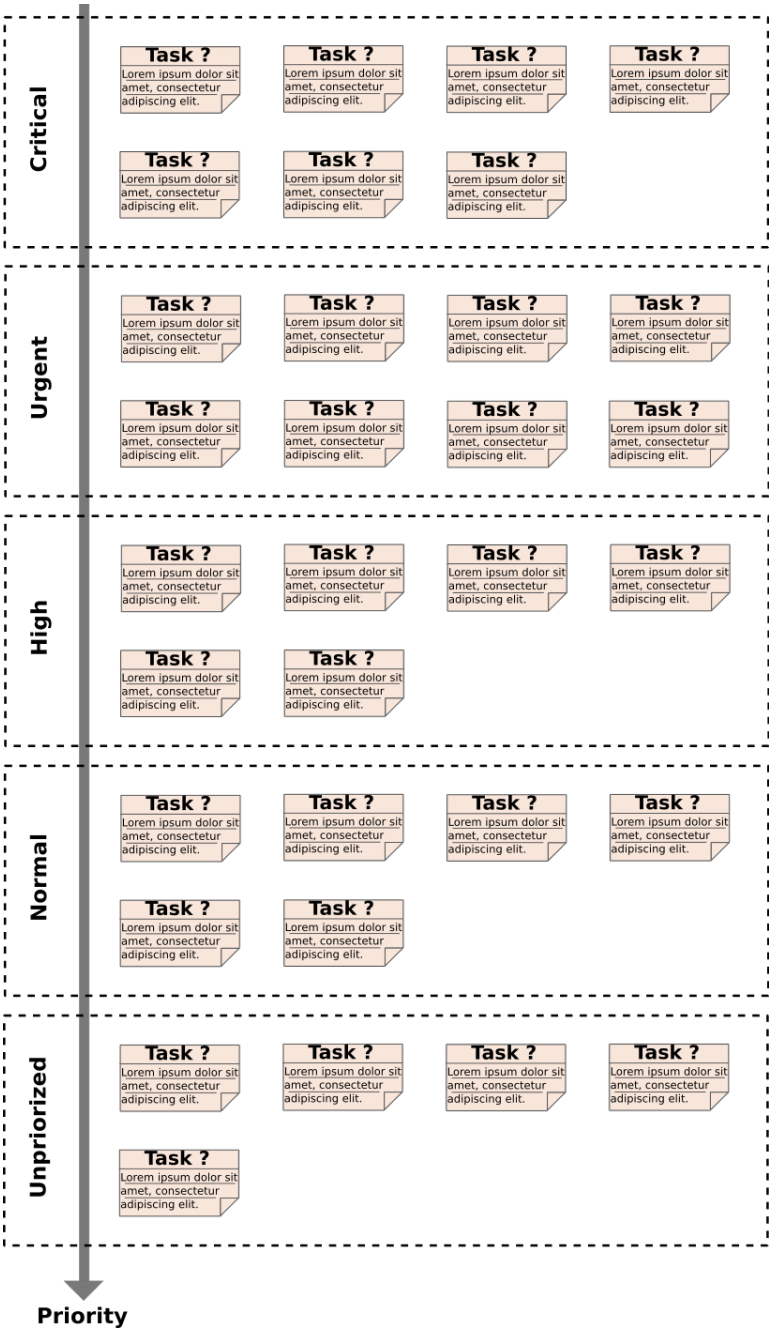
2.4.2 Composing the sprint

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.

2.4.3 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.**
- Third, 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 ...

2.5. 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.

2.5.1 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

2.5.2 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

2.5.3 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*:

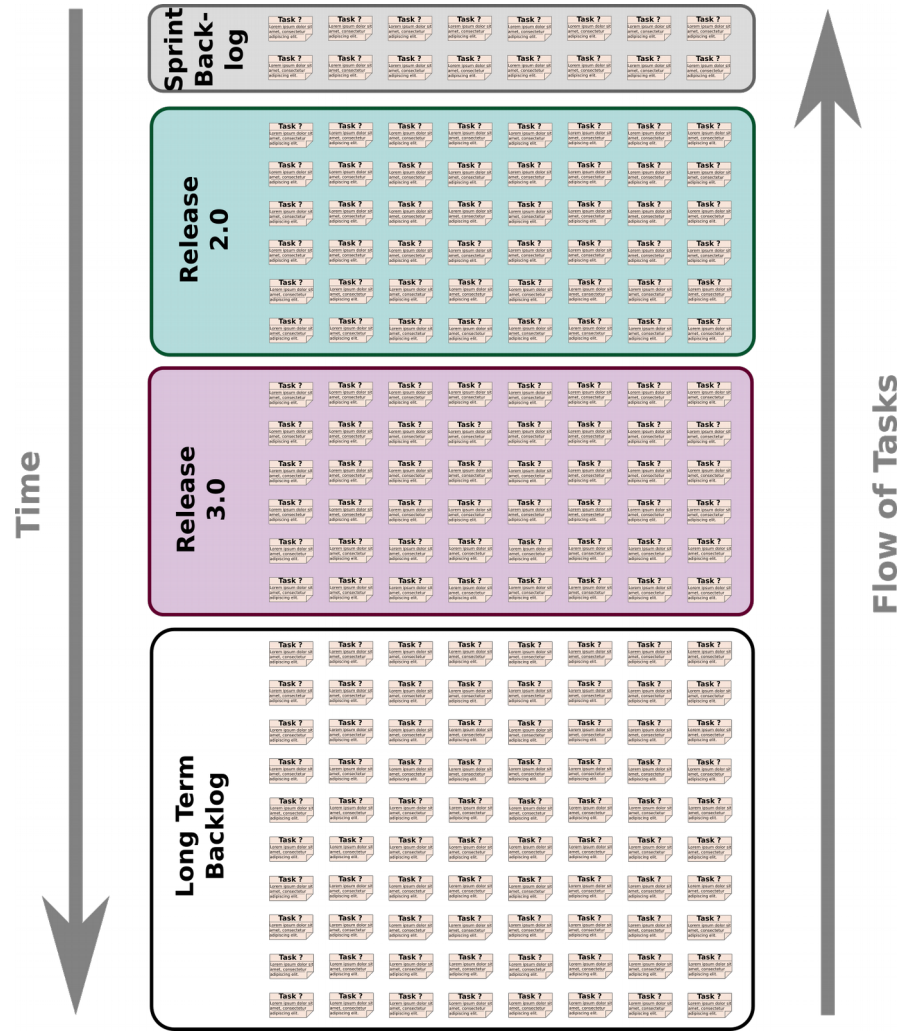
- We demonstrate new functionalities to our internal user representatives
- We take minutes of the meeting in the form of new tasks added to the backlog or updates to existing tasks

2.6 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.

2.6.1 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.

2.6.2 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 ot 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.

2.6.3 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.

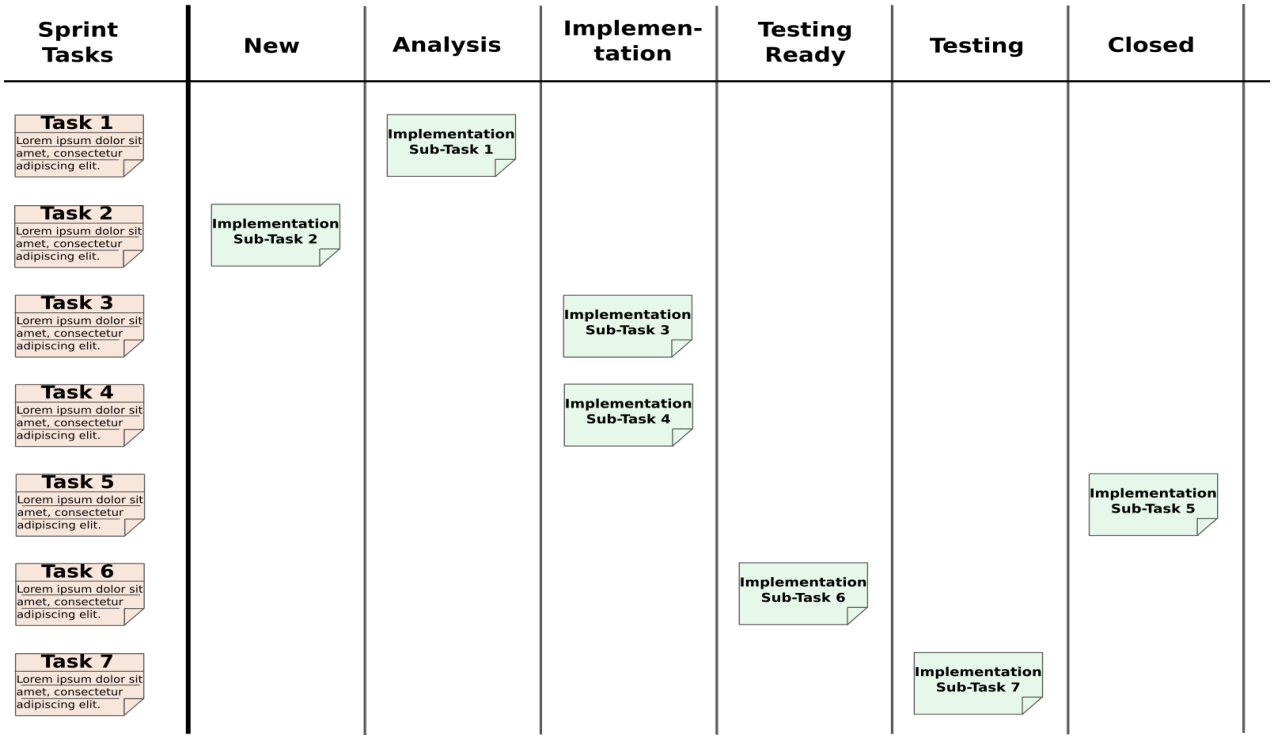
We have a special way of handling such new features or bug fixes which takes the form of a special tracker named **wish**.

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*.

2.6.4 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.

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.

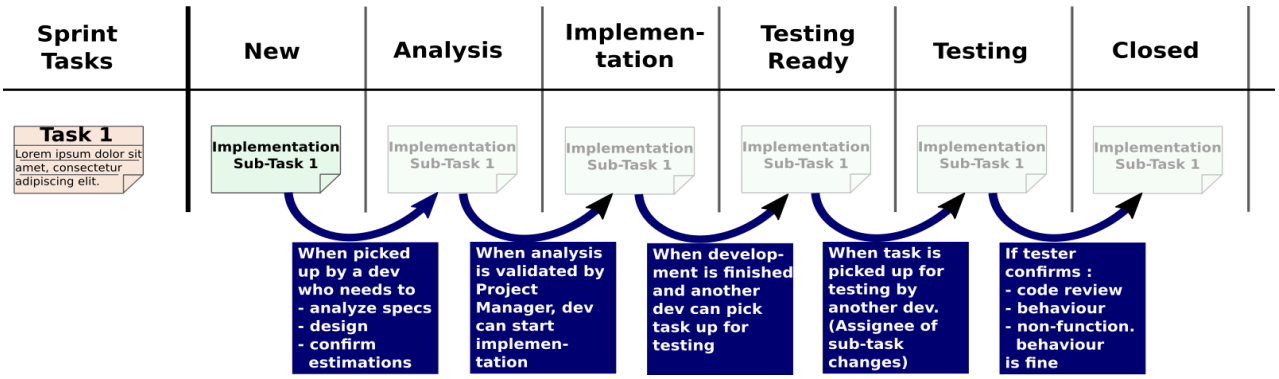
The subtasks (Implementation of ...) can be assigned to different developers when it is passed from a developer to another one.
A In this case, the assignee of the main task (the parent) remains the developer who did most work on it.
In addition, when a tester takes a subtask from "**Testing Ready**" 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:



2.7 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 responsiveness 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.

3. Agile Planning : tools and processes

All the work on Agility in the Software Engineering Business in the past 20 years, initiated by Kent Beck, Ward Cunningham and Ron Jeffries, comes from the finding that traditional engineering methodologies apply only poorly to the Software Engineering business.

If you think about it, we are building bridges from the early stages of the Roman Empire, three thousand years ago. We are building heavy mechanical machinery for almost three hundred years. But we are really writing software for only fifty years. In addition, designing a bridge or a mechanical machine is a lot more concrete than designing a Software. When an engineering team has to work on the very initial stage of the design of a bridge or mechanical machine, everyone in the team can picture the result in his mind in a few minutes and breaking it down to a set of single Components can be done almost visually in one's mind.

A software, on the other hand, is a lot more abstract. This has the consequence that a software is much harder to describe than any other engineering product which leads to many levels of misunderstanding.

The waterfall model of Project Management in Software Engineering really originates in the manufacturing and construction industries.
Unfortunately, for the reasons mentioned above, despite being so widely used in the industry, it applies only pretty poorly to the Software Engineering business. Most important problems it suffers from are as follows:

- **Incomplete or moving specification:** due to the abstract nature of software, it's impossible for business experts and business analysts to get it right the first time.
- **The tunnel effect:** we live in a very fast evolving world and businesses need to adapt all the time. The software delivered after 2 years of heavy development will fulfill (hardly, but let's admit it) the requirements that were true two years ago, not anymore today.
- **Drop of Quality to meet deadlines:** An engineering project is always late, always. Things are just a lot worst with software.
- **Heightened tensions between teams:** The misunderstanding between teams leads to tensions, and it most of the time turns pretty ugly pretty quick.

So again, some 20 years ago, Beck, Cunningham and Jeffries started to formalize some of the practices they were successfully using to address the uncertainties, the

overwhelming abstraction and the misunderstandings inherent to software development. They formalized it as the eXtreme Programmingmethodology.

A few years later, the same guys, with some other pretty well known Software Engineers, such as Alistair Cockburn and Martin Fowler, gathered together in a resort in Utah and wrote the Manifesto for Agile Software Development in which they shared the essential principles and practices they were successfully using to address problems with more traditional and heavyweight software development methodologies.

Today, Agility is a lot of things (See 1. Agile Landscape from Deloitte) and the set of principles of practices in the whole Agile family is very large. Unfortunately, most of them require a lot of experience to be understood and then applied successfully within an organization.

Unfortunately, the complexity of embracing a sound Agile Software Development Methodology and the required level of maturity a team has to have to benefit from its advantages is really completely underestimated.
I cannot remember the number of times I heard a team pretending it was an Agile team because it was doing a Stand up in the morning and deployed Jenkins to run the unit tests at every commit. But yeah, honestly I cannot blame them. It is actually difficult to understand Agile Principles and Practices when one never suffered from the very drawbacks and problems they are addressing.

I myself am not an agilist. Agility is not a passion, neither something that thrills me nor something that I love studying in my free time. Agility is to me simply a necessity. I discovered and applied Agile Principles and practices out of necessity and urgency, to address specific issues and problems I was facing with the way my teams were developing software.

The problem on which I am focusing on here is Planning. Waterfall and RUP focus a lot on planning and are often mentioned to be superior to Agile methods when it comes to forecasting and planning.
I believe that this is true when Agility is embraced only incompletely. As a matter of fact, I believe that Agility leads to much better and much more reliable forecasts than traditional methods mostly because:

- With Agility, it becomes easy to update and adapt Planning and forecasts to always match the evolving reality and the changes in direction and priority.
- When embracing agility as a whole, the tools put in the hands of Managers and Executive are first much simpler and second more accurate than traditional planning tools.

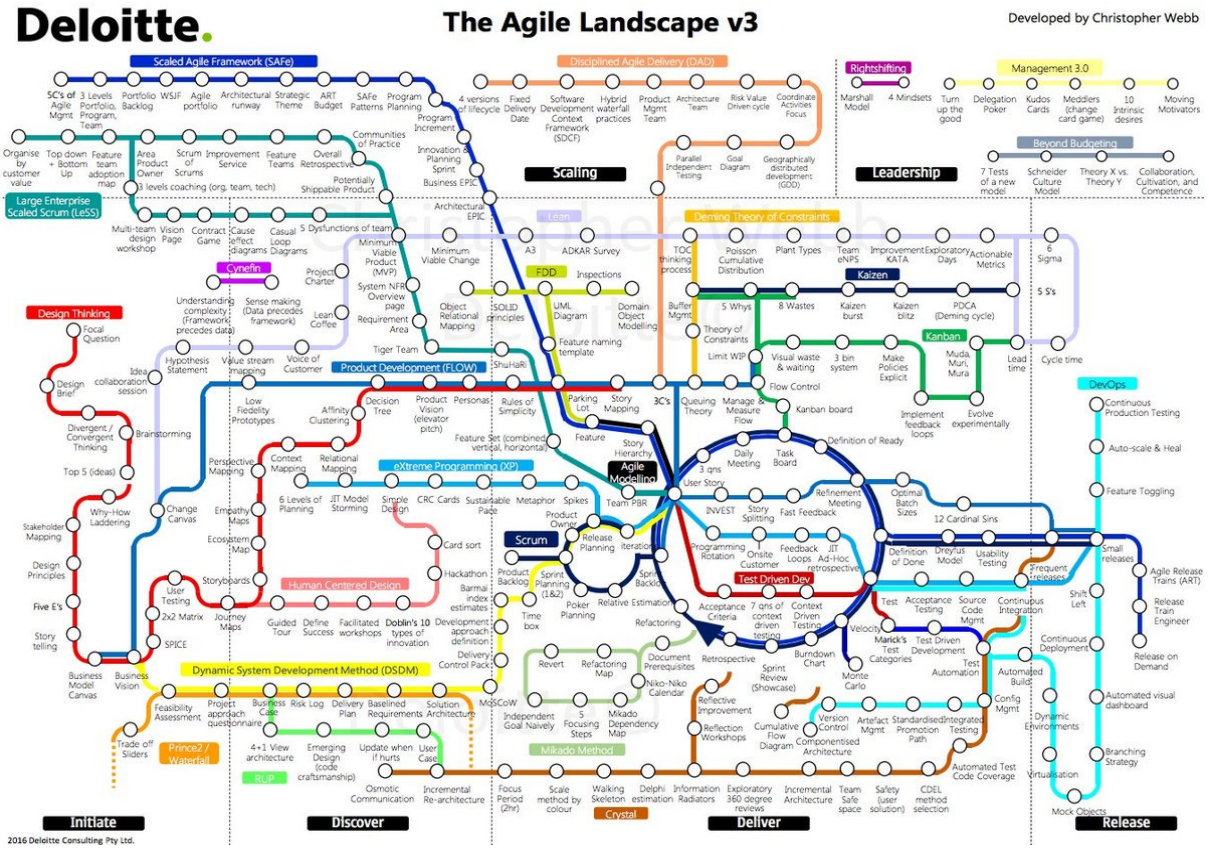
In this report, I intend to present the fundamentals, the roles, the processes, the rituals and the values that I believe a team would need to embrace to achieve

success down the line in Agile Software Development Management - Product Management, Team Management and Project Management - with the ultimate goal of making planning and forecasting as simple and efficient as it can be.
All of this is a reflection of the tools, principles and practices we have embraced or are introducing in my current company.

3.1 Introduction

As stated in my abstract above, embracing sound Agile principles and applying relevant Agile practices is all but easy.
First, out of all the Agile methods available and described and the overwhelming set of practices and principles, an organization needs to understand which makes sense to it. Adopting a method, a set or principles or practices blindly, because the paper said it was good, or because the Scrum Master believes it is *state of the art* makes only little sense.
The set of methods described nowadays is pretty huge and unfortunately, each and every of these practices make sense whenever a team, an organization or a whole corporation suffers from a drawback or an issue it addresses or simply benefits from its advantages.

The whole set of Agile methods along with their principles and practices are brilliantly represented by Chris Web on the following infographic:



(Source : Christopher Webb - LAST Conference 2016 Agile Landscape -

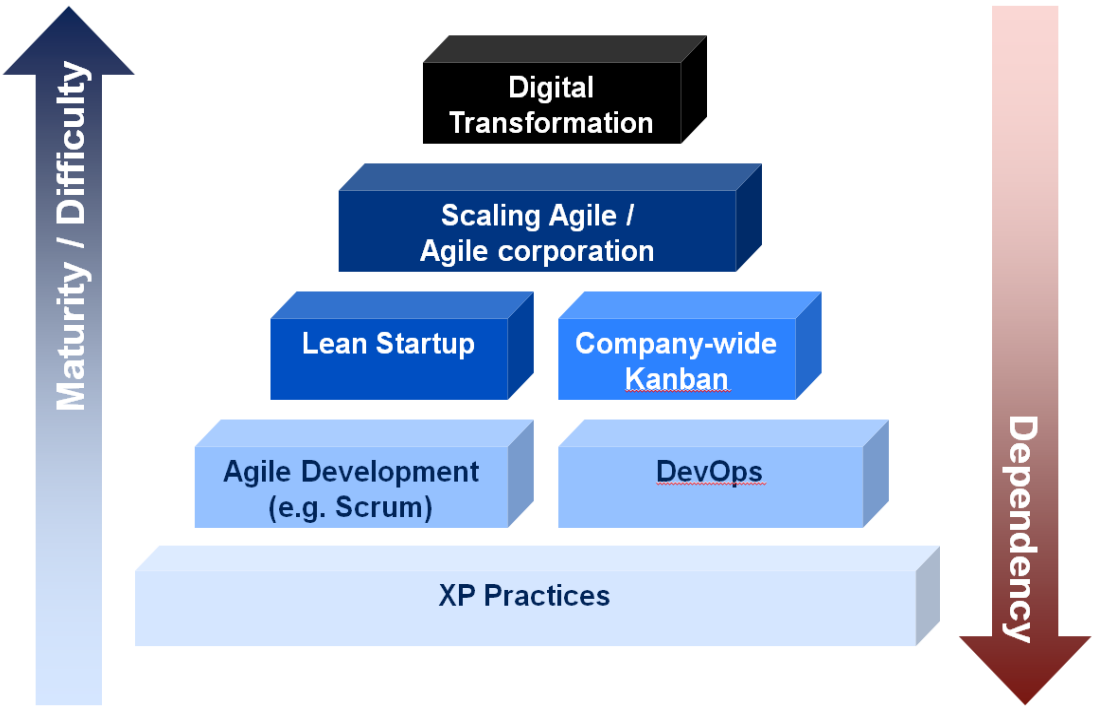
https://www.slideshare.net/ChrisWebb6/last-conference-2016-agile-landscape-presentation-v1)

Junior teams should go with a *base method* that makes sense to it, such as Scrum or Kanban **while remembering that none of it makes sense without a strict respect to the whole set of XP principles and practices**.

More experienced teams will likely come up with their own methodology, cleverly built from the principles and practices of several underlying methods.

Again, in my opinion **XP is the most fundamental building block on which all the rest is built**, not a method among others.
I often read papers online presenting XP as one Agile Software Development Method among others. My point of view is very different. I strongly believe - and experience everyday - that XP proposes the fundamental principles and practices on which are built all the other methods.
Without a thorough adoption of XP principles and practices, one cannot benefit from the full advantages of Agility. In addition, some principles and practices proposes by other methods such as DevOps, leverage on some XP principles and practices but never voids them.

When explaining this, I like to recover this schema I wrote a few years ago when I was doing consulting missions around Agility and Digital Transformation:



This reads as follows:

- 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), embracing 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

But then again, referring to Chris Webb's Agile Landscape, picking up the practices that make sense and have an added value in any context is the choice of every organization. Every different mature agile organization will use a slightly different set of practices than every other.

I will now be presenting the fundamental set of practices I deem important when it comes to successfully embracing Agile Planning and Agile Software Development.

3.2 The Fundamentals

The set of practices I deem essential to embrace *Agile Planning* comes from the following methods: XP, Scrum, Kanban, DevOps, Lean Startup and a lot of Visual Management tricks.

3.2.1 eXtreme Programming

e**X**treme **P**rogramming (XP) is the most fundamental software development method from the Agile tree< that focuses on the implementation of an application, without neglecting the project management aspect. XP is suitable for small teams with changing needs. XP pushes to extreme levels simple principles.

The *eXtreme programming* method was invented by Kent Beck, Ward Cunningham, Ron Jeffries and Palleja Xavier during their work on the project *C3*. C3 was the calculation of compensation project at Chrysler.
Kent Beck, project manager in March 1996, began to refine the development method used on the project. It was officially born in October 1999 with Kent Beck's *Extreme Programming Explained* book.

In the book Extreme Programming Explained, the method is defined as:

- An attempt to reconcile the human with productivity;
- A mechanism to facilitate social change;

- A way of improvement;
- A style of development;
- A discipline in the development of computer applications.

Its main goal is to reduce the costs of change. In traditional methods, needs are defined and often fixed at the start of the IT project, which increases the subsequent costs of modifications. XP is committed to making the project more flexible and open to change by introducing core **values**, **principles** and **practices**:

The principles of this method are not new: they have existed in the software industry for decades and in management methods for even longer. The originality of the method is to push them to the extreme:

- Since the code review is a good practice, it will be done permanently (by a binomial);
- Since the tests are useful, they will be done systematically before each implementation;
- Since the design is important, it will be done throughout the project (refactoring);
- Since simplicity makes it possible to advance faster, we will always choose the simplest solution;
- Since understanding is important, we will define and evolve metaphors together;
- Since the integration of the modifications is crucial, we will do it several times a day;
- Since the needs evolve rapidly, we will make cycles of development very rapid to adapt to the change.

Activities

- Coding
- Testing
- Listening
- Designing

Values

- Communication
- Simplicity
- Feedback
- Courage
- Respect

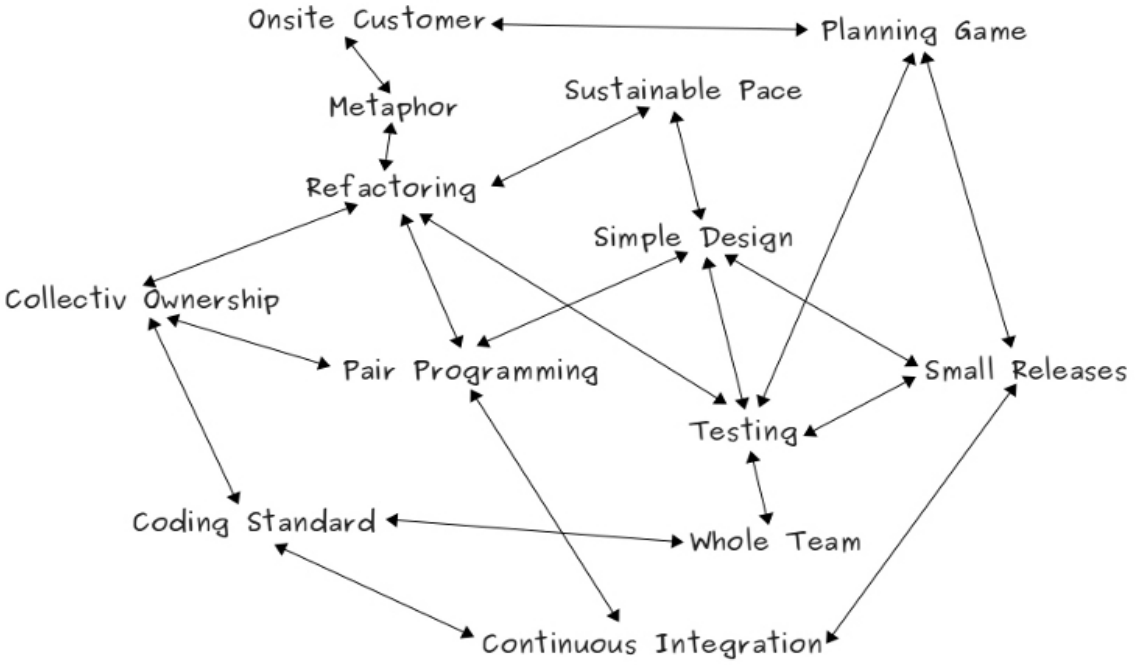
Principles

- Feedback
- Assuming Simplicity
- Embracing changes

Practices

- Pair programming
- Planning Game
- Test-Driven Development
- Continuous Integration
- Refactoring
- Coding Standards
- Collective Code Ownership
- On site customer
- Simple Design
- System metaphor
- Sustainable Pace
- The Planning game
- Boy Scout Rule
- No premature Optimization
- Confirm bugs by failing tests

The practices listed by the eXtreme Programming method form the fundamental Software Engineering Practices of Agility.
Interestingly, one cannot pick up a subset of these practices and believe that it should work. Kent Beck uses the following schematic to illustrate how these practices work together and depend on each others:



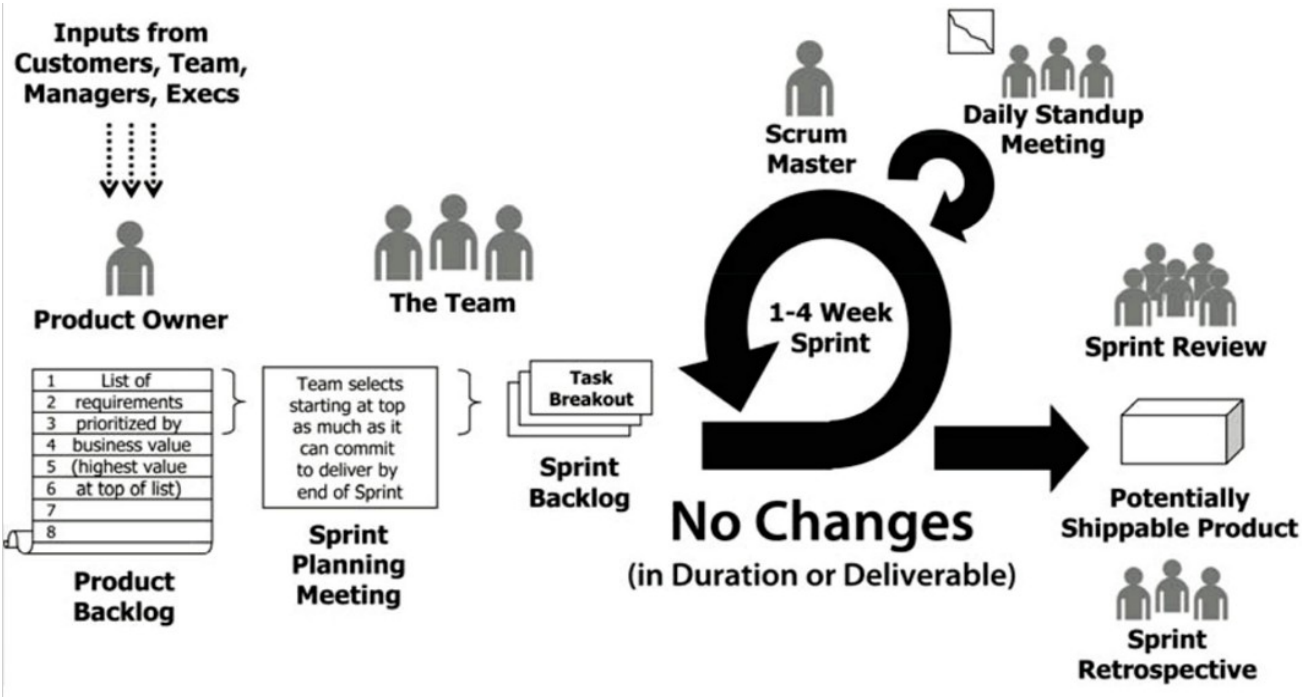
All of this makes a lot of sense if you think of it: doing refactorings without TDD would be suicidal, Continuous Integration without TDD as well, Testing without simple design is complicated, Simple Design is enforced by TDD, etc.

3.2.2 Scrum

Scrum is a schematic organization of complex product development. It is defined by its creators as an "*iterative holistic framework that focuses on common goals by delivering productive and creative products of the highest possible value*"

This organizational scheme is based on the division of a project into time boxes, called "*sprints*". A sprint can last between a few days and a month (with a preference for two weeks).
Each sprint starts with an estimate followed by operational planning. The sprint ends with a demonstration of what has been completed.
Before starting a new sprint, the team makes a retrospective. This technique analyzes the progress of the completed sprint, in order to improve its practices (Continuous Improvement / Kaizen).
The work flow of the development team is facilitated by its self-organization, so there should be no formal Project Manager but a Team Leader instead with a coaching role more than a management role.

The Scrum process can be represented as follows:



Some more information about scrum is available here : 2.1.3 Scrum.

Working with 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, 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*."

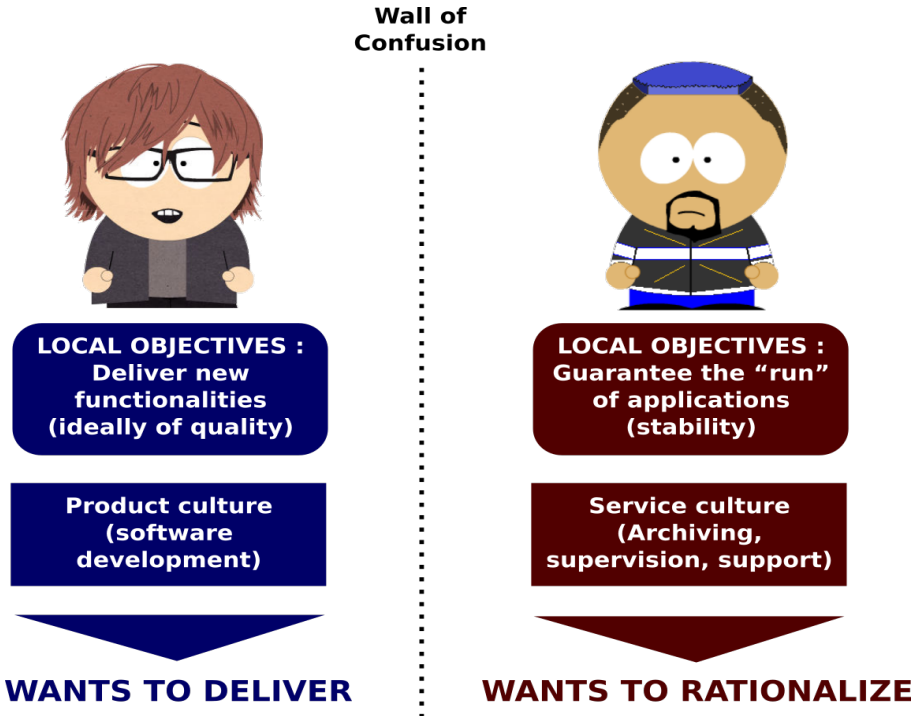
In Scrum these numbers are called **Story Points** - or **SP**.

3.2.3 DevOps

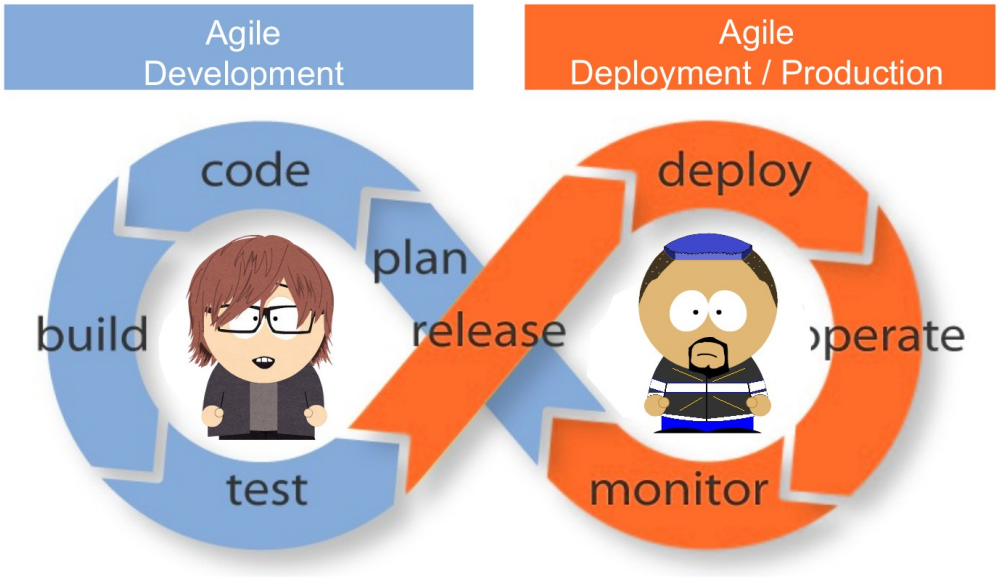
DevOps is a methodology capturing the practices adopted from the very start by the web giants who had a unique opportunity as well as a strong requirement to invent new ways of working due to the very nature of their business: the need to evolve their systems at an unprecedented pace as well as extend them and their business sometimes on a daily basis.

DevOps is not a question of tools, or mastering chef or docker. DevOps is a methodology, a set of principles and practices that help both developers and operators reach their goals while maximizing value delivery to the customers or the users as well as the quality of these deliverables.

The problem comes from the fact that developers and operators - while both required by corporations with large IT departments - have very different objectives.



DevOps consists mostly in extending agile development practices by further streamlining the movement of software change thru the build, validate, deploy and delivery stages, while empowering cross-functional teams with full ownership of software applications - from design thru production support.



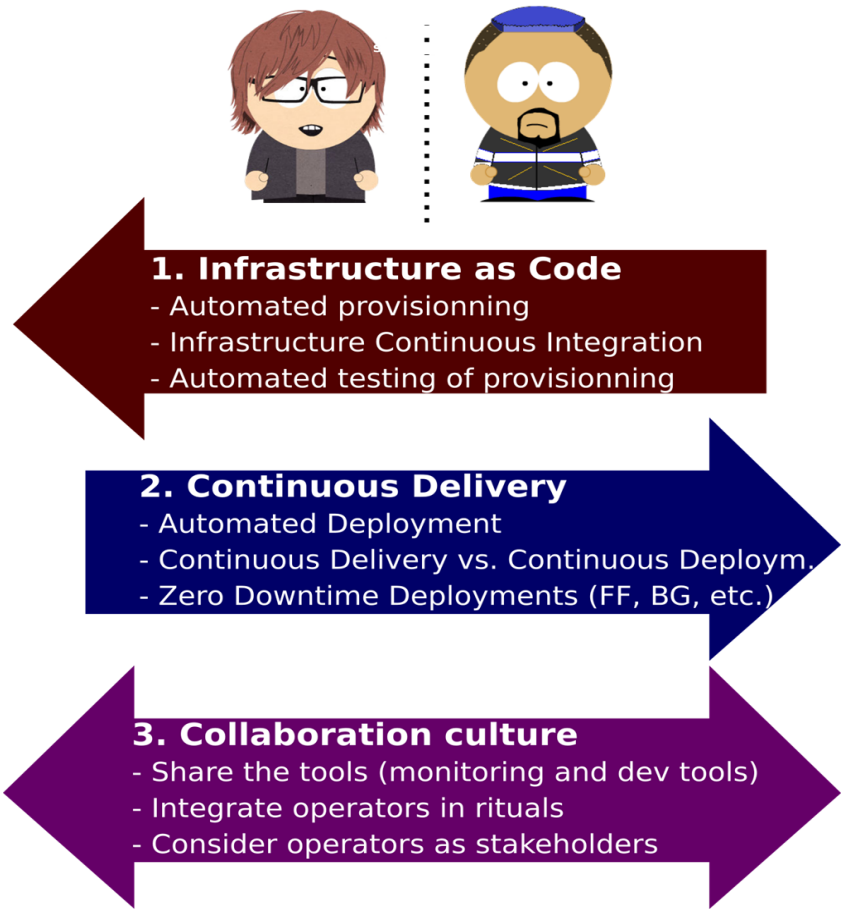
DevOps encourages **communication**, **collaboration**, **integration** and **automation** among software developers and IT operators in order to improve both the speed and quality of delivering software.

DevOps teams focus on standardizing development environments and automating delivery processes to improve delivery predictability, efficiency, security and maintainability. The DevOps ideals provide developers more control of the production environment and a better understanding of the production infrastructure.

DevOps encourages empowering teams with the autonomy to build, validate, deliver and support their own applications.

DevOps is a revolution that aims at addressing the wall of confusion between development teams and operation teams in big corporations having large IT departments where these roles are traditionally well separated and isolated.

So what are the core principles ?

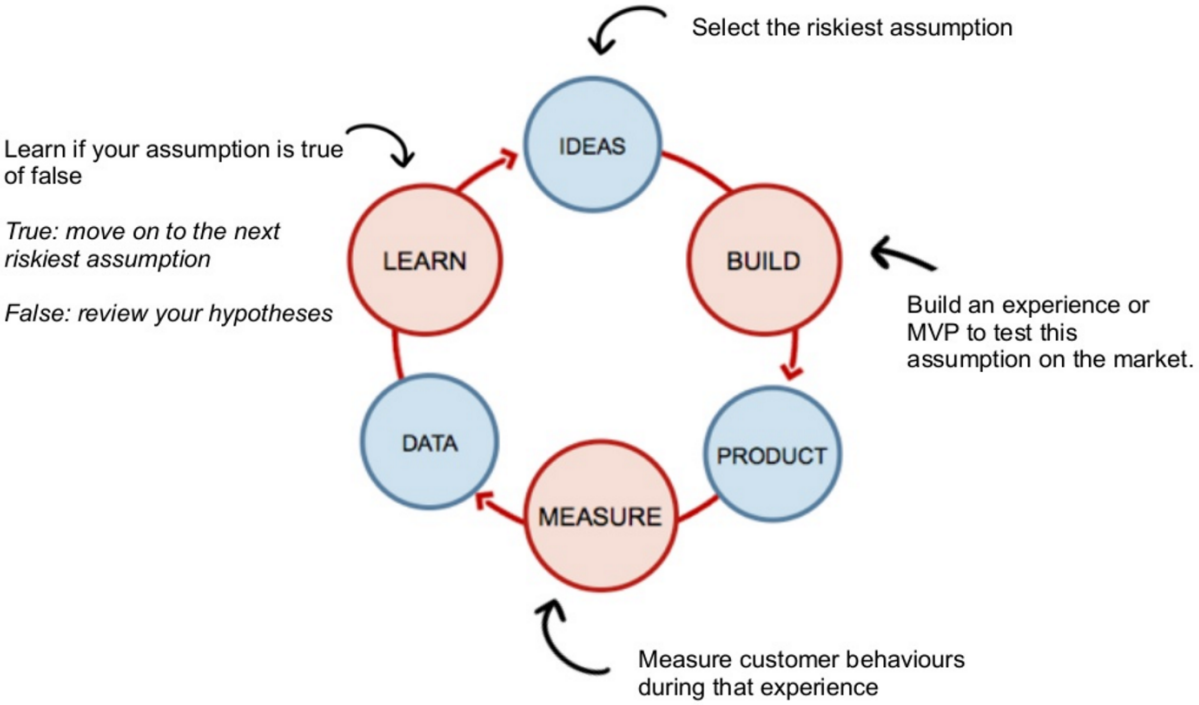


These principles and practices are presented in details in 4. DevOps explained.

3.2.4 Lean Startup

Some years ago, Eric Ries, Steve Blank and others initiated The Lean Startup movement. The Lean Startup is a movement, an inspiration, a set of principles and practices that any entrepreneur initiating a startup would be well advised to follow.

In my opinion, the most fundamental aspect of Lean Startup is the *Build-Measure-Learn* loop.
The fundamental activity of a startup is to turn ideas into products, measure how customers respond, and then learn whether to pivot or persevere. All successful startup processes should be geared to accelerate that feedback loop.



The five-part version of the Build-Measure-Learn schema helps us see that the real intent of building is to test "*ideas*" - not just to build blindly without an objective. The need for "data" indicates that after we measure our experiments we'll use the data to further refine our learning. And the new learning will influence our next ideas. So we can see that the goal of Build-Measure-Learn isn't just to build things, the goal is to build things to validate or invalidate the initial idea.

The four steps to the Epiphany

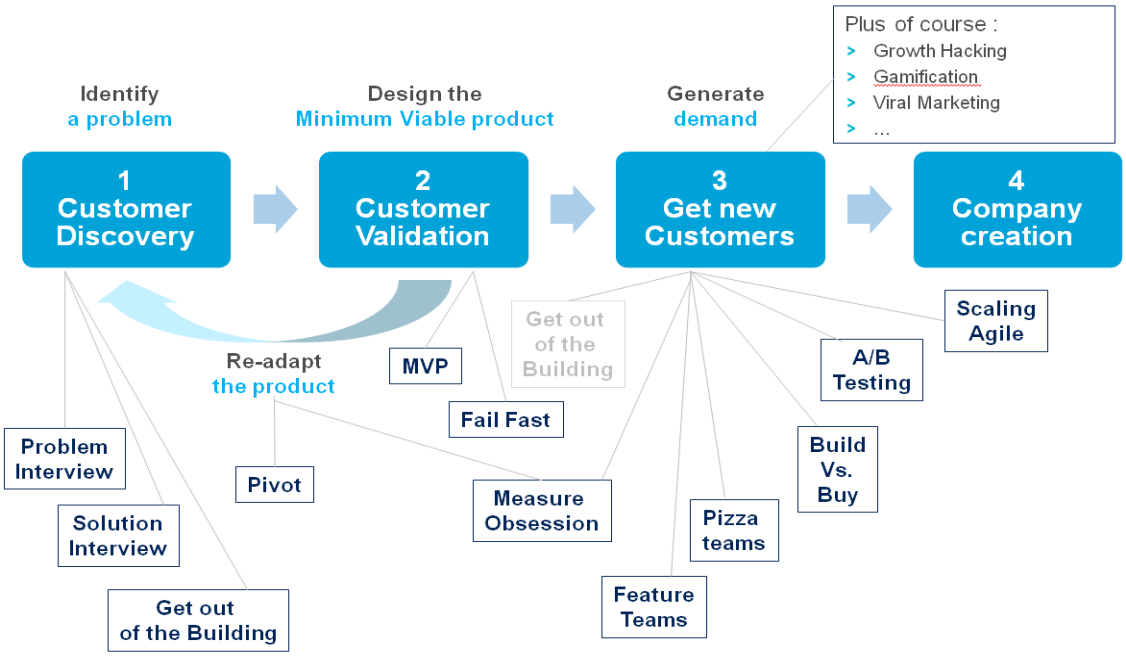
Shortly put, Steve Blank proposes that companies need a **Customer Development process** that complements, or even in large portions replaces, their *Product Development Process*. The *Customer Development process* goes directly to the theory of Product/Market Fit.
In "*The four steps to the Epiphany*", Steve Blank provides a roadmap for how to get to Product/Market Fit.

The four stages the *Customer Development Model* are: customer discovery, customer validation, customer creation, and company creation.

1. **Customer discovery**: understanding customer problems and needs

2. **Customer validation**: developing a sales model that can be replicated
3. **Customer creation / Get new Customers**: creating and driving end user demand
4. **Customer building / Company Creation**: transitioning from learning to executing

We can represent them as follows:



the most essential principles and practices introduced and discussed by *the Lean Startup* approach are added to the schema above.

These principles and practices are discussed in length here : 5. The Lean Startup - A focus on Practices.

3.2.5 Visual Management and Kanban

Visual Management is an English terminology that combines several *Lean management* concepts centered on visual perception. The aim is to put the information and its context in order to make the work and the decision-making obvious.

Visual Management is an answer to the well known credo "*You can't manage what you can't see*". It finds its root in *Obeya War Rooms*:

(Source : http://alexsibaja.blogspot.ch/2014/08/obeya-war-room-powerful-visual.html)

Obeya (from Japanese "*large room*" or "*war room*") refers to a form of project management used in many Asian companies, initially and including Toyota, and is a component of *lean manufacturing* and in particular the Toyota Production System. During the product and process development, all individuals involved in managerial planning meet in a "*great room*" to speed communication and decision-making. This is intended to reduce "*departmental thinking*" and improve on methods like email and social networking. The Obeya can be understood as a team spirit improvement tool at an administrative level.

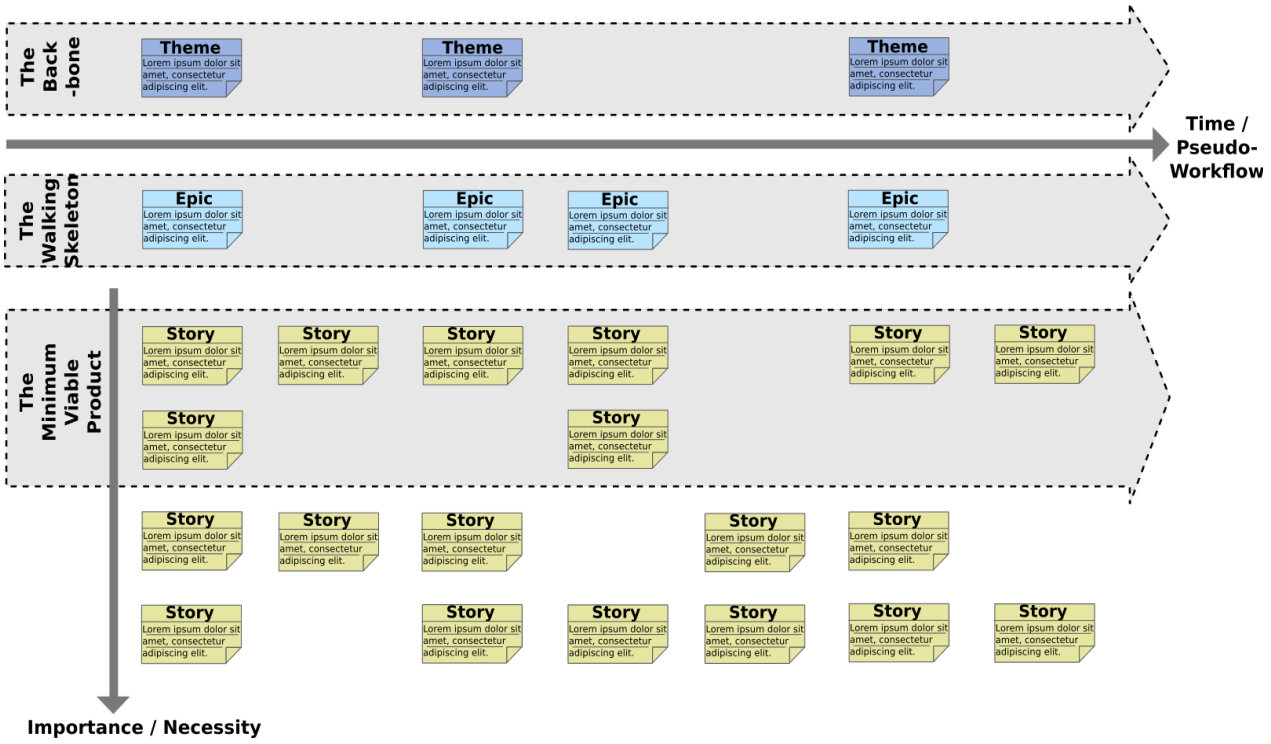
Nowadays, visual management is very much linked to *Lean Management* and Lean Startup, but IMHO it's really a field on its own. In the field of **Agile Planning,** I believe that Visual Management with sound tools and approaches is not optional. At the end of the day, as we ill see, a good Project Management tool is a tool than enables anyone in the company to understand what is achievable in a given time or what time is required to deliver a given scope within a few minutes. And **nothing competes with Visual Tools** in this regards.

I will introduce here the fundamental tools I believe an Agile team should consider when it comes to Visual Management:

3.2.5.1 Story Map

The purpose of the Story Map is that arranging user stories into a helpful shape - a map - is usually deemed as most appropriate.
A Story Map is a visual management tool aimed at presenting the situation of the Software or the features to be implemented in a clear and graphical way. A Story Map is composed by user stories (see below).

A small story map may look like something like this:

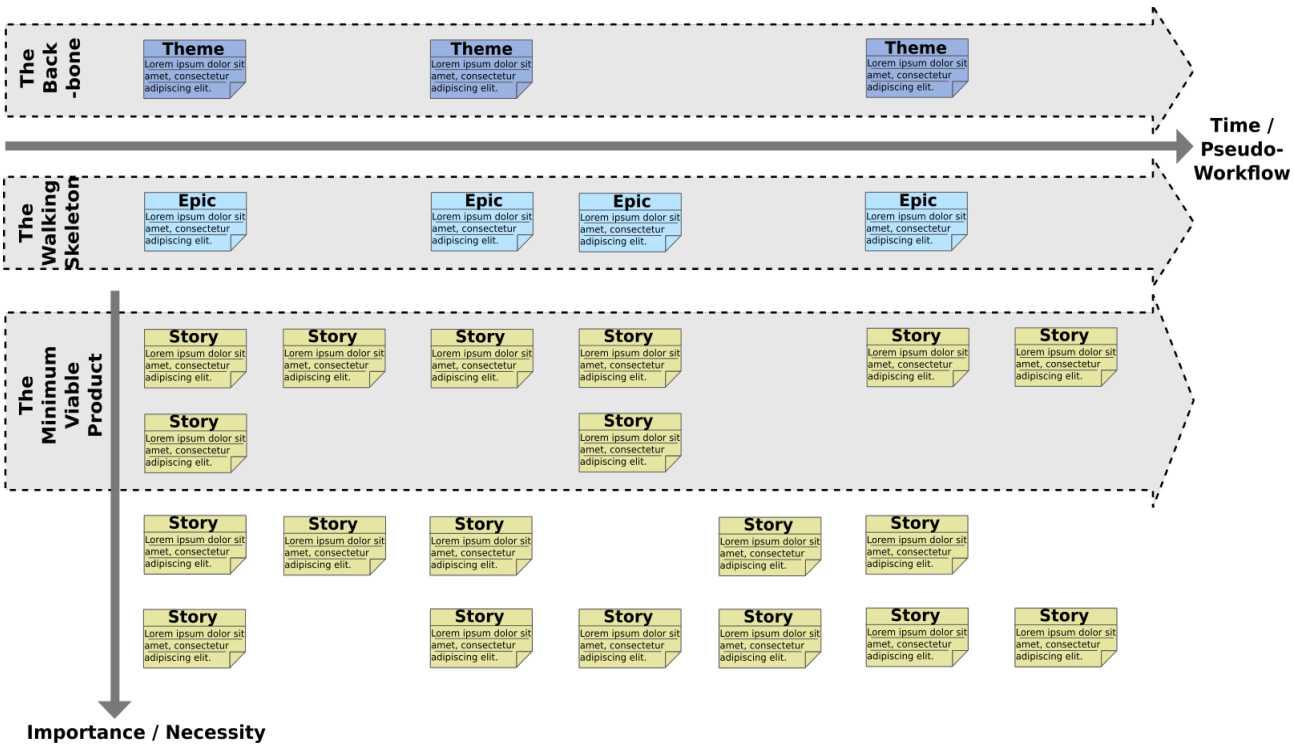


At the top of the map are "big stories." We call them **themes**. A theme is sort of a big thing that people do - something that has lots of steps, and doesn't always have a precise workflow. A theme is a big category containing actual user stories grouped in **Epics**.

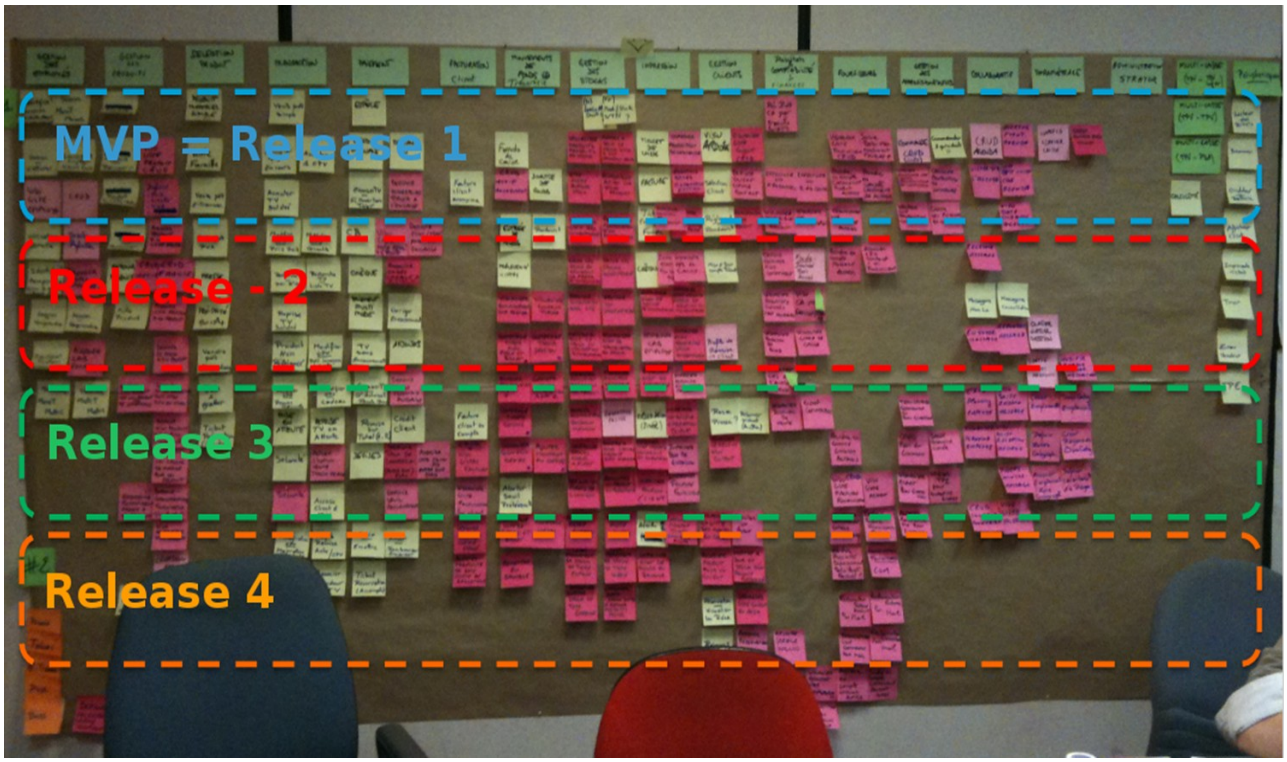
Epics are big user stories such as the one mentioned in example above. They usually involve a lot of development and cannot be considered as is in an actual product backlog. For this reason, Epics are split in a sub-set of **stories**, more precise and concrete that are candidate to be put in an actual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

I presented more information on Story Maps here : 2.3.2 Story Maps.
For the moment, let's just remember that there is an important notion of **priority** on the vertical scale: the lower a story, the lesser its priority.
There is also a les obvious notion of priority horizontally: stories on the left should be implemented first since they have a greater value than the stories on the right, but all of that of course with respect of the more important vertical priority.
Long story short: the development team needs to implement all the stories of a row, from left to right, before it can consider the stories of the next row.

An pretty good and straightforward example of a Story Map related to an *email client application*:



And a real world example built during an real life Workshop:

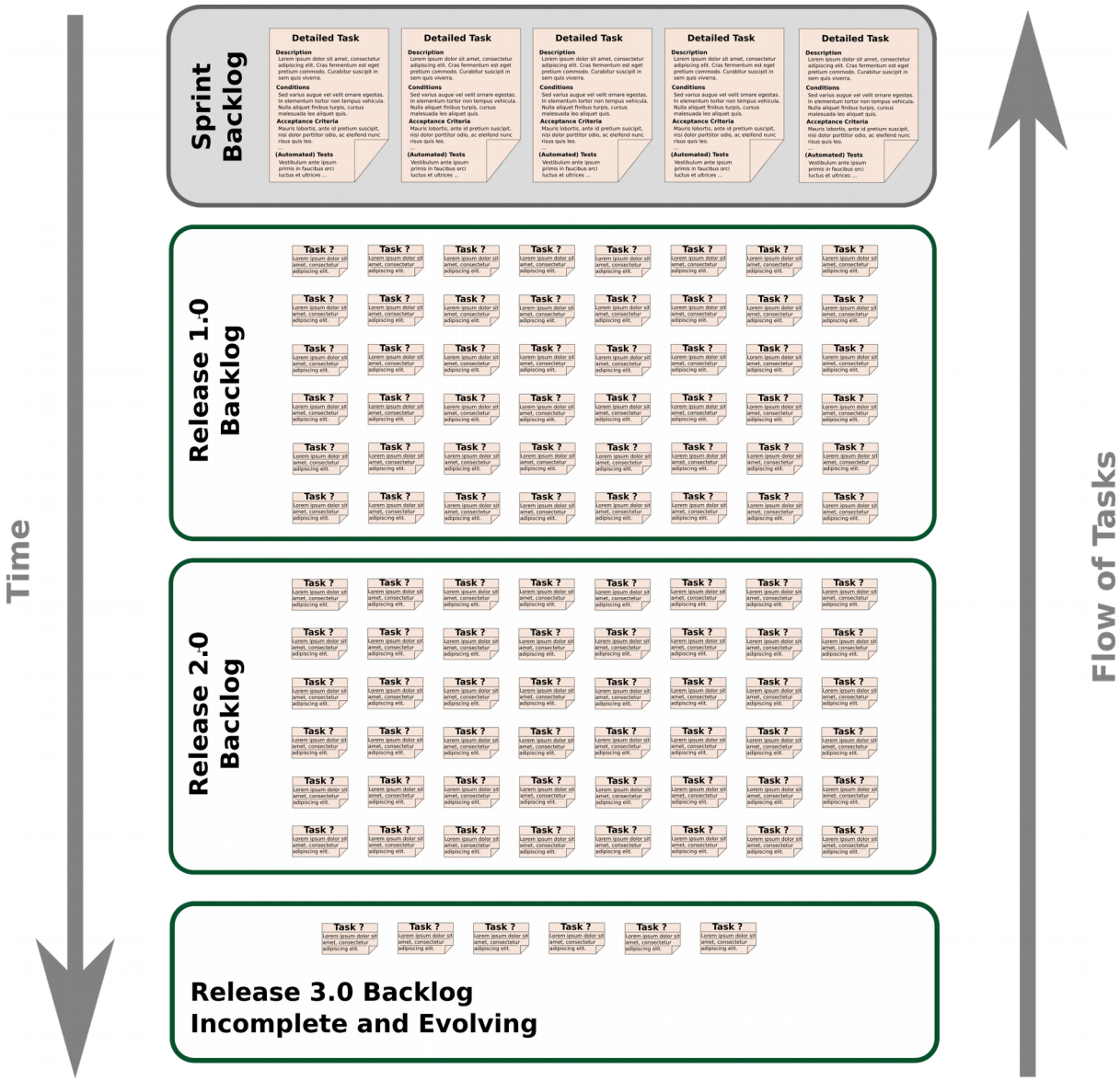


(Copyright OCTO Technology / *Unfortunately I haven't been able to recover the source*)

A story Map is usually a visual tool, laid down on the wall of a shared meeting room or even the development team open-space. Distributed teams may consider digital tools but a physical, real and visual map on a wall is way better.

3.2.5.2 Product Backlog

The product backlog is the tool used by the Development tool to track the tasks to be implemented. These development tasks should be linked to a User Story on the Story Map.
As such, the product backlog should be seen as a much more detailed and technical version of the Story Map.



The product backlog shows the same releases than the Story Map. The development tasks in the current sprints should have a more detailed form than the development

tasks not analyzed yet during *Sprint Planning*.
In a general way, the product Backlog should be kept synchronized with the Story Map and the reverse is true as well. Every User Story on the Map is broken down in development tasks in the Product Backlog and all tasks in the backlog should be attached to a User Story on the Map.

Their difference is as follows:

- **Story Map** : The Story Map is a management tool. It is a visual tool used by the *Product Management Team* to drive the high level development of the product and to defined releases and priorities.
- **Product Backlog** : The product Backlog is a technical project management tool, not a visual management tool. Its is usually supported by a digital tool (such as *Jira* or *Redmine*) and aims at organizing at a fine level the development team activities.

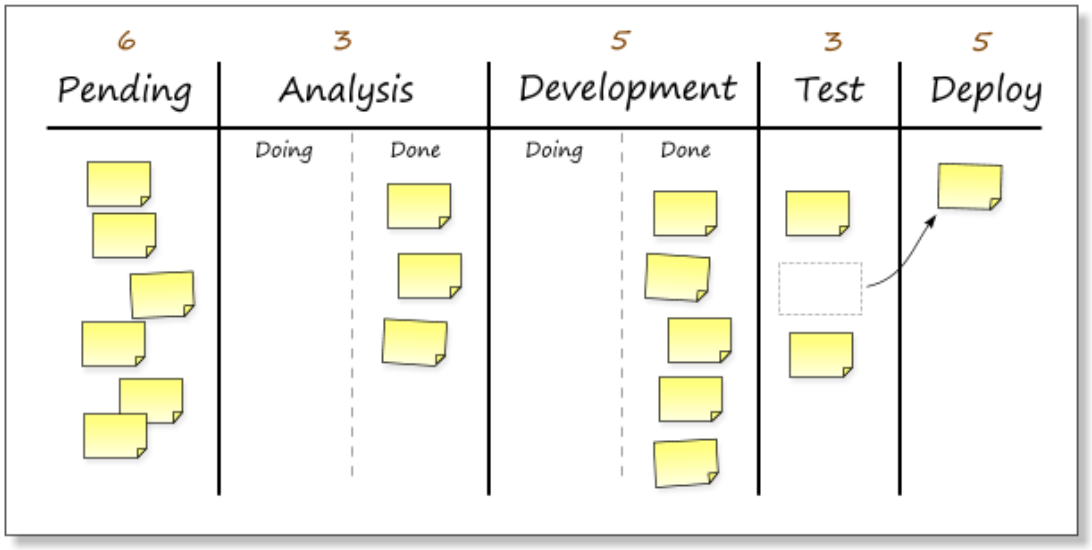
Some important constraints should be noted right away:

- Each and every developer activity, not matter how quick and small, should be well identified by a development task in the product backlog.
- Each and every development task should be linked to a User Story on the Story Map. I cannot stress enough how much this is important.

3.2.5.3 Kanban Board

Kanban is 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.



Kanban boards can be mixed with Story Maps to follow the development of the tasks scheduled for next releases as far as their delivery on the current development version of the product.
In this case, the left-most column of the Kanban board becomes the Story Map containing the Stories to be developed while the right-most column of the Kanban board contains the User Stories identifying features already provided by the product.

I myself call such a mix of Story Map and Kanban a **Product Kanban Board**.

An real-world example of such a mix of Story Maps and Kanban boards could be as follows:

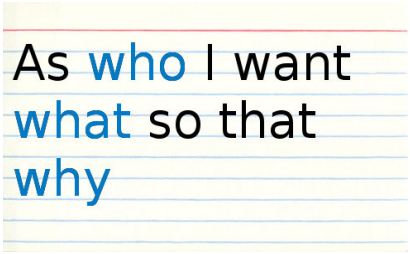
3.2.5.4 User Stories

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:

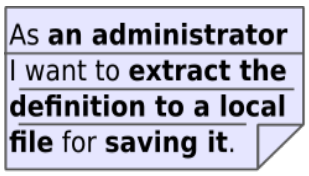
As a <type of user>, I want <some goal> so that <some reason>.

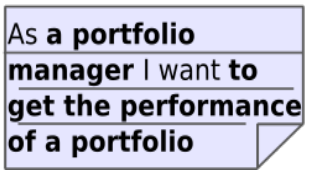
User stories are often written on sticky notes and arranged on walls or tables to facilitate planning and discussion.
As such, they strongly shift the focus from writing about features to discussing them. In fact, these discussions are more important than whatever text is written.

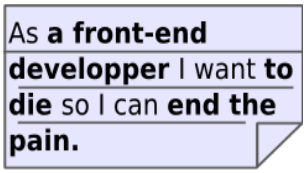


It's the product owner's responsibility to make sure a product backlog of agile user stories exists, but that doesn't mean that the product owner is the one who writes them. Over the course of a good agile project, you should expect to have user story examples written by each team member.
Also, note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







User Stories are used to track existing features as well as features to be developed on a mix of Story Map and Kanban, the **Product Kanban Board**.

3.3. Principles

Having covered the fundamentals, we will now go through the principles required for **Agile Planning** and see how the principles and practices introduced in the previous section should be used to achieve *reliable forecasts and planning* with Agile methodologies.

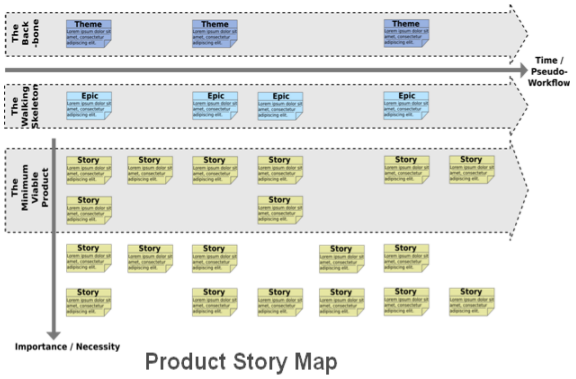
We should now discover:

- **The tools**, mostly visual management tools that the organization should adopt.
- **The Organization** to be put in place with required roles and committees.

- **The processes** that should be respected and that will lead to accurate estimations and forecasts.
- **The Rituals** supporting the processes.
- **The Values** the team has to embrace to successfully run the processes and deploy the required practices.

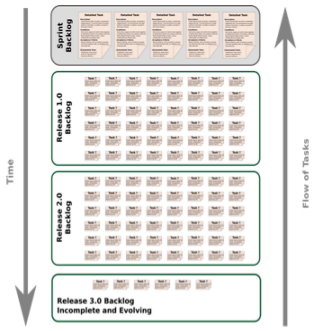
3.3.1 The tools

The tools that the organization should adopt are as follows:



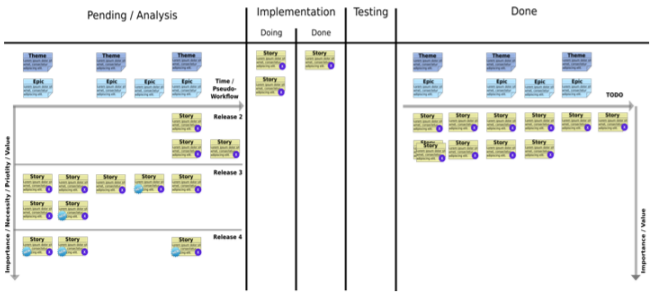
Product Story Map

- Product Management Tool
- Visual Tool



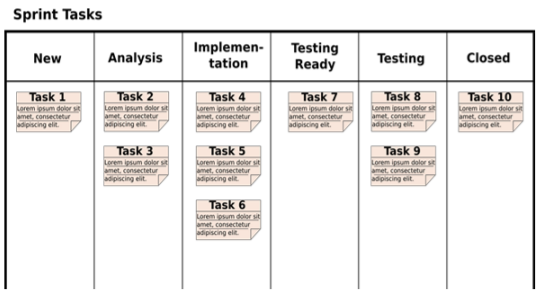
Product Backlog

- Development Team Management Tool
- Digital Tool (not visual)
  - Jira, Redmine, etc.



Product Kanban Board

- Project Management Tool
- Visual Tool



Sprint Kanban

- Sprint Management Tool
- Visual Tool

I believe that I introduced these tools in length in the section 3.2.5 Visual Management and Kanban so I won't be adding a lot. We will see in the next section related to processes how these tools are used and how they complement each other by addressing different needs.

3.3.2 The Organization

The organization to put in place consists in identifying **roles** as well as **committees and teams**.

3.3.2.1 Required roles

The required roles are as follows:

- **Team Leader** : The Team Leader animates the Team rituals (such as Sprint Planning, Sprint Retrospective, Daily scrum) and acts as a coach and a mentor to the development team. He is not a manager, he is a leader (Lead by Example, Management 3.0, etc.). He also represents the development team in other rituals (PMC).
  At the end of the day, the Team Leader should not be made responsible for neither the team successes nor the team failures, the whole team should be accountable for this.
  If the team leader is solely responsible for the Team's performance, then we will be tempted to shortcut quality or mess some rituals to speed up the pace and successfully respect some artificial deadline or else. When that is the case, the team requires a *Scrum Master* who should guarantee the Scrum rituals and processes are well respected.
  In my opinion it makes a lot more sense to avoid such situation by making sure everyone in the team is accountable for the team performance and also responsible for the proper respect of the defined Agile processes and rituals. In this case, the Team Leader becomes an arbitrator, a facilitator, a coach and a support, not a manager. At the end of the day, management is too important to be left to managers ;-)
- **Architect** : The Architect (or architects) should be the most experienced developer(s), the one(s) with the biggest technical and functional knowledge. There can be several architects, a lead architect, a technical architect, etc. This doesn't really matter.
  The important thing is that the architect should be entitled to take architecture decision by still referring to the whole team as much as possible. The architect leads the Architecture Committee where architecture decisions are taken.
  The architect, with the help of the tech leads, provides guidance and support to developers. he is also responsible to check the Code Quality, leading the code reviews, and ensure the sticking to Code conventions, etc.
- **Tech Leads and Developers** : The tech leads and developers form the core of the development team, they eventually develop the software.

Tech Leads are coaches and supports to developers and represent them in the Architecture Committee.

- **Product Owner** : The product Owner represents the stakeholders and drives priorities in good understanding with the market and customer needs. He is not a leader, he is an arbitrator and acts as the bridge between the business requirements and the development team.
  I can only recommend the reader to watch the magnificent video "Agile Product Ownership in a Nutshell" from Henrik Kniberg.
- **Business representatives** : Business representatives (sales, customers, etc.) have to be involved in the Product Management Committee by the product Owner whenever required.

Why bother ?

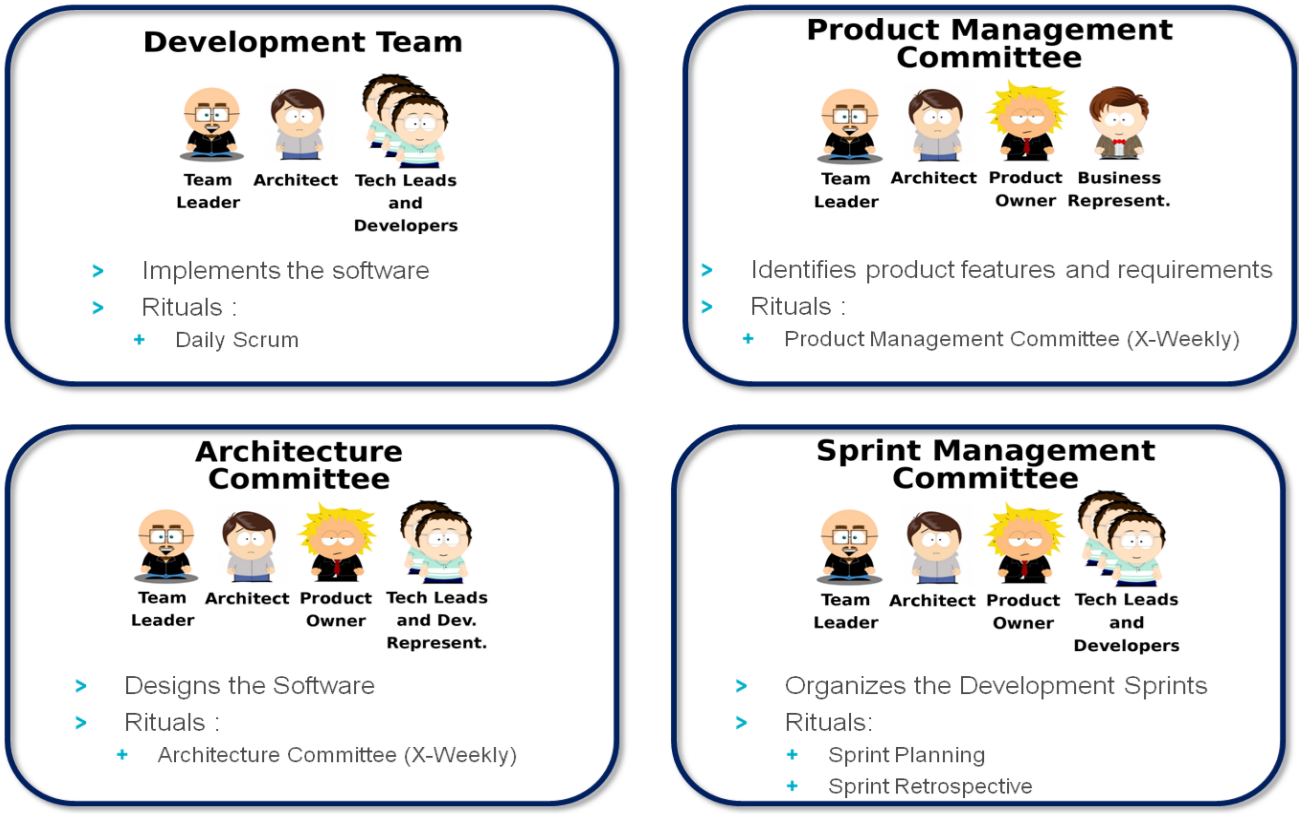
Roles are required mostly for two reasons : **efficiency** and **focus**:

- **Efficiency**: roles are required to avoid having the whole organization meeting all the time at every meeting for every possible concern.
- **Focus**: every role owner should acts as required by his role and put himself in the right mindset for every ritual. Rituals are eventually a role playing game. Roles are not functions ! We are not speaking hierarchy here, it's more a question of role play : when someone is assigned a role, his objective is to act and challenge the matters being discussed in correspondence with his role !

As an important note, roles can well be shared. A same co-worker can have multiple roles if required, even though it would be better to avoid this.

3.3.2.2 Required Committees and teams

Required committees and teams are as follows:



- **Development team** : The development team is responsible to develop the software. It is composed by Developers, Tech Leads, Architects and the Team Leader. At the end of the day, they're all developers and even the Team Leader should be able to spend a ratio of his time developing the Software (Lead by Example). Its essential ritual is the daily scrum every day.
- **Product Management Committee** : The Product Management Committee - or PMC - is composed by the Development Team Leader, The Architect(s) and The Product Owner. The Product Owner should convoke business representatives as required. The PMC is responsible for identifying the new features to be added to the product and prioritize them. It should take place every week or every two weeks at most.
  The PMC identifies new features as User Stories and Uses the Story Map to track them and prioritize them. Priorities are redefined and adapted as Stories Estimations (in Story Points) are refined. This process is explained later.
  Priorities should be set in respect to **the value** and **the cost** (in SP) of each and every story.
- **Architecture Committee** : The Architecture Committee is composed by the Team Leader, The Architect(s), The Product Owner, the Tech Leads and representatives of the Development team.
  The Architecture Committee is responsible to analyze user stories and define

Development Tasks. Every story should be specified, designed and discussed. Screen mockups if applicable should be drawn, acceptance criteria agreed, etc.
Since the Architecture Committee is also responsible for estimating Stories, it's important that representatives of the Development Team, not only the Tech Leads and the Architects, but simple developers as well, take part in it. Ideally, there should be a rotation and at every meeting a different couple of developers should be convoked. This is required to have everyone agreeing on the estimations.
The Architecture Committee should take place every week or every two weeks at most as well and ideally not long after the PMC.

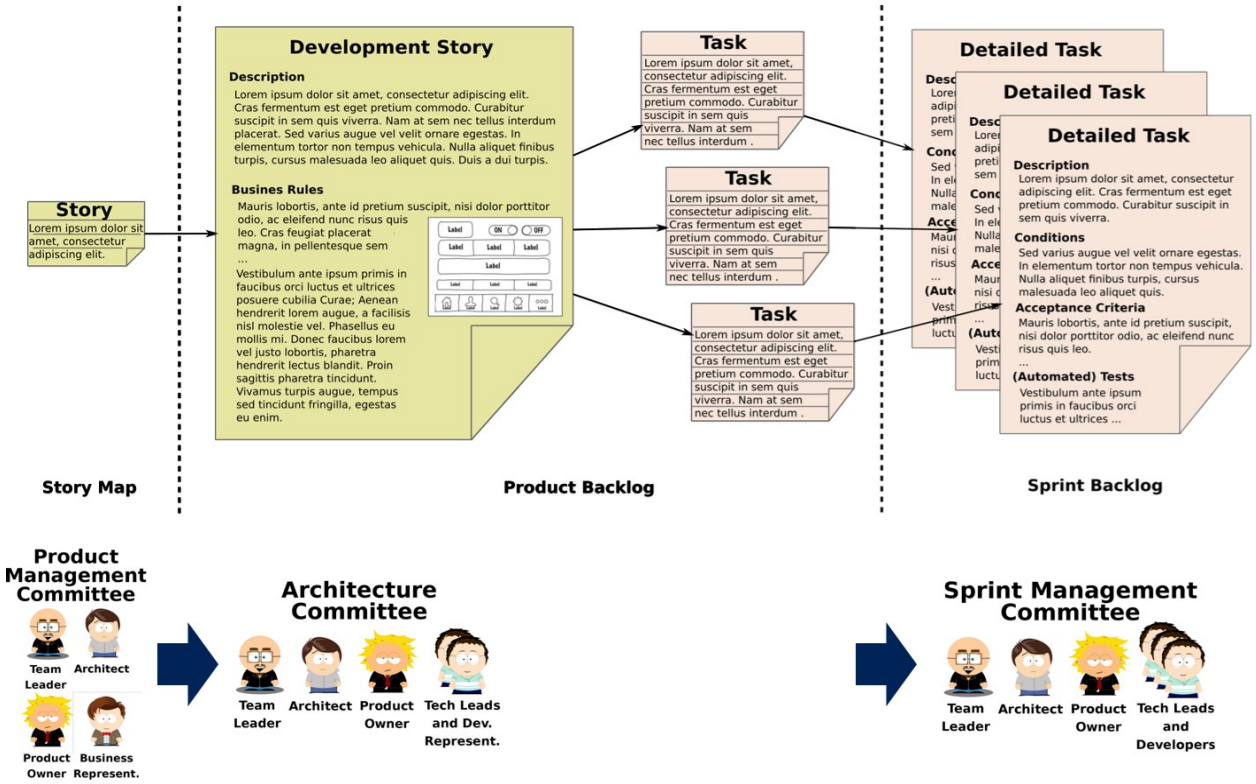
- **Sprint Management Committee** : The Sprint Management Committee is basically composed by the Development Team plus the Product Owner.
During Sprint Planning, the Sprint Management Committee discusses the implementation concerns of the tasks specified by the Architecture Committee and challenge the estimations if required. The Development Tasks defined by the Architecture Committee are detailed as much as possible.
During Sprint retrospective, the Sprint Management Committee discussed the issues and drawbacks encountered during former sprint and agrees on an action plan to address them.

3.3.3 The Processes

I will be presenting now the various processes that are required to achieve the ultimate goal of Agile Planning : reliable forecasts and planning.

3.3.3.1 Design Process

The *Design process* consists in breaking a *User Story* identified by the PMC into *Development tasks* that developers can understand and work on.
It can be illustrated as follows:



A. Identification of User Stories

The **PMC** produces a **User Story** laid down on the Story Map.

B. From User Stories to Development Stories

The **Architecture Committee** analyzes every new story and for each of them it creates a **Development Story** on the Product Backlog.

Such a *Development Story* is not anymore a simple post-it in a Story Map, it is a *digital User story* created in the backlog management tool such as Jira or Redmine. The Development Story is specified and design. It should contain:

- The initial user story from the Story Map as it was expressed at that time.
- A complete description of the purpose and intents 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 precise and exhaustive 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.
- Acceptance criteria (functional and non-functional) and a complete testing procedure.

C. From Development Stories to Development Tasks

The **Architecture Committee** also breaks the **Development Story** down in several **Development Tasks**.
Development tasks should be split by logical or functional units or layers. For instance, one task could be related to the GUI while another one could be related to the database changes, etc. But if it is possible, it is always better not to split them by layer but rather vertically by sub-feature.
What should never be done is splitting a Story in tasks by the type of job, for instance development, unit test, integration tests. That should never ever be done. A developer, or a couple of developers should always implement a sub-feature entirely, with all the required tests, functional tests, migration scripts, etc.

D. From Development Tasks to Detailed Tasks

The **Sprint Management Committee**, during *Sprint Planning* recovers all these **Development Tasks** and analyzes them further.

The questions to be answered at this time are:

- Are all the information provided by the Architecture Committee clear enough or are some precisions required ?
- Is there any unforeseen impact on other parts of the software ?
- Is there any tool or specific environment setup or configuration required to implement these tasks ?
- etc.

Specifically the developers that were not present at Architecture Committee when a task has been designed should challenge it and make sure they understand not only what need to be done but really also how to do it precisely.
At this stage, the new findings should lead to a refinement of the initial estimations agreed by the Architecture Committee.

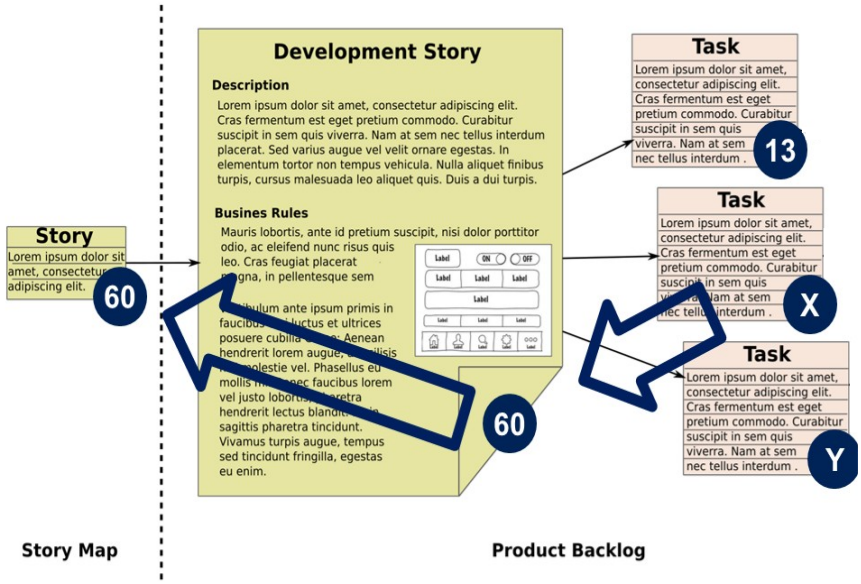
3.3.3.2 Estimation Process

What we want eventually, is a Story Map containing estimations for all the Stories that have been analyzed by the Architecture Committee.
The result we want to achieve here can be represented as follows:

Each and every story that has been broken down by the Architecture Committee and created in the Product Backlog is clearly identified: it has an estimation expressed as a total number of Story Points.
That number corresponds to the total of the estimations in SP of the individual *Development Tasks* underneath.

A. Initial Estimations

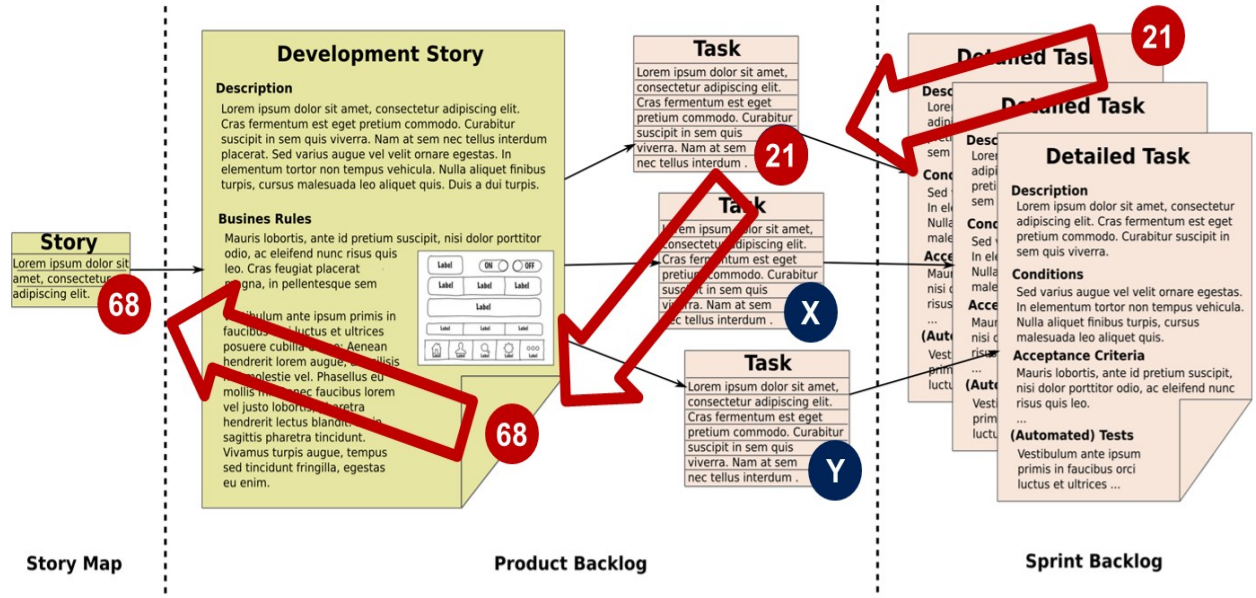
At this stage, The **Architecture Committee** is in charge of the Initial Estimations.
After a Story has been broken down in tasks, each and every of these tasks is estimated by the Committee using the *Planning Poker* approach.
The sum of the estimations of every individual tasks is reported on both the Development Story (Product Backlog) and the User Story (Story Map):



B. Refined Estimations

When the **Sprint Management Committee** recovers the **Development Tasks** to refine them, there might be new impacts discovered, new unforeseen refactorings required, etc.

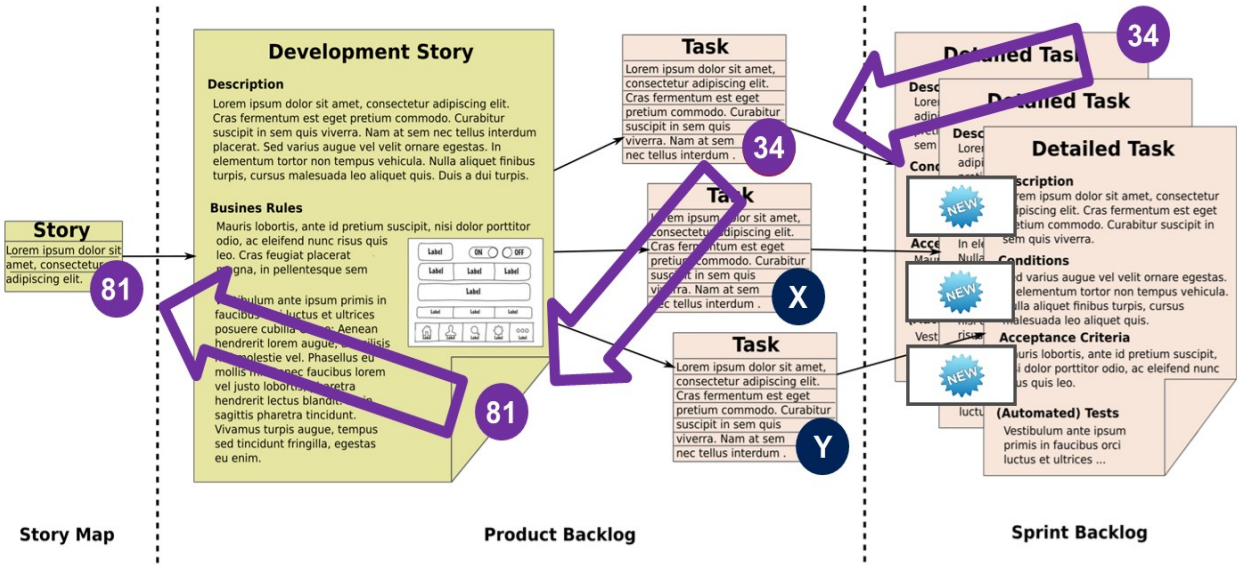
The Sprint Management Committee should challenge the initial estimations with their new findings and adapt the estimations accordingly.
Again, these new *Refined Estimations* should be reported on both the Development Story (Product Backlog) and the User Story (Story Map):



C. Final Estimations

Eventually, during the sprint, it can happen that a developer discovers that a task will take a bigger time than expected, or, in the contrary, a much shorter time. Reporting such changes in estimations at this very late stage is maybe not important for Scrum, since the sprint is already filled, but it's important for both the Sprint Management Committee and the Architecture Committee to be notified about them in order to improve the way they do estimations.
As part of Continuous Improvement (Kaizen), the Architecture Committee needs to identify where the gap comes from and try to have more accurate estimations next time.

So even at this stage, when a developer discovers gaps or shortcut, it's important that any impact in terms of estimation is reported as far as to the Story Map:



Why bother ?

The management tool is the story map, not the product backlog. The product backlog is a technical tool to organize the development activities. It's not a management tool.

The Product Management Committee should be able to decide about priorities using solely the Story Map. In addition, it should be possible to forecast a delivery date using solely the Story Map.
For this reason, the Story Map should contain as up to date as possible estimations.

Everyone in the company should be able to take is little calculator, go in front of the story map and know precisely when a task will be delivered.
We'll see how soon !

What about updating estimations after the task has been completed and we know how much time we spent on it ?

One needs to understand what we're trying to achieve here.

We're trying to continuously improve our ability to come up with accurate and reliable estimations based on the information we have. When we estimate tasks at *ARCHCOM* or *Sprint Planning*, we only have analysis information at our disposal, we have no clue about any post-implementation information such as the actual time that will be spent on the task.
As such, while it is very important to improve our ability to estimate using analysis information (as done at ARCHCOM), it makes no sense to update estimations after implementation since actual implementation time is an information we will never have before implementing the task.

Again, we want to improve our ability to estimate using the information we have. And actual implementation time is an information we don't have so it's useless in regards to improving the estimation process and as such doesn't trigger any estimation update.

In addition, the estimation process is a comparison game, not an evaluation game (or less). An Estimation in SP should have no clear relationship with actual implementation time, for many reasons, among them the fact the different developers have different capacity. A 10 SP task is always a 10 SP task, for every developer. But it may well represent 4 days of work for a junior developer and 2 days of work for a senior developer.
This aspect is a very important reason behind the rationality to think in terms of SP instead of Man/Days. And of course SP should be a measure of the whole team capacity, not individuals.

This is why we don't bother updating estimations after actual implementation. Nevertheless, we should still use that knowledge to improve our estimations, but actually trying to update the estimation in SP makes no sense.

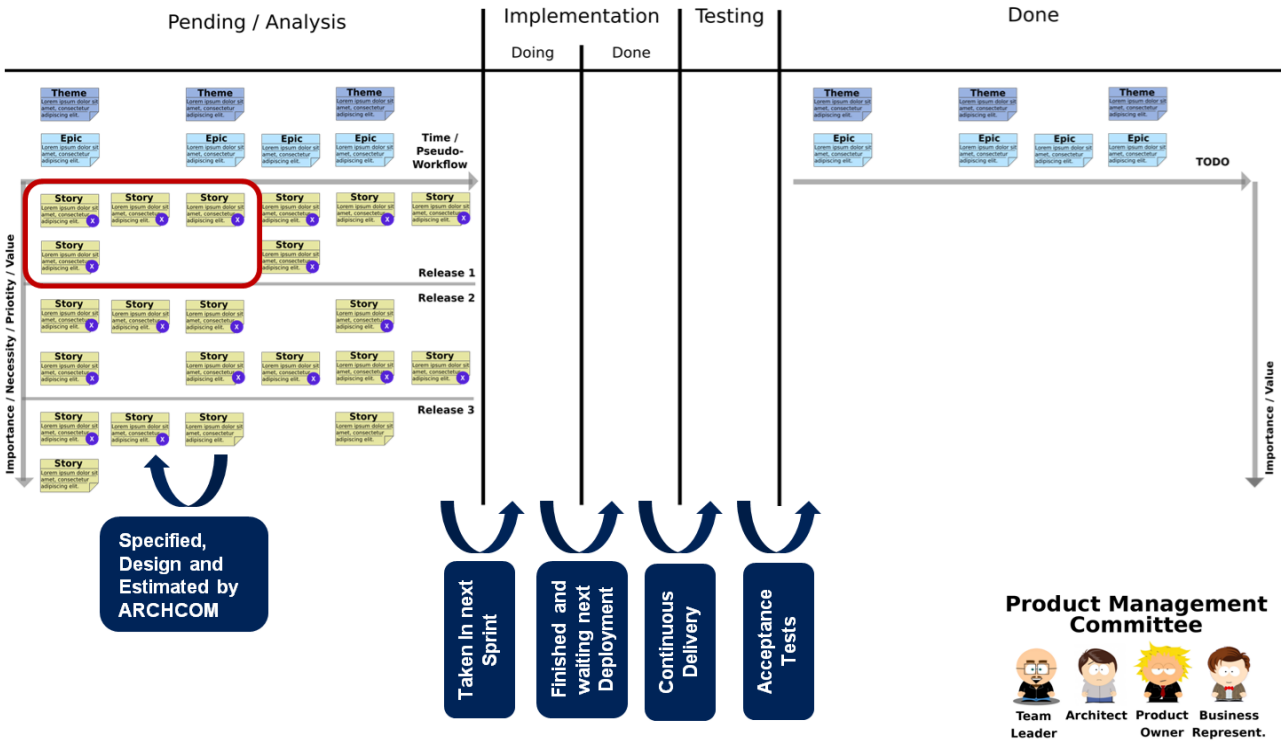
3.3.3.3 Product Kanban Board Maintenance Process

Maintaining the *Product Kanban Board* (Mix of Story Map and Kanban Board) as up to date as possible with latest activities of the development team as well as the latest estimations is important.
Again, The *Product Kanban Board* is the only tool that should be required by the Product Management Committee to come up with estimations and forecasts.

We will now see how this *Product Kanban Board* should be maintained throughout the sprints and how it is used.

A. Initial Stage: before the first sprint of the nest release

We start with a Board of the following shape:



The boxes in blue indicate how a *User Story* is moved across the board when it advances in the analysis and development process:

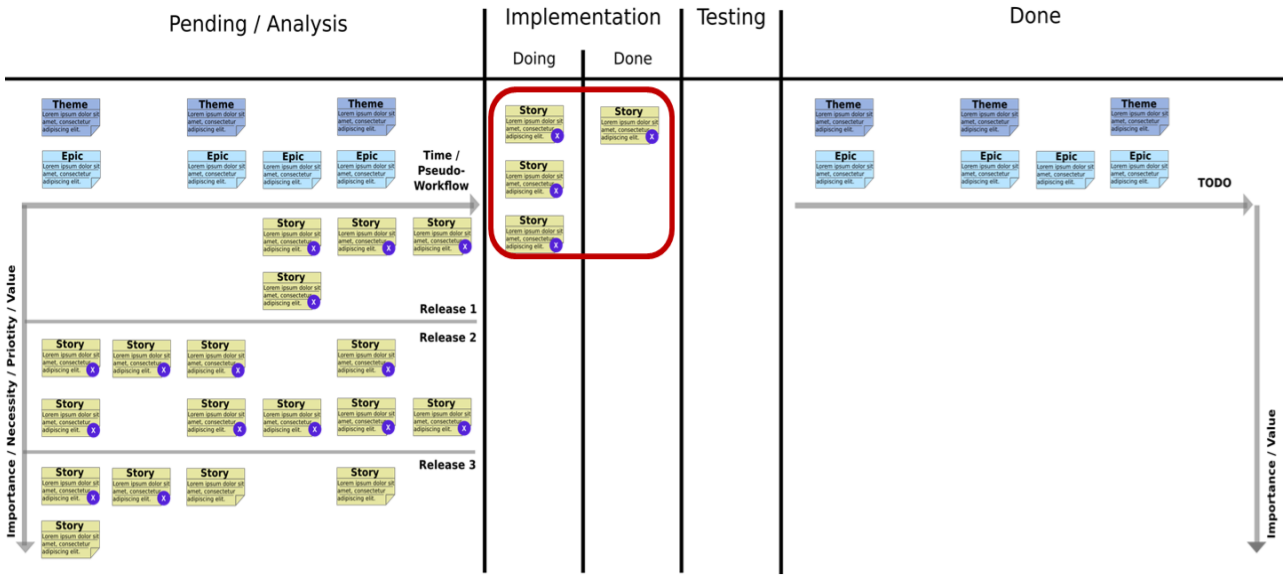
- First, when the Architecture Committee has done analyzing and breaking down the Story, the estimation it came up with is reported on the User Story in the violet pellet.
- Then, A Story is moved to **Implementation / Doing** when a first of its development tasks is being implemented in the current sprint
- It is moved to **Implementation / Done** when the last of its development tasks is done being implemented (meaning completely done : with automated tests, IT tests, etc. At this stage it's simply waiting the next *continuous deliver* build to be available on Test environment for acceptance tests.
- When the *Continuous Delivery* build has been executed, the Story is moved to **Testing**.
- When the product Owner either tested the Story (or delegated such tests) and accepts the results, the Story is moved to **Done**

The Story Map on the left is a pretty standard Story Map, where releases are identified.

The Story Map on the right, on the other hand, drops the notion of releases completely. It identifies the features as they are available as a whole in the current development version of the product, regardless of both past releases and releases to come.
A story identifying a new feature is simply added to it to capture the fact that the feature is now available on the development version.
On the other hand, a story identifying a modification of an existing feature should be **merged with the original story**, potentially leading to a new story, corresponding to the new way of expressing the feature.

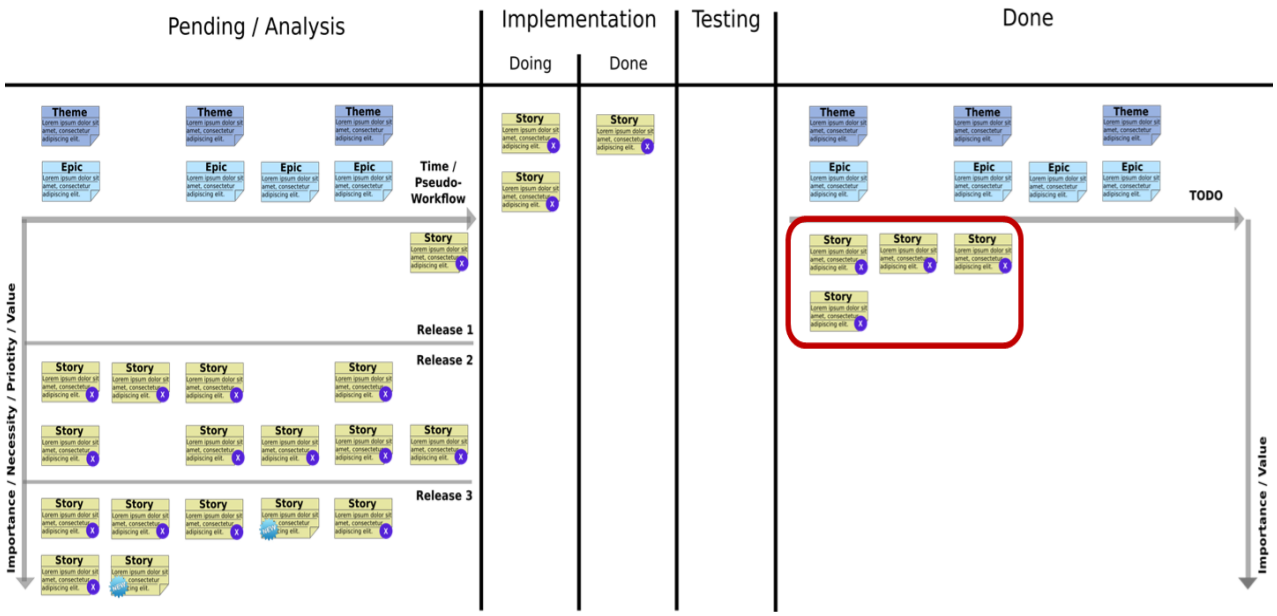
B. During the first sprint

During the first sprint after this initial stage, the Kanban board in the middle identifies the Stories that are being worked on and their status:



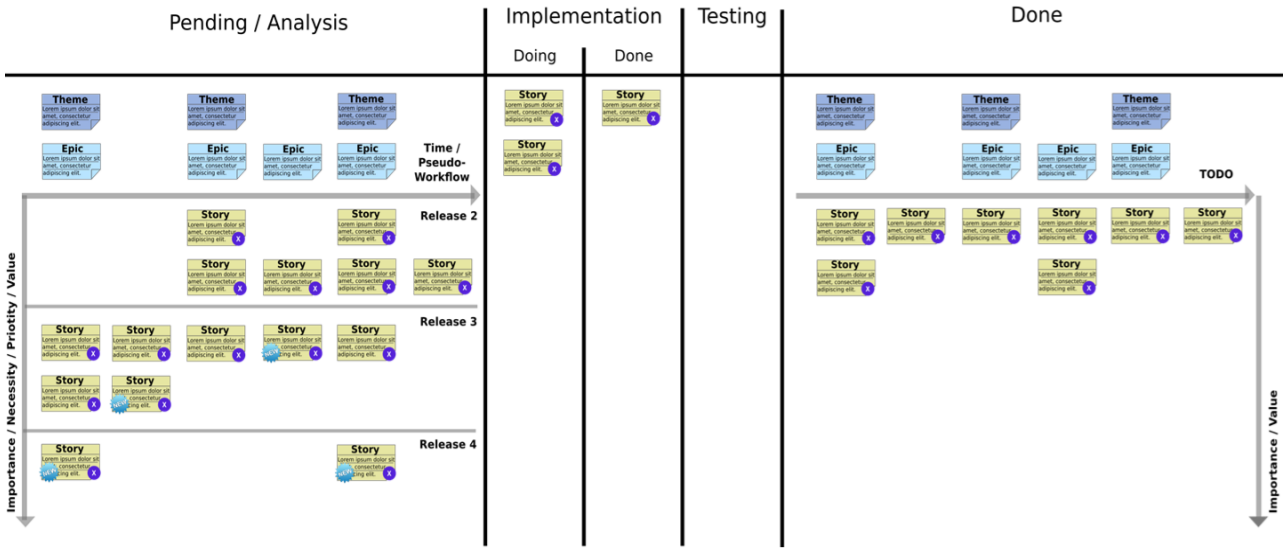
C. During the second sprint

After first sprint, developed stories are put on the Story Map on the right and a next set of Stories are being developed:



D. After the first release

After the first release, we can see that all the tasks from the first release of the Story Map on the left have been moved to the Story Map on the right.
The Story Map on the left is adapted and the next releases are shifted up.



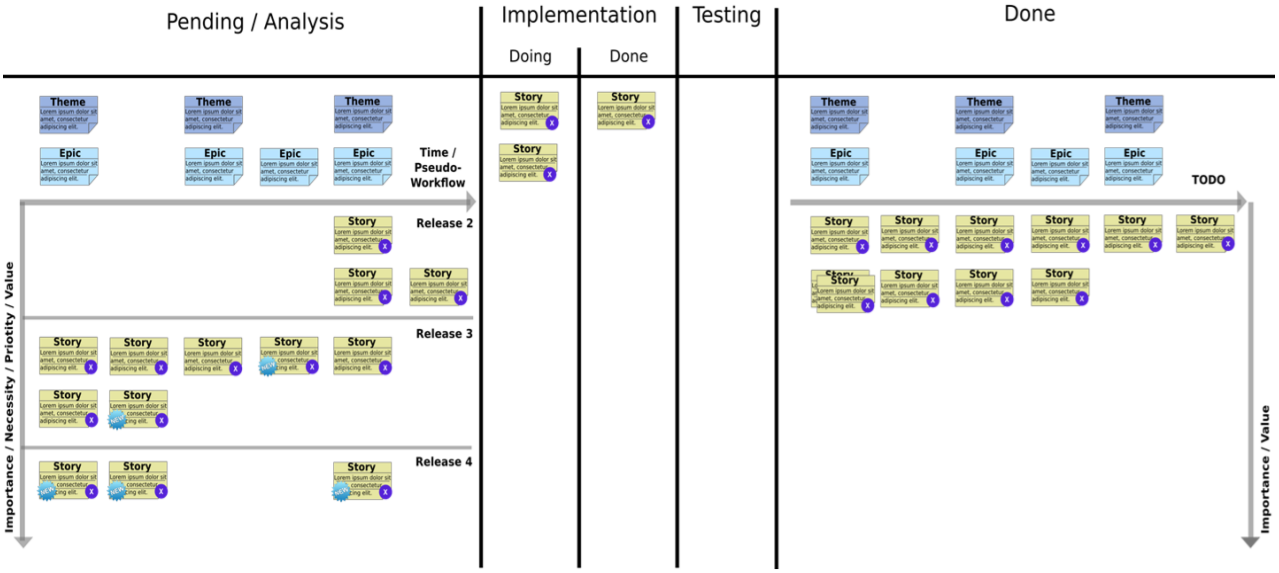
Notes:

- Actual releases **will** differ: we can release potentially at every end of Sprint. Releases identified on the Story Map on the left will likely be broken down in smaller releases.

- Again, one should embrace **Continuous Delivery**: The development Team releases at every end of sprint. Making it a customer release is a Product Management Decision
- One should consider feature flipping (see 4.3.5 Zero Downtime Deployments) in order not to compromise a potential release with a story that would not have been completely implemented in one sprint.

E. No notion of release in *Done* (Right Story Map)

The Story Map on the right shouldn't have any notion of releases. It represents the Product as is the current development version and it makes no sense anymore remembering there which task has been developed in which release.

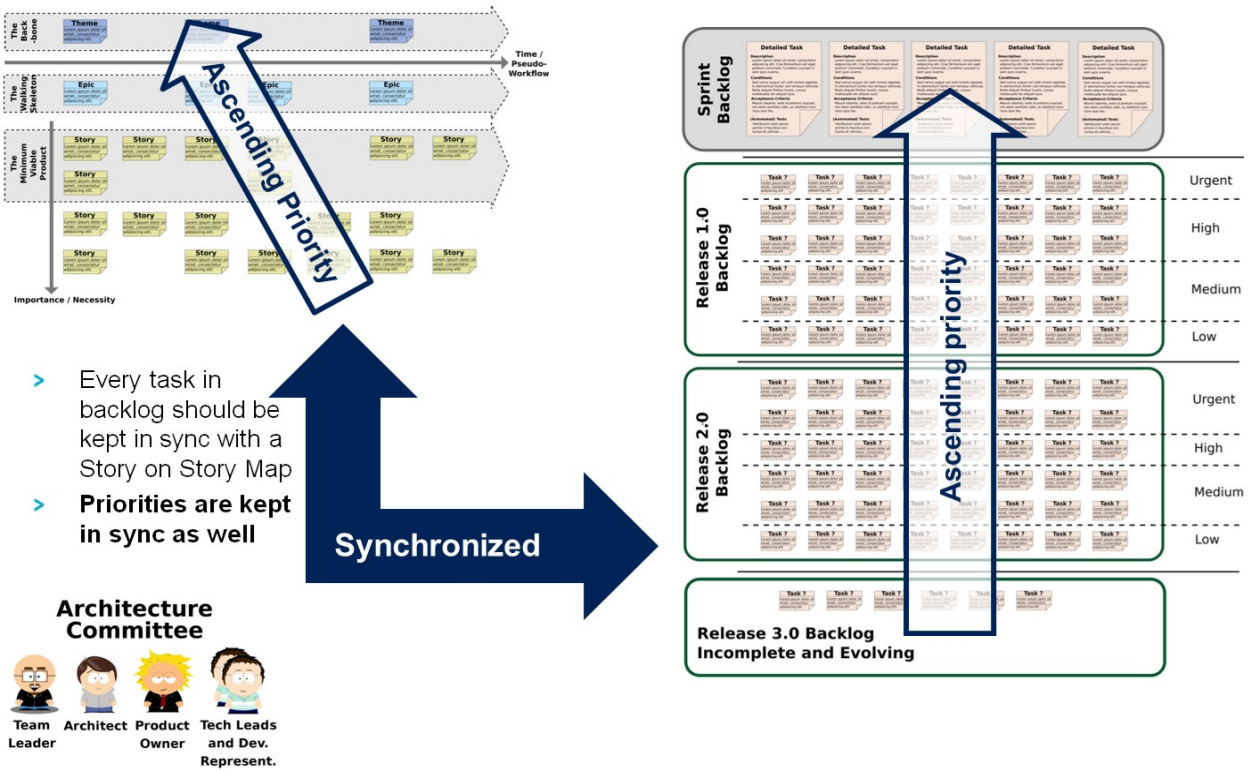


Also, User stories on the right may need to be merged whenever they relate to the same feature.

3.3.3.4 Story Map and Backlog synchronization Process

The priorities of the *Development Tasks* on the *Product Backlog* should match and follow the priorities of the *User Stories* on the *Story Map*.
When a story priority changes, the priorities of the corresponding Development Tasks on the Product Backlog should be changed in order to reflect the new priority of the User Story.

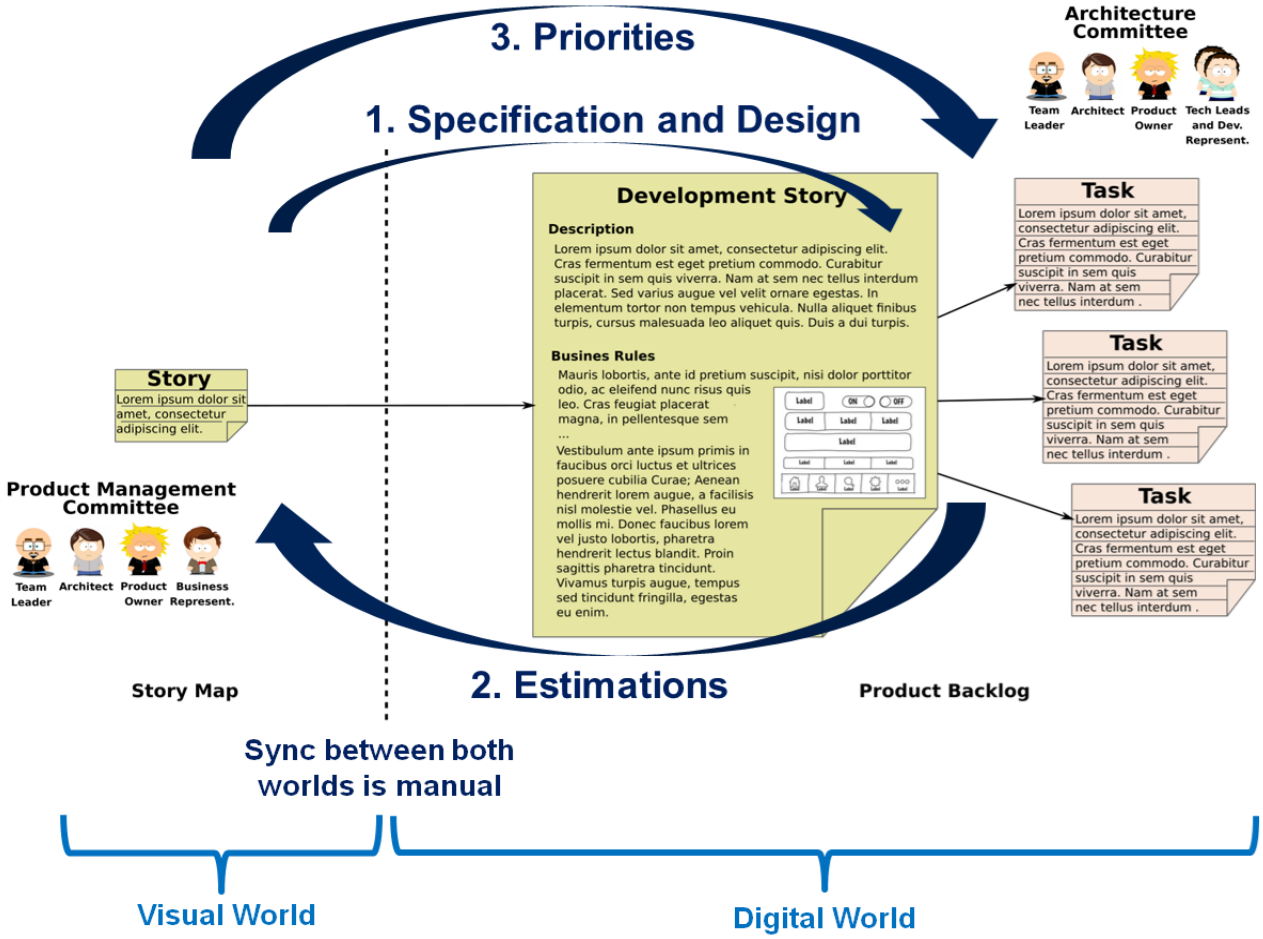
The principle is as follows:



In terms of process, things occur this way:

1. The *Architecture Committee* takes Stories created by the Product Management Committee, designs them and estimates them.
2. The *Product Management Committee* learns about Stories Estimations and re-prioritizes the Story Map accordingly
3. The *Architecture Committee* synchronizes the priorities of the corresponding Development Tasks.

This can be represented this way:



Let's see now how all of this is used to be able to achieve its ultimate objective : reliable planning and forecasting.

3.3.3.5 Forecasting

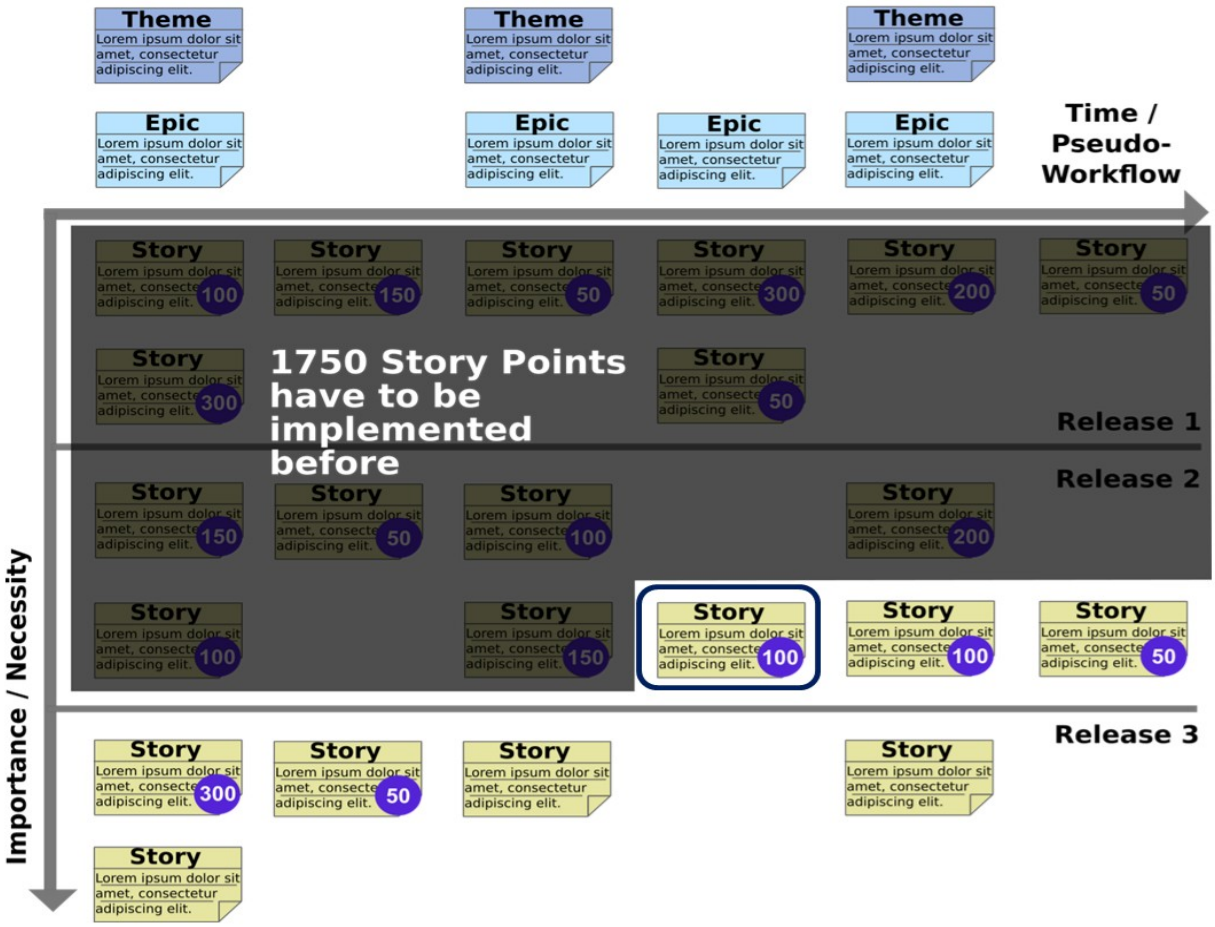
So ... forecasting, finally.
At the end of the day, pretty much everything I have presented above, all these tools, charts and processes are deployed towards this ultimate objective: doing planning and being able to produce accurate forecasts.

If one respects well the processes presented above and use the tools the right ways, one should end up with the Story Map presented in 3.3.3.2 Estimation Process, hence Stories that hold the indication of a pretty accurate estimation in Story Points.

In addition, a story map holds an important notion of priority: the development team needs to implement all the stories of a row, from left to right, before it can consider the stories of the next row.

So how does one know when a story will be implemented by the development team? The answer is simple: when all stories of the previous rows as well as all stories on

the left on the same row are implemented.
From there, calculating the amount of Story Points to be developed before a specific story can be implemented is straightforward:



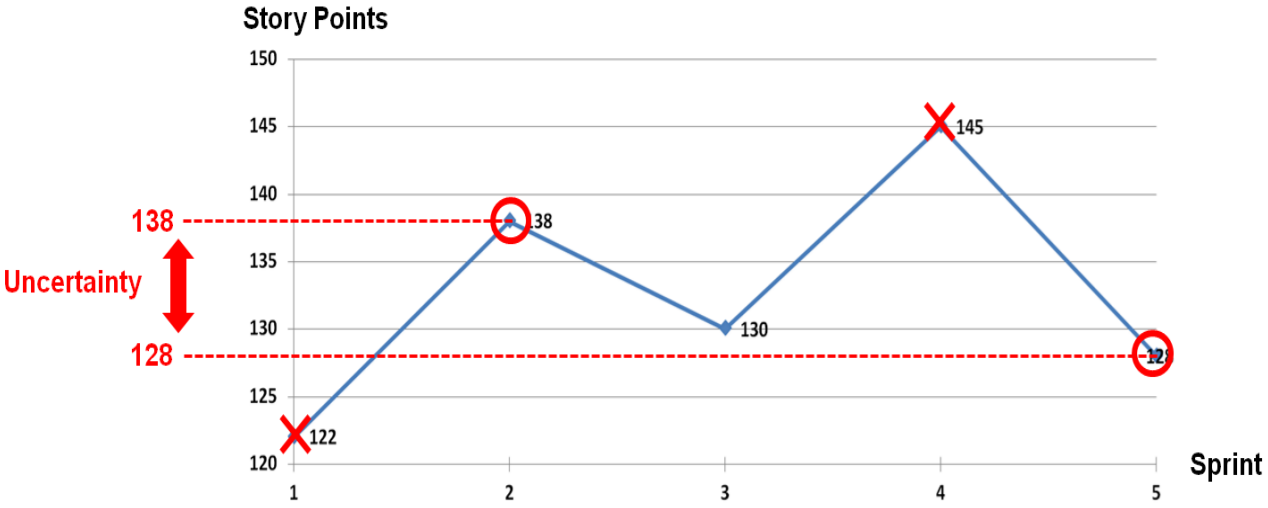
Recovering the example introduced in #3.3.2 Estimation Process, if we want to know when the Story with the blue box around it, we have first to know how many story points have to be implemented first, 1750 SP in this example.

Base on this, we know that this story will be **delivered once all the stories before it will be implemented plus this story as well**, hence 1750 + 100 SP = 1850 SP.

Estimating a delivery date

In order to estimate a delivery date for that story, we need to know how much time is required to deliver these 1850 SP.
Here comes the notion of Sprint capacity, or rather Spring velocity. Strictly speaking, Agilists speak of capacity when reasoning of man days and Sprint velocity when reasoning in Story Points.
I myself use Sprint capacity for both cases.

Computing Sprint velocity requires to have all the practices described in introduction in place for several Sprints. I will come back on practices in the next chapters so I'm leaving them aside for now.
If the Agile Team is mature in regards to its practices, it can compute the Sprint Capacity be looking at the range of Story Points achieved during 5 last sprints:



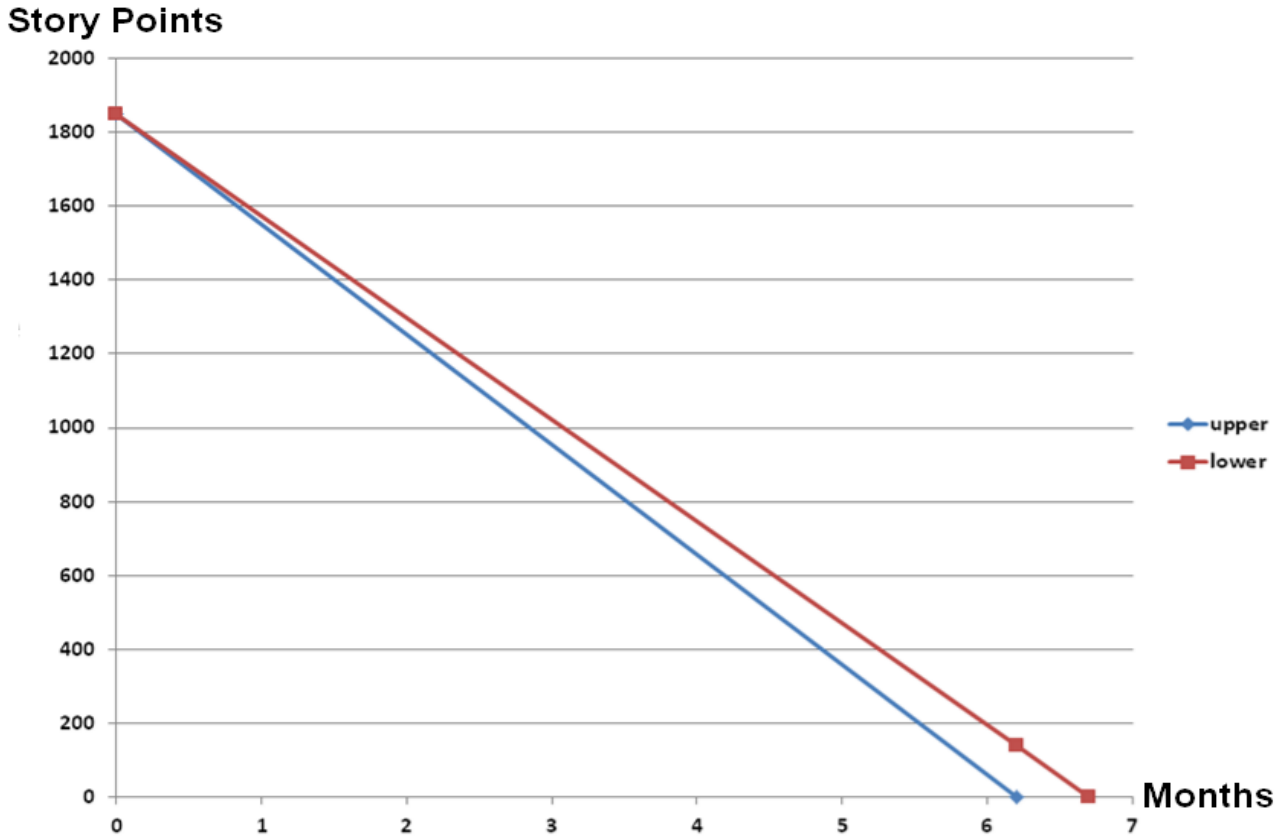
We don't use the most extreme, minimum and maximum values. Extreme values most of the time explain themselves by external factors: people get sick, leaves on holidays, tasks are sometimes finished in next sprint, etc.
Instead, out of five sprints, we'll use the second maximum value and the last-but-one value.

We use this range, and not a single value of average or median, to address a fundamental aspect of software engineering: the uncertainty.
The range gives us a lower value and an upper value which we will use as follows.

- **In case of *fixed time***, when we have a fixed delivery date, the lower and upper values give us the minimum or maximum set of features we can have implemented at that date.
- **In case of *fixed scope***, when we have to release a version of the software with a given set of features, the lower and upper values will give us the earliest date and the latest date at which we can release.

As a sidenote, when we count Story Points implemented in a sprint, we should focus on developer tasks, not User Stories, since User Stories are too coarse grained.
A User story can well take several sprints to be completed. A developer task within one of these stories should not. Tasks should be designed in such a way that they are small enough to always fit a sprint.

Recovering the example above, let's imagine we are want to achieve a fixed scope, we want to know, using the Story Map as it is, how much time will be required to implement these 1850 Story Points.

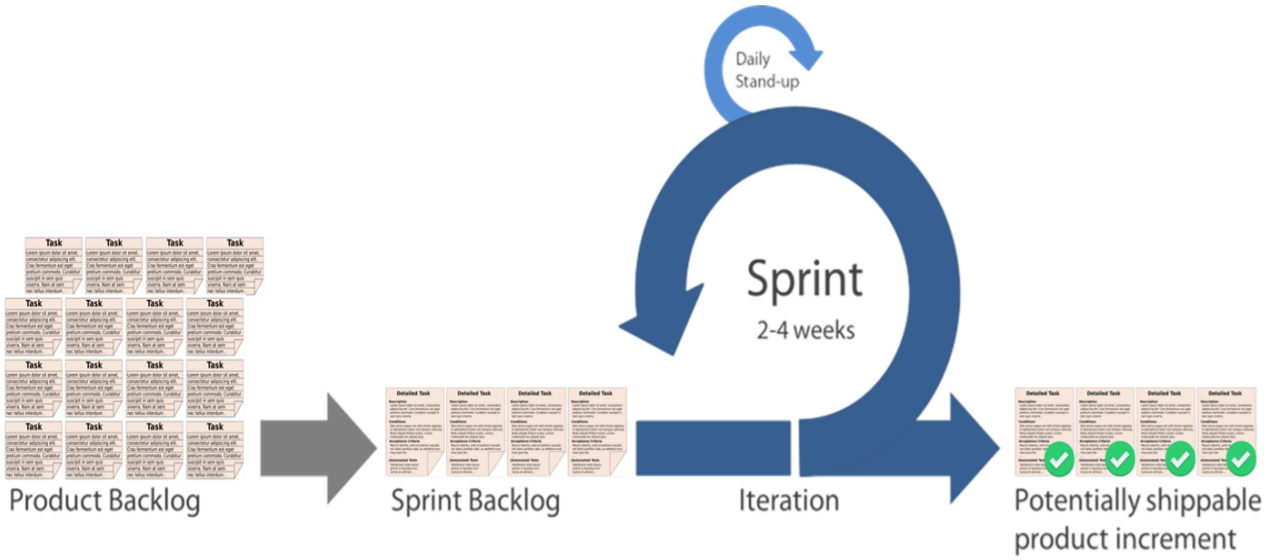


- Using the lower limit of 128 SP per sprint, it would take us 15 sprints to complete the scope, hence 30 weeks or 6.7 months
- Using the upper limit of 138 SP per sprint, it would take us 14 sprints to complete the scope, hence 28 weeks or 6.2 months

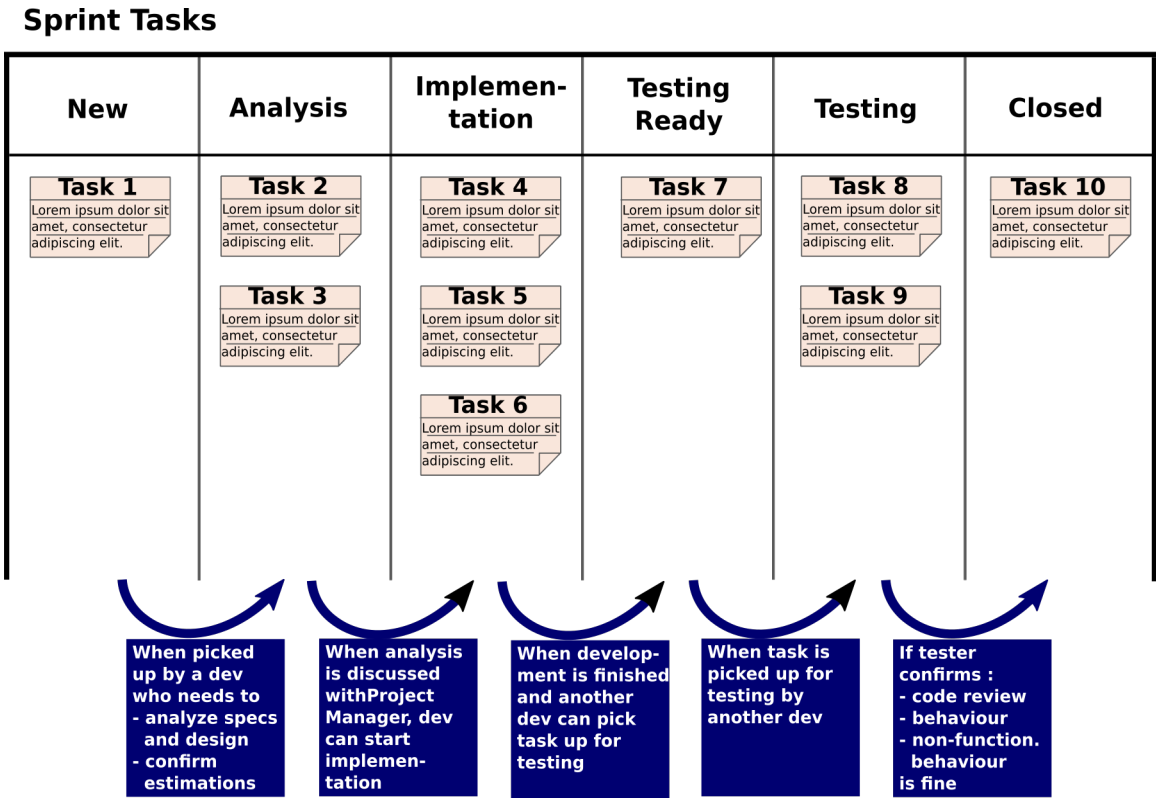
Based on this, the PMC or the Product Owner can communicate to the stakeholders that the feature would be release not before 6 months but before 7 months.

3.3.3.6 Development process: Scrum

I said enough about Scrum in this paper already.
Let me just introduce this chart that does a great job in introducing the notion of **Product Increment** as a shippable version of the product since we adopt Continuous Delivery:



This allows me to present the last tool I mentioned in the introduction of this sprint, which is the **Sprint Kanban Board**:



The sprint Kanban board is used to track the progress of tasks within the sprint and enables to organize developer activities.

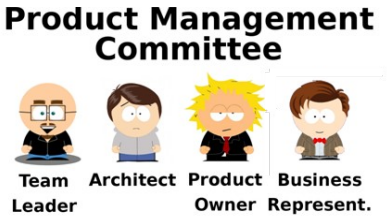
Some people use extensively *burndown charts* to track the proper progress of a sprint or the product backlog towards a specific release as a whole. I myself never

find it so useful. I really get all I want to know about how a release or a specific sprint is doing by using the Product Backlog, the Product Kanban Board or the Sprint Kanban.

3.3.4 The Rituals

Rituals of the various teams are as follows.
Committees are rituals by themselves, the difference between a team and a committee is that a committee gathers solely for a specific ritual

3.3.4.1 Product Management Committee



The *Product Management Committee* gather every X weeks. It really depends of the corporation, the size of the team, the rate at which new functional requirements appear. Every few 2 weeks should be sufficient in general, otherwise the frequency can increase as far as every week.

The duties of the *Product Management Committee* are as follows:

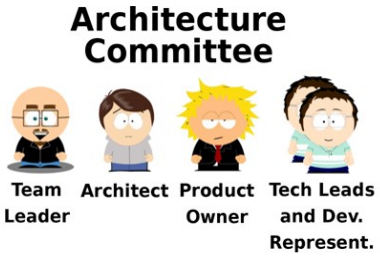
Story Mapping

- Identification of new needs and requirements (also technical and technological!)
- Breakdown of these requirements in User Stories
- "*Guessing*" of an Initial Priority of a User Story based on Value (and foreseen size)

Maintenance (update) of Priorities

- Setting of Actual Priorities based on Estimations from Architecture Committee
- Review of priorities of Whole Story Map after update of estimations
    - From Sprint Management Committee
    - From Development Team

3.3.4.2 Architecture Committee



The *Architecture Committee* also gather every X weeks. It should meet at least few minutes (coffee break) but not more than one or two days after *Product Management Committee*.
The Architecture Committee recovers the last User Stories designed at PMC and synchronizes the Product Backlog with the Story Map. Stories are specified, designed and broken downs in Development Tasks.

The duties of the *Architecture Committee* are as follows:

Specification and Design of User Stories

- Specification of functional and non-functional requirements
- Identification of business rules
- Identification of Acceptance criteria
- Design of GUI
- Architecture and Design of Software

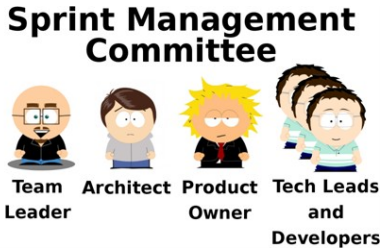
Estimation of User Stories

- Breakdown in individual Development Tasks
    - This needs to be done sufficiently in advance
- Estimation of Development Tasks
- Computing of total Estimation and reporting on User Story
- Continuous Improvement: understanding of gaps in estimation after notification of Sprint Committee and how to improve

Software Architecture

- Identification and maintenance of Coding Standards and Architecture Standards
- Review of ad'hoc architecture topics

3.3.4.3 Sprint Management Committee



The *Sprint Management Committee* gathers at every beginning and end of sprint. A sprint starts with the *Sprint Planning* and ends with *Sprint Demo* and *Sprint Retrospective*:

Sprint Planning

- Discuss Development Tasks to ensure whole team has a clear view of what needs to be done → Detailed Tasks
- Review and challenge estimations of Detailed Tasks. Update estimation of User Story accordingly
- Feed the Sprint Backlog with such Detailed Tasks until Sprint Capacity is reached

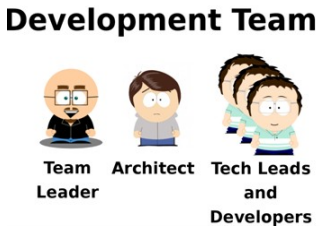
Sprint Retro

- Review Tasks not completed and create task identifying GAP for next Sprint. Update estimations.
- Review SP achieved during sprint and review Sprint Capacity
- Discuss issues encountered during Sprint and identify action points. Update processes and rituals accordingly
- Continuous Improvement: understanding of gaps in tasks and estimations and how to improve

Sprint Demo

- End of Sprint / really optional with Continuous Delivery and Continuous Acceptance Tests
- Present sprint developments and integrate feedback. Create new tasks and update estimations.

3.3.4.4 Development Team - Daily Scrum



The daily scrum happens every day, ideally early in the moment, at the time all the team is in the office.
The scope of the daily scrum is as follows:

Round table - every team member presents:

- Past or current development task
- Status on that task and precise progress
- Next steps
- Next task if former is completed
- Identification of unforeseen GAPS and adaptation of estimations

Identification of challenges, issues and support needs

- Scheduling of ad'hoc meeting and required attendees to discuss specific issues

3.3.5 The Values

Sticking rituals, respecting principles and enforcing practices is difficult.

- It's difficult to ensure and behaves in such a way that breaking the build (failing tests) is an exception.
- It's difficult to respect the boyscout rule.
- It's a lot more difficult to design things carefully and stick to the KISS principle.
- It's difficult and a lot of work to keep the Story Map and Product Backlog in sync and up-to date with the reality.
- It's difficult to stick to the TDD approach.
- It's difficult not to squeeze the Kaizen phase at the end of every meeting and being objective when it comes to analyzing strengths and weaknesses.

All of this make two Agile values especially important: **Discipline** and **courage**.
Both are utmost important and essential to address these difficulties.

Sticking to the Scrum rituals, enforcing TDD and other XP principles and practices require courage and discipline. It also requires a lot of discipline to Maintain and synchronize the Product Backlog and the Story Map.
Updating the estimations of the User Stories continuously as the understanding of the work to be done progresses also takes a lot of discipline.

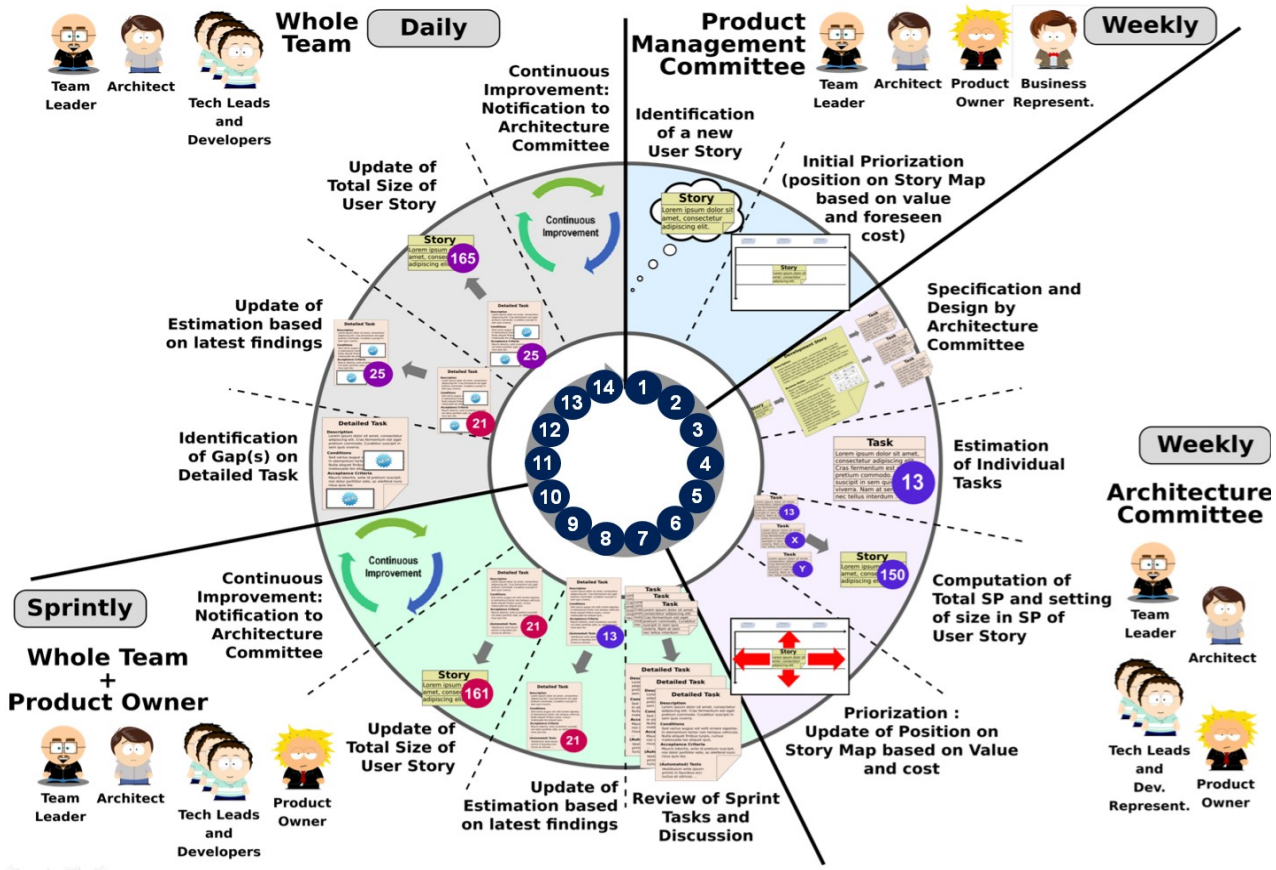
Finally, discipline and courage are enforced by a strict definition of the processes and rituals and a proper maintenance of this definition as the culture and practices evolve.
At the end of the day, defining these committees and rituals is all about that. Why are all these committees / teams / rituals required if eventually a person can have several roles? Because they enforce discipline: they are scheduled and have precise agendas.

3.4 Overview of the whole process

The whole process looks as follows:

- Product Management Committee (X-Weekly)
    - **1** Identification of a new User Story
    - **2** Initial foreseen priority (i.e. release) depending on value and initial estimation (oral)
- Architecture Committee (X-Weekly)
    - **3** Design and specification by architecture committee : Story → Development Story → Task
    - **4** Estimation of individual tasks
    - **5** Computation of total SP and setting of size of Development Story and User Story
    - **6** Re-prioritization (based on new estimation)
- Sprint Planning + Sprint retrospective (Sprintly)
    - **7** Review of TaskS and discussion : Task → Detailed Task
    - **8** Adaptation of Estimation on TaskS
    - **9** Update of Total Size of Development Story and User Story
    - **10** Notification to Architecture Committee (Kaizen / Sprint retrospective)
- Daily Scrum
    - **11** Identification of Gap on Task
    - **12** Adaptation of Estimation on Task
    - **13** Update of Total Size of Development Story and User Story
    - **14** Notification to Architecture Committee (Kaizen / Sprint retrospective)

In a graphical way:

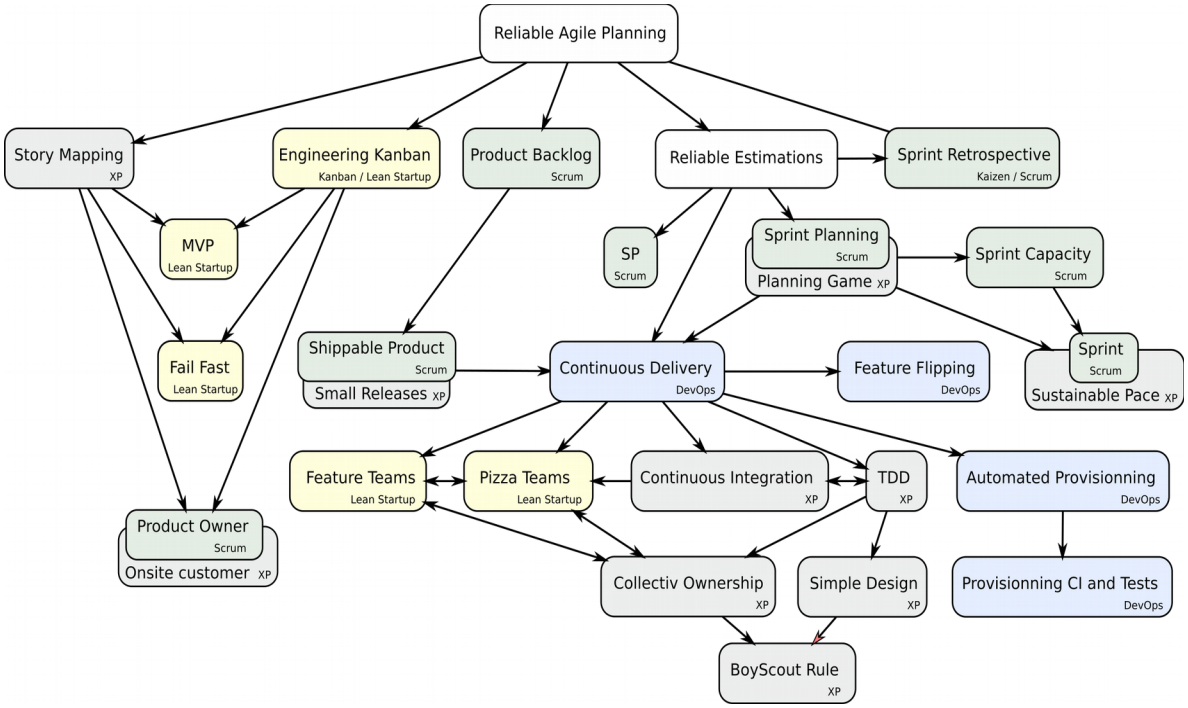


3.5 Return on Practices

As stated a lot of times in this paper, all of this, reliable planning and true agility, require a strong commitment of the team to Agile practices and principles.

One cannot apply only a small subset of the Agile Practices and believe he will achieve true agility and Reliable Agile Planning.
The Agile practices I listed in introduction form a package with strong dependencies between each other.

IMHO the dependencies are as follows:



An arrow denotes a dependency between two practices.

Explanations of a few of these dependencies:

- You cannot imagine **reliable planning and forecasting** if you don't provide the management with appropriate tools : **Story Map** and **Kanban boards**. Also, it's going to be difficult without a proper technical tool for the development team: **The Product Backlog**.
  Finally, it obviously requires **Reliable Estimations**.
- **Reliable estimations** need to have manageable and well **planned sprints**. 1 week sprints are too small, a lot can happen in 1 week while 3 weeks are too big in my opinion, the fluctuations are too important. I strongly believe that 2 weeks sprints is the right size when it comes to having an accurate and reliable Sprint Capacity (or Velocity) in **SP**.
  With 2 weeks sprints only, the development team cannot afford spending time on releasing the **Shippable Product**, releasing should be a completely automated procedure and in this regards **Continuous Delivery** is not optional.
- Then achieving **Continuous Delivery** requires a lot of things and a good mastery of common XP and DevOps Practices.

3.6. Conclusion

Management needs a management tool to take enlightened decision. The product backlog should not be a management tool, it's really rather the development team's

internal business. The Story Map, on the other hand, is a simple, visual and effective management tool.
All the rituals and processes introduced in this paper are deployed towards the same ultimate goal: **enabling the management to use the Story Map as a management tool for planning and forecasting.** In addition, the specific form of Story Map introduced here, the **Product Kanban Board**, becomes also a Project Management Tool aimed to tracking the progresses of the development team.

The difficulty, the reason why it requires a strict enforcement of processes and rituals, is to synchronize the Story Map and the Product Backlog.
Since the development team works mostly with the Product Backlog, the later has eventually the accurate and realistic information about size and time of deployment, through the notion of Story Points.
But this is in no help for the management, hence the reason why it is required to backfeed the estimations put in the Product Backlog to the Story Map.

Eventually, if these processes and rituals are respected and well applied, anyone in the company can come in front of the *Product Kanban Board* with a little calculator and compute the delivery date (or rather the range) for any given story.
Anyone can use the Story Map to compute how much work can be done for any given date, or what time is required to deliver a specific scope.

All of this with a simple calculator and a few seconds, without Excel, without any Internet connection, without any complicated too nor any pile of paper, just a calculator ... or a brilliant mind.

Now having said that, I would like to conclude this paper by mentioning that the processes and tools I am presenting here work for us. They may not work as is for another organization. It's up to every organization to discover and find the practices and principles that best fit its needs and individuals.
As an example, the association of two Story Maps, the "*to do*" on the left and the "*done*" on the right of a Kanban Board for the needs of both Product and Project Management is a really personal recipe. While I myself got the idea from another organization, I haven't seen that often.
This shows in my opinion the very best qualities of an agilist: the curiosity to discover new ways of working and the courage to try them or invent them.

4. DevOps explained

So ... I've read a lot of things recently on DevOps, a lot of very interesting things ... and, unfortunately, some pretty stupid as well. It seems a lot of people are increasingly considering that DevOps is resumed to mastering chef, puppet or docker containers. This really bothers me. DevOps is so much more than any tool such as puppet or docker.

This could even make me angry. DevOps seems to me so important. I've spent 15 years working in the engineering business for very big institutions, mostly big financial institutions. DevOps is a very key methodology bringing principles and practices that address precisely the biggest problem, the saddest factor of failure of software development projects in such institutions : the *wall of confusion* between developers and operators.

Don't get me wrong, in most of these big institutions being still far from a large and sound adoption of an Agile Development Methodology beyond some XP practices, there are many other reasons explaining the failure or slippage of software development projects.
But the *wall of confusion* is by far, in my opinion, the most frustrating, time consuming, and, well, quite stupid, problem they are facing.

So yeah... Instead of getting angry I figured I'd rather present here in a concrete and as precise as possible article what DevOps is and what it brings. Long story short, DevOps is not a set of tools. **DevOps is a methodology** proposing a set of **principles and practices**, period. The tools, or rather the toolchain - since the collection of tools supporting these practices can be quite extended - are only intended to support the practices.
In the end, these tools don't matter. The DevOps toolchains are today very different than they were two years ago and will be very different in two years. Again, this doesn't matter. What matters is a sound understanding of the principles and practices.

Presenting a specific toolchain is not the scope of this article, I won't mention any. There are many articles out there focusing on DevOps toolchains. I want here to take a leap backwards and present the principles and practices, their fundamental purpose since, in the end, this is what seems most important to me.

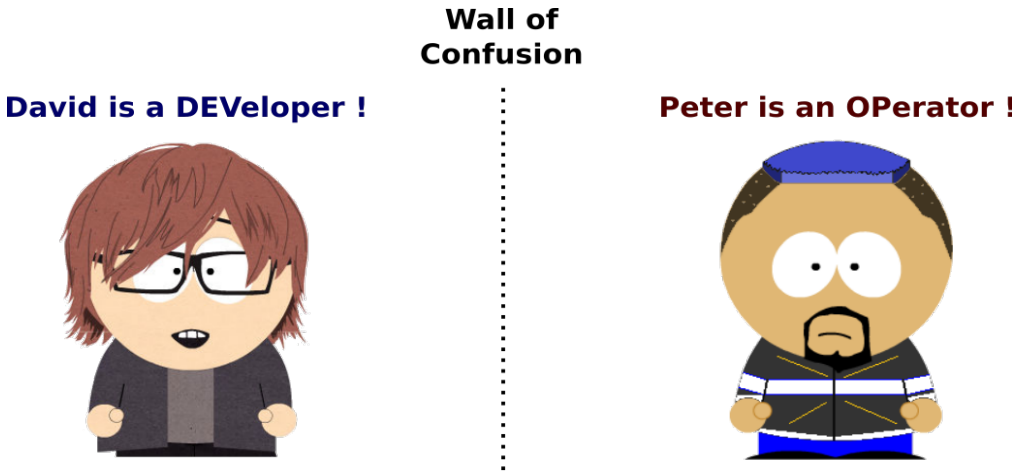
DevOps is a methodology capturing the practices adopted from the very start by the web giants who had a unique opportunity as well as a strong requirement to invent new ways of working due to the very nature of their business: the need to evolve their systems at an unprecedented pace as well as extend them and their business sometimes on a daily basis.

While DevOps makes obviously a critical sense for startups, I believe that the big corporations with large and old-fashioned IT departments are actually the ones that can benefit the most from adopting these principles and practices. I will try to explain why and how in this article.

4.1 Introduction

DevOps is not a question of tools, or mastering chef or docker. DevOps is a methodology, a set of principles and practices that help both developers and operators reach their goals while maximizing value delivery to the customers or the users as well as the quality of these deliverables.

The problem comes from the fact that developers and operators - while both required by corporations with large IT departments - have very different objectives.



This difference of objectives between developers and operators is called the **wall of confusion**. We'll see later precisely what that means any why I consider this something big and bad.

DevOps is a methodology presenting a set of principles and practices (tools are derived from these practices) aimed at having both these personas working towards an unified and common objective : **deliver as much value as possible for the company**.

And surprisingly, for once, there is a magic silver bullet for this. Very simply, the secret is to **bring agility to the production side**!
And that, precisely that and only that, is what DevOps is about !

But there are quite a few things I need to present before we can discuss this any further.

4.1.1 The management credo

What is the sinews of war of IT Management ? In other words, when it comes to Software Development Projects, what does management want first and foremost ?

Any idea ?

Let me put you on tracks : what is utmost important when developing a startup ?

Improve Time To Market (TTM) of course !

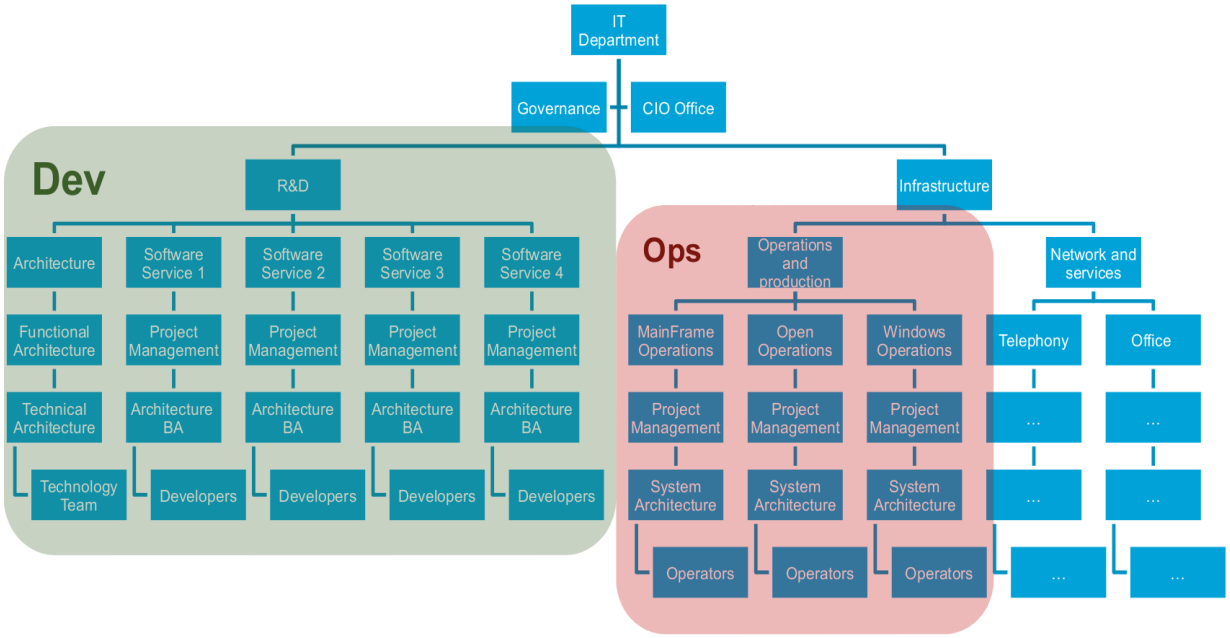
The **Time To Market** or TTM is the length of time it takes from a product being conceived until its being available to users or for sale to customers. TTM is important in industries where products are outmoded quickly.
In software engineering, where approaches, business and technologies change almost yearly, the TTM is a very important KPI (Key Performance Indicator).
The TTM is also very often called **Lead Time**

A first problem lays in the fact (as believed by many) that TTM and product quality are opposing attributes of a development process. As we will see below, improving quality (and hence stability) is the objective of operators while reducing lead time (and hence improving TTM) is the objective of developers.
Let me explain this.

An IT organization or department is often judged on these two key KPIs : the quality of the software, where the target is to have as little defects as possible, and the TTM, where the target is to be able to go from business ideas (often given by business users) to production - making the feature available to users or customers - as soon as possible.
The problem here is that most often these two distinct objectives are supported by two different teams : the *developers*, building the software, and the *operators*, running the software.

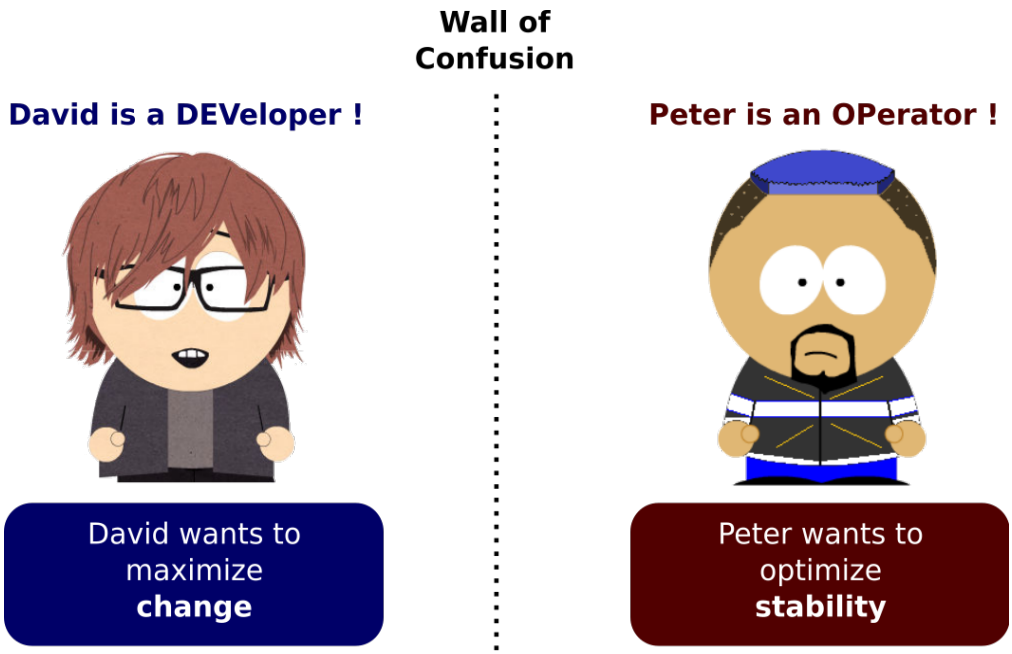
4.1.2 a typical IT organization

A typical IT organization, in a corporation owning an important IT department, looks as follows :



Mostly for historical reasons (operators come from the hardware and telco business most often), operators are not attached to the same branch than developers. Developers belong to R&D while operators most of the time belong to Infrastructure department (or dedicated operation department).

Again, they have different objectives:

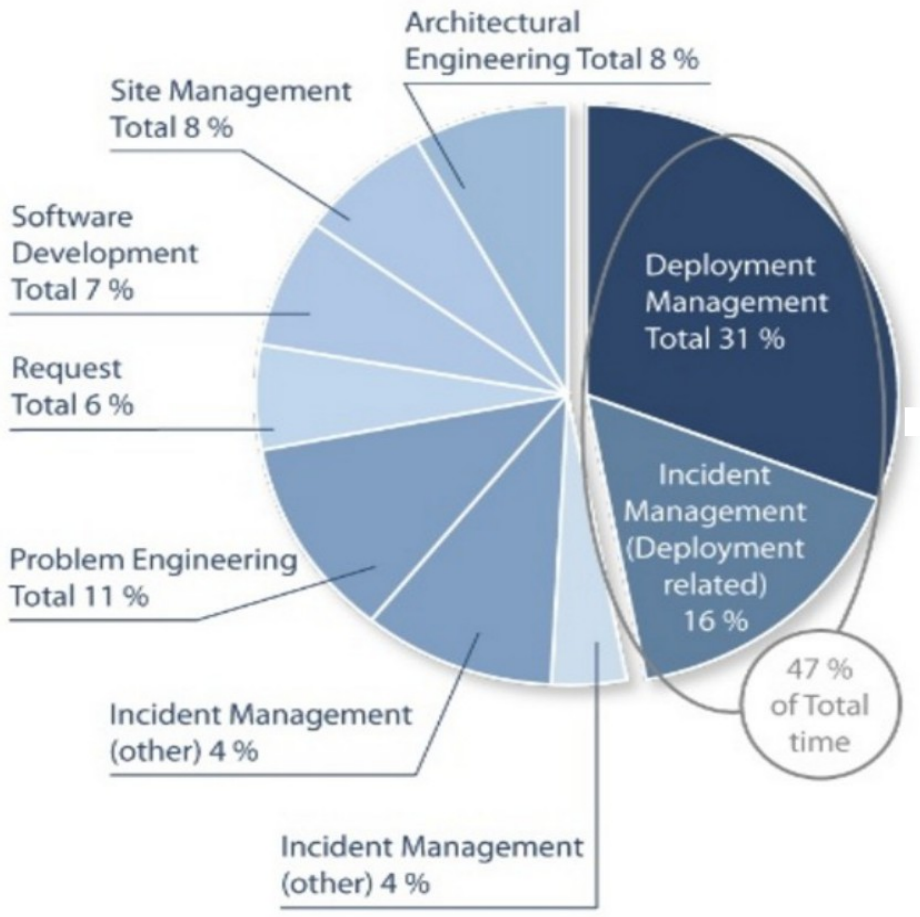


In addition, and as a sidenote, these both teams sometimes even run on different budget. The development team uses the *build* budget while the operation team uses

the *run* budget. These different budgets and the increasing needs to control and shorten the costs of IT in corporation tend to emphasize the opposition of objectives of the different teams.
(In parenthesis: nowadays, with the always and everywhere interconnection of people and objects pushing the digitalization of businesses and society in general, the old Plan / Build / Run framework for IT budgeting makes IMHO really no sense anymore, but that is another story)

4.1.3 Ops frustration

Now let's focus on operators a little and see, in average, how a typical *operation team* spends its time:



(Source : Study from Deepak Patil [Microsoft Global Foundation Services] in 2006, via James Hamilton [Amazon Web Services]
http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_POA20090226.pdf)

So almost 50% (47) of total time of Production Teams is dedicated to deployment related topics:

- Actually doing deployment or

- Fixing problems related to deployments

This is actually a pretty crazy KPI, one that should have been followed much sooner. The truth is, operator teams have been since their inception in the early age of Computer Engineering - 40 years ago, at the time computers were massively introduced in the industry - this kind of hackers running tons of commands manually to perform their tasks. They are used to long checklists of commands or manual processes to perform their duties.
Somehow, they suffer from the "*We always did it like this*" syndrome and challenged very little their ways of working over these 40 years.
But if you think of it, this is really crazy. In average, operators spend almost 50% of their time doing deployment related tasks!

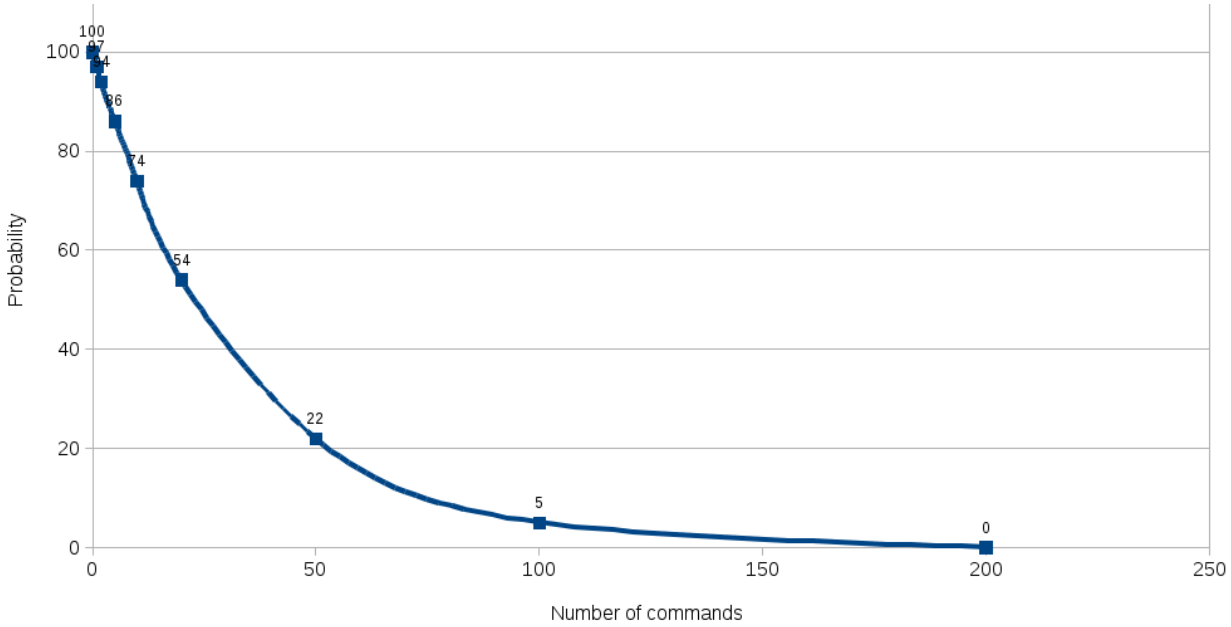
This underlines two critical needs for evoluting these processes:

1. Automate the deployments to reduce the 31% time dedicated to these currently manual tasks.
2. Industrialize them (just as software development has been industrialized, thanks to XP and Agile) to reduce the 16% related to fixing these deployment related issues.

4.1.4 Infrastructure automation

In this regards, another statistic is pretty enlightening:

Probability of succeeding an installation expressed as a function of the number of manual operation



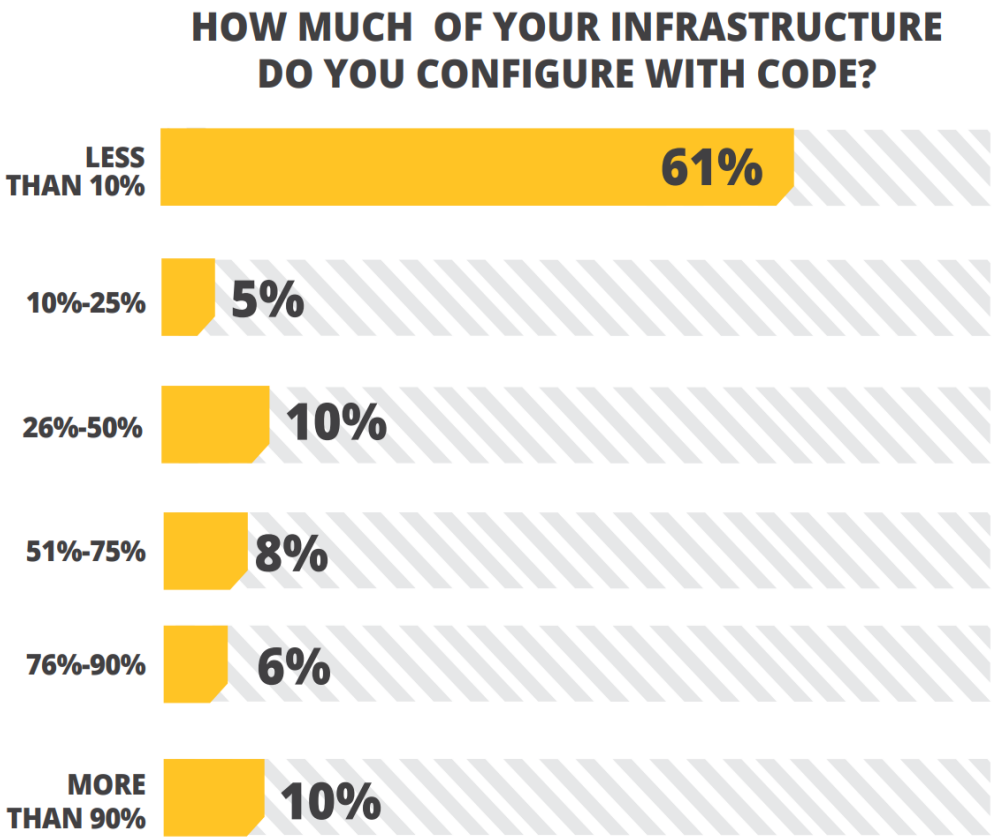
This is read the following way :

- With only 5 manual commands, the probability of succeeding an installation drops to 86% already.
- With 55 manual commands, the probability of succeeding an installation drops to 22%.
- With 100 manual commands, the probability of succeeding an installation is close to 0! (2%)%

Succeeding the installation means that the software behaves in production as intended. Failing it means something will go wrong and some analysis will be required to understand what went wrong with the installation and some patches will need to be applied or some configuration corrected.

So automating all of this and avoiding manual commands at all cost seems to be rather a good idea, doesn't it ?

So what's the status in this regards in the industry:



(Source : IT Ops & DevOps Productivity Report 2013 - Rebellabs - http://pages.zeroturnaround.com/rs/zeroturnaround/images/it-ops-devops-productivity-report-2013%20copy.pdf)

(To be perfectly honest, this statistic is pretty old - 2013 - I would expect a little different numbers nowadays)

Nonetheless, this gives a pretty good idea of how much is still to be accomplished in regards to Infrastructure automation and how much DevOps principles and practices are very important.

Again the web giants had to come up with a new approach, with new practices to address their needs of responsiveness. What they started their engineering business in their early days, the practices they put in place is at the root of what is today DevOps.

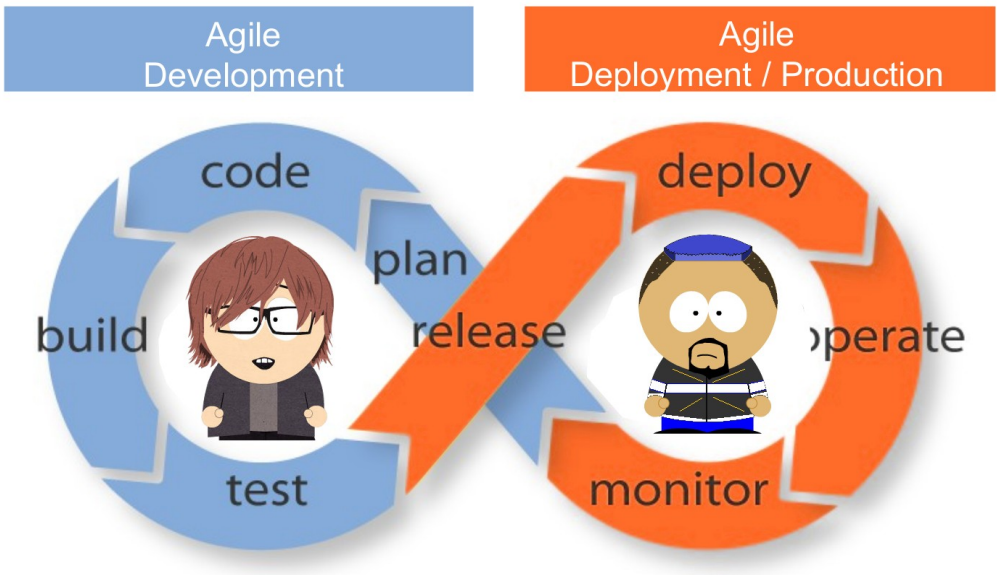
Let's look at where the web giants stand now in this regards. A few examples:

- Facebook has thousands of devs and ops, hundreds of thousands of servers. In average, an operator takes care of 500 servers (think automation is optional ?). They do two deployments a day (concept of deployment ring)
- Flickr does 10 deployments a day
- Netflix designs for failure! The software is designed from the grounds up to tolerate system failures. They test it all the time in production: 65'000 failure tests in production daily by killing random virtual machines ... and measuring that everything still behaves OK.

So what is their secret ?

4.1.5 DevOps : For once, a magic silver bullet

The secret is simply to **Extend Agility to Production**:



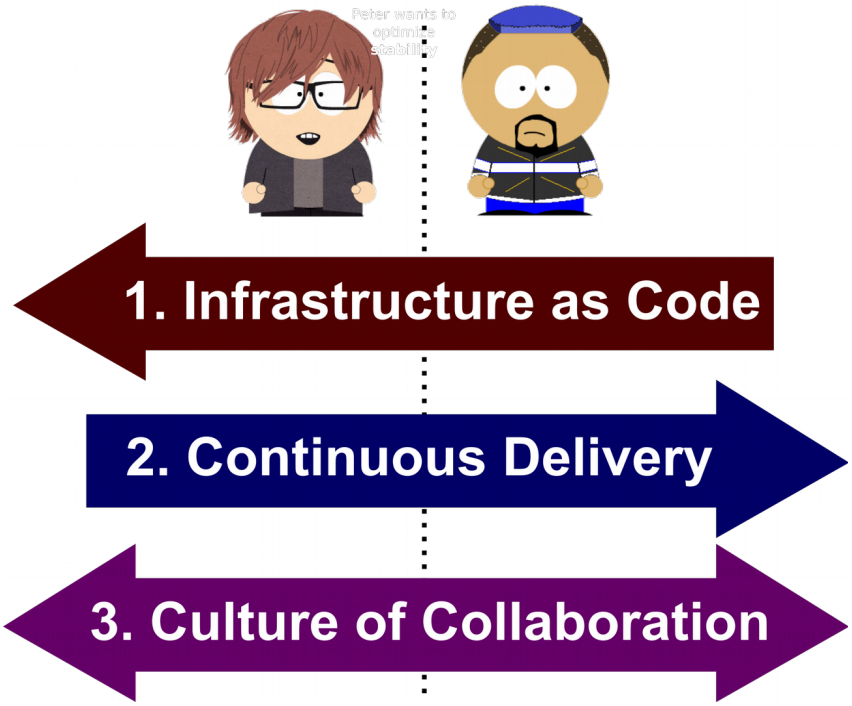
DevOps consists mostly in extending agile development practices by further streamlining the movement of software change thru the build, validate, deploy and delivery stages, while empowering cross-functional teams with full ownership of software applications - from design thru production support.

DevOps encourages **communication**, **collaboration**, **integration** and **automation** among software developers and IT operators in order to improve both the speed and quality of delivering software.

DevOps teams focus on standardizing development environments and automating delivery processes to improve delivery predictability, efficiency, security and maintainability. The DevOps ideals provide developers more control of the production environment and a better understanding of the production infrastructure.

DevOps encourages empowering teams with the autonomy to build, validate, deliver and support their own applications.

So what are the core principles ?



We'll now dig into these 3 essential principles.

4.2 Infrastructure as Code

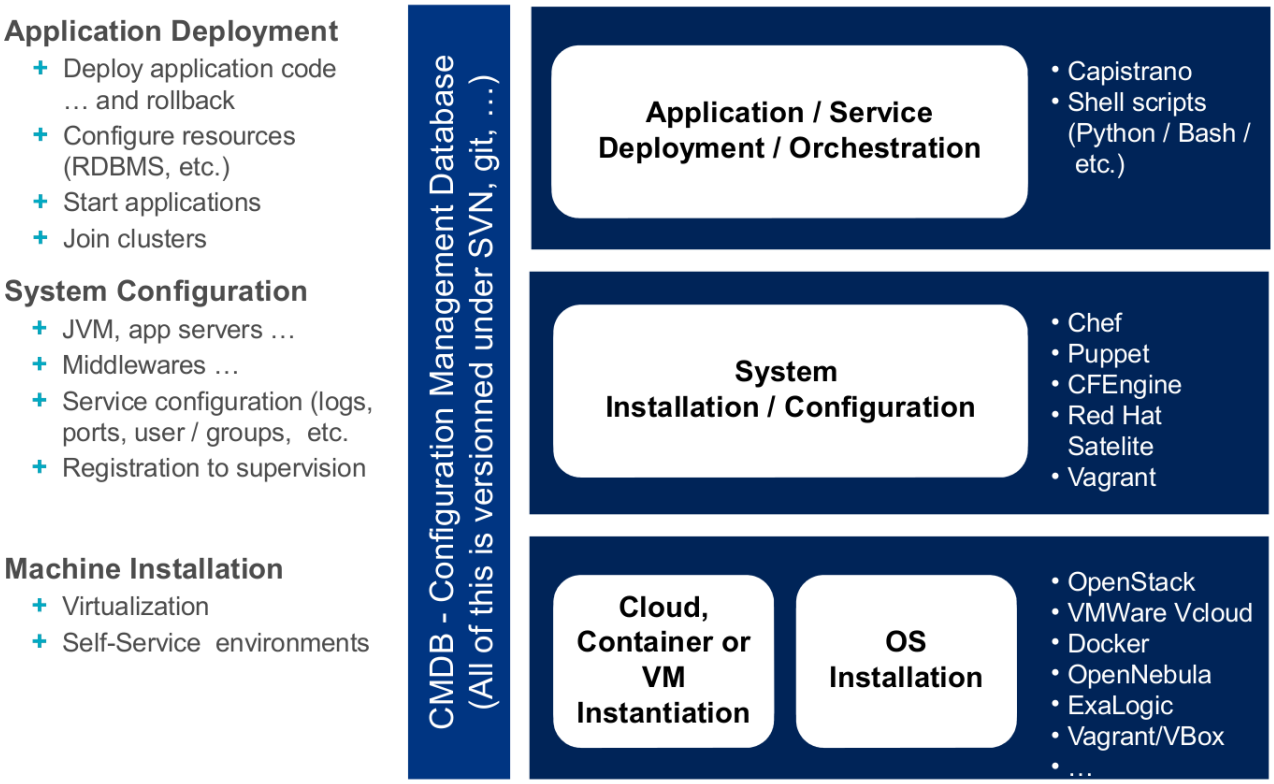
Because humans make mistakes, because the human brain is terribly bad at repetitive tasks, because humans are slow compared to a shell script, and because we are humans after all, we should consider and handle infrastructure concerns just as we handle coding concerns!

Infrastructure as code (IaC) is the prerequisite for common DevOps practices such as version control, code review, continuous integration and automated testing. It consists in **managing** and **provisioning** computing infrastructure (containers, virtual machines, physical machines, software installation, etc.) and their

configuration **through machine-processable definition** files or scripts, rather than the use of interactive configuration tools and manual commands.

I cannot stress enough how much this is a key principle of DevOps. It is really applying software development practices to servers and infrastructure.
Cloud computing enables complex IT deployments modeled after traditional physical topologies. We can automate the build of complex virtual networks, storage and servers with relative ease. Every aspect of server environments, from the infrastructure down to the operating system settings, can be codified and stored in a version control repository.

4.2.1 Overview



In a very summarized way, the levels of infrastructure and operation concerns at which automation should occur is represented on this schema. The tools proposed as examples on the schema above are very much oriented towards *building* the different layers. But a devops toolchain does much more than that.
I think it's time I tell a little more about the notion of DevOps Toolchains.

4.2.2 DevOps Toolchains

Because DevOps is a cultural shift and collaboration between development, operations and testing, there is no single DevOps tool, rather, again, a set of them, or *DevOps toolchain* consisting of multiple tools. Such tools fit into one or more of

these categories, which is reflective of the software development and delivery process:

- **Code** : Code development and review, version control tools, code merging
- **Build** : Continuous integration tools, build status
- **Test** : Test and results determine performance
- **Package** : Artifact repository, application pre-deployment staging
- **Release** : Change management, release approvals, release automation
- **Configure** : Infrastructure configuration and management, Infrastructure as Code tools
- **Monitor** : Applications performance monitoring, end user experience

Though there are many tools available, certain categories of them are essential in the DevOps toolchain setup for use in an organization.

Tools such as Docker (containerization), Jenkins (continuous Integration), Puppet (Infrastructure building) and Vagrant (virtualization platform) among many others are often used and frequently referenced in DevOps tooling discussions as of 2016.

Versioning, Continuous Integration and Automated testing of infrastructure components

The ability to **version** the infrastructure - or rather the infrastructure building scripts or configuration files - as well as the ability to **automated test** it are very important.
DevOps consists in finally adopting the same practices XP brought 30 years ago to software engineering to the production side.
Even further, Infrastructure elements should be **continuously integrated** just as software deliverables.

4.2.3 Benefits

There are so many benefits to DevOps. A non-exhaustive list could be as follows:

- **Repeatability and Reliability** : building the production machine is now simply running that script or that puppet command. With proper usage of docker containers or vagrant virtual machines, a production machine with the Operating System layer and, of course, all the software properly installed and configured can be set up by typing one single command - **One Single Command**. And of course this building script or mechanism is continuously integrated upon changes or when being developed, continuously and automatically tested, etc.

Finally we can benefit on the operation side from the same practices we use with success on the software development side, thanks to XP or Agile.

- **Productivity** : one click deployment, one click provisioning, one click new environment creation, etc. Again, the whole production environment is set-up using one single command or one click. Now of course that command can well run for hours, but during that time the operator can focus on more interesting things, instead of waiting for a single individual command to complete before typing the next one, and that sometimes for several days...
- **Time to recovery !** : one click recovery of the production environment, period.
- **Guarantee that infrastructure is homogeneous** : completely eliminating the possibility for an operator to build an environment or install a software slightly differently every time is the only way to guarantee that the infrastructure is perfectly homogeneous and reproducible. Even further, with version control of scripts or puppet configuration files, one can rebuild the production environment precisely as it was last week, last month, or for that particular release of the software.
- **Make sure standards are respected** : infrastructure standards are not even required anymore. The standard is the code.
- **Allow developer to do lots of tasks themselves** : if developers become themselves suddenly able to re-create the production environment on their own infrastructure by one single click, they become able to do a lot of production related tasks by themselves as well, such as understanding production failures, providing proper configuration, implementing deployment scripts, etc.

These are the few benefits of IaC that I can think of by myself. I bet there are so many much more (suggestions in comments are welcome).

4.3 Continuous Delivery

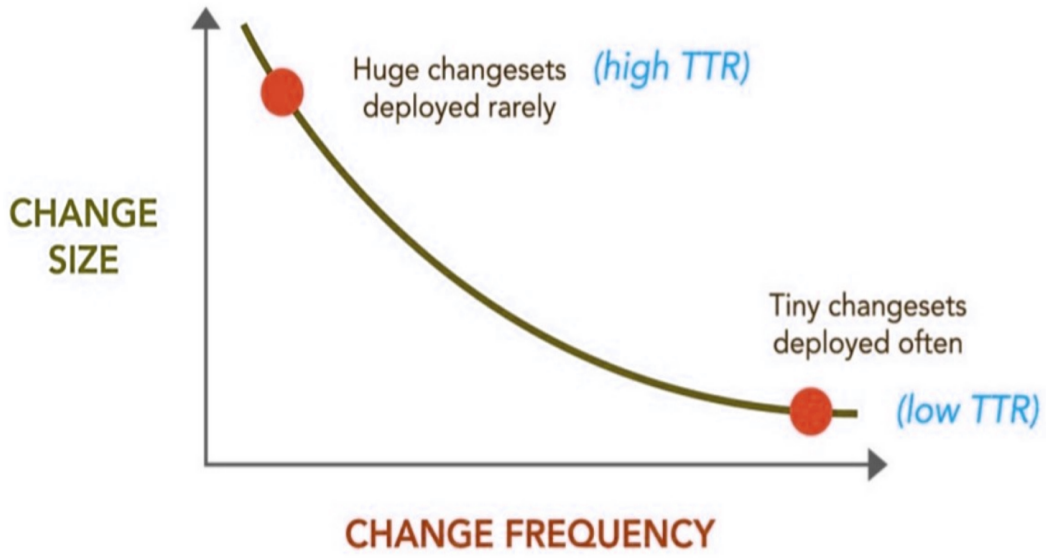
Continuous delivery is an approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time. It aims at building, testing, and releasing software faster and more frequently.
The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for continuous delivery.

Important note : Continuous Delivery ≠ Continuous Deployment - continuous delivery is sometimes confused with continuous deployment. Continuous deployment means that every change is automatically deployed to production.

Continuous delivery means that the team ensures every change can be deployed to production but may choose not to do it, usually due to business reasons. In order to do continuous deployment one must be doing continuous delivery

The key ideas behind continuous deliveries are:

- **The more often you deploy, the more you master the deployment process and the better you automate it**. If you have to do something 3 times a day, you **will** make it bullet proof and reliable soon enough, when you will be fed up of fixing the same issues over and over again.
- **The more often you deploy, the smallest will be the changesets you deploy** and hence the smallest will be the risk of something going wrong, or the chances of losing control over the changesets
- **The more often you deploy, the best will be your TTR (Time to Repair / Resolution)** and hence the sooner will be the feedback you will get from your business users regarding that feature and the easier it will be to change some things here and there to make it perfectly fit their needs (TTR is very similar to TTM in this regards).



(Source : Ops Meta-Metrics: The Currency You Pay For Change - http://fr.slideshare.net/jallspaw/ops-metametrics-the-currency-you-pay-for-change-4608108)

But continuous delivery is more than building a shippable, production-ready version of the product as often as possible. Continuous delivery refers to 3 key practices:

- Learn from the fields
- Automation

- Deploy more often

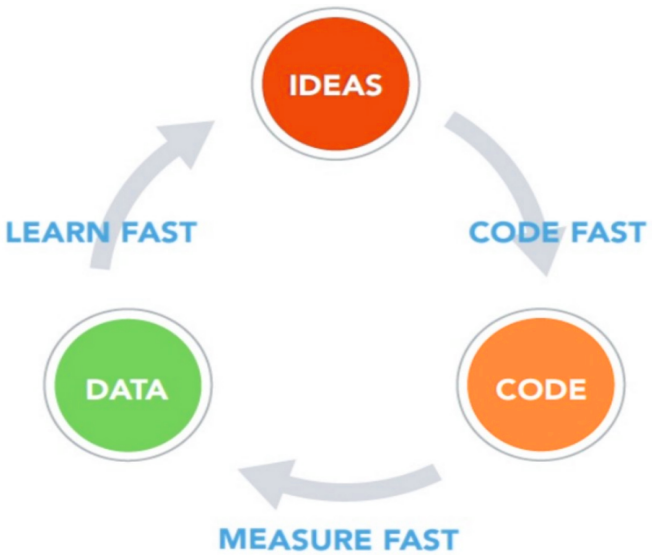
4.3.1 Learn from the field

Continuous Delivery is key to be able to **learn from the field**. There is no truth in the development team, the truth lies in the head of the business users. Unfortunately, no one is able to really clearly express his mind, his will in a specification document, no matter the time he dedicates to this task. This is why Agility attempts to put the feature in the hands of the users to get their feedback as soon as possible, at all cost.
Doing Continuous delivery, as far as continuous deployment, and hence reducing lead time to its minimal possible value, is key to be able to learn the truth from the users, as soon as possible

But the truth doesn't come out in the form of a formal user feedback. One should never trust its users or rely on formal feedback to learn from users. One should trust its own measures.
Measure obsession is a very important notion from the *Lean Startup* movement but it's also very important in DevOps. One should measure everything! Finding the right metrics enabling the team to learn about the success or failures of an approach, about what would be better and what has the most success can be sometimes tricky. One should always take too many measures instead of missing the one that would enable the team to take an enlightened decision.

Don't think, know! And the only way to know is to measure, measure everything: response times, user think times, count of displays, count of API calls, click rate, etc. but not only. Find out about all the metrics that can give you additional insights about the user perception of a feature and measure them, all of them!

This can be represented as follows:



4.3.2 Automation

Automation has already been discussed above in section 4.2 Infrastructure as Code.

I just want to emphasize here that continuous delivery is impossible without a properly and 100% automation of all infrastructure provisioning and deployment related tasks.
This is very important, let me repeat it once more: setting up an environment and deploying a production ready version of the software should take one click, one command, it should be entirely automated. Without it, it's impossible to imagine deploying the software several times a day.

In section 4.3.5 Zero Downtime Deployments below we will mention additional important techniques helping Continuous Delivery as well.

4.3.3 Deploy more often

The DevOps credo is:

"If it hurts, do it more often !"

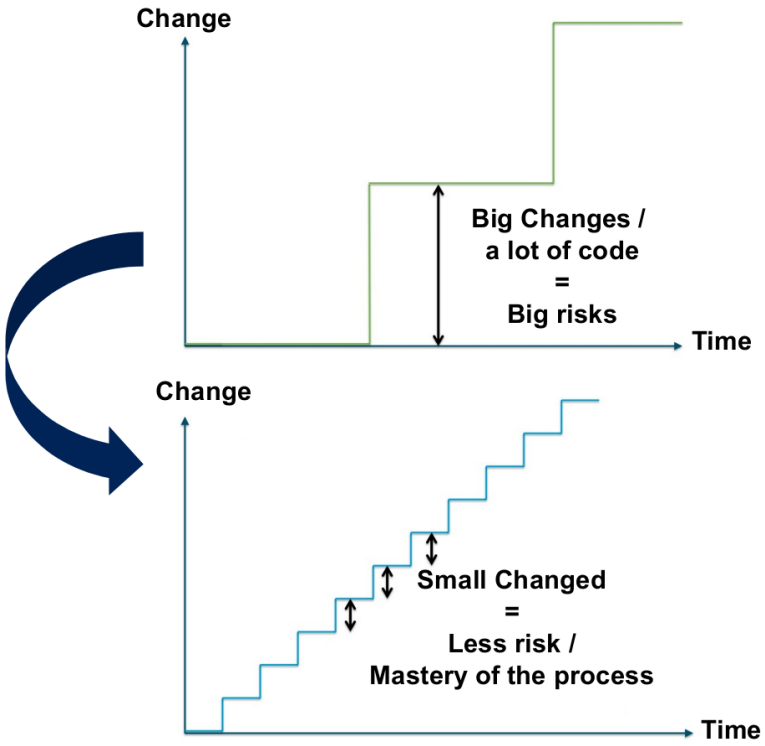
This idea of doing painful things more frequently is very important in agile thinking. Automated Testing, refactoring, database migration, specification with customers, planning, releasing - all sorts of activities are done as frequently as possible.

There are three good reasons for that:

1. Firstly most of these tasks become much more difficult as the amount of work to be done increases, but when broken up into smaller chunks they compose easily.
   Take Database migration for instance: specifying a large database migration involving multiple tables is hard and error prone. But if you take it one small change at a time, it becomes much easier to get each one correct.
   Furthermore you can string small migrations together easily into a sequence. Thus when one decomposes a large migration into a sequence of little ones, it all becomes much easier to handle. (As a sidenote, this is the essence of database refactoring)

2. The second reason is *Feedback*. Much of agile thinking is about setting up feedback loops so that we can learn more quickly. Feedback was already an important and explicit value of Extreme Programming. In a complex process, like software development, one has to frequently check where one stands and make course corrections. To do this, one must look for every opportunity to add feedback loops and increase the frequency with which one gets feedback so one can adjust more quickly.

3. The third reason is *practice*. With any activity, we improve as we do it more often. Practice helps to iron out the kinks in the process, and makes one more familiar with signs of something going wrong. If you reflect on what you are doing, you also come up with ways to improve your practice.
   With software development, there's also in addition the potential for automation. Once one has done something a few times, it's easier to see how to automate it, and more importantly one becomes more motivated to automate it. Automation is especially helpful because it can increase speed and reduce the chance for error.



Now one question remains : **how often to deliver with DevOps ?**

There is no straight answer to that. It really depends on the product, the team, the market, the company, the users, the operation needs, etc.
My best answer would be as follows: If you don't deliver at least every 2 weeks - or at the end of your sprint duration period - you do not even do Agile, not to speak of DevOps.
DevOps encourages to deliver as frequently as possible. In my understanding (please challenge that in the comments if you like), you should train your team to be able to deliver as frequently as possible. A sound approach, the one I'm using with my team is to deliver twice a day on a QA environment. The delivery process is fully automated: twice a day, at noon and at midnight, the machinery starts, builds the software components, runs integration tests, builds the Virtual Machines, start them, deploys the software components, configures them, runs functional tests, etc.

4.3.4 Continuous Delivery requirements

What does one need **before** being able to move to Continuous Delivery?
My checklist, in a raw fashion :

- Continuous integration of both the software components development as well as the platform provisioning and setup.
- TDD - Test Driven Development. This is questionable ... But in the end let's face it: TDD is really the single and only way to have an acceptable coverage of the code and branches with unit tests (and unit tests makes is so much easier to fix issues than integration or functional tests).
- Code reviews ! At least codereviews ... pair programming would be better of course.
- Continuous auditing software - such as Sonar.
- Functional testing automation on production-level environment
- Strong non-functional testing automation (performance, availability, etc.)
- Automated packaging and deployment, independent of target environment

Plus sound software development practices when it comes to managing big features and evolutions, such as *Zero Downtime Deployments* techniques.

4.3.5 Zero Downtime Deployments

"Zero Downtime Deployment (ZDD) consists in deploying a new version of a system without any interruption of service."

ZDD consists in deploying an application in such a way that one introduces a new version of an application to production without making the user see that the application went down in the meantime. From the user's and the company's point of view it's the best possible scenario of deployment since new features can be introduced and bugs can be eliminated without any outage.

I'll mention 4 techniques:

1. Feature Flipping
2. Dark launch
3. Blue/Green Deployments
4. Canari release

Feature flipping

Feature flipping allows to enable / disable features while the software is running. It's really straightforward to understand and put in place: simply use a configuration properly to entirely disable a feature from production and only activate it when its completely polished and working well.

For instance to disable or activate a feature globally for a whole application:

```
if Feature.isEnabled('new_awesome_feature')
    # Do something new, cool and awesome
else
    # Do old, same as always stuff
end
```

Or if one wants to do it on a per-user basis:

```
if Feature.isEnabled('new_awesome_feature', current_user)
    # Do something new, cool and awesome
else
    # Do old, same as always stuff
end
```

Dark Launch

The idea of *Dark Launch* is to use production to simulate load!

It's difficult to simulate load of a software used by hundreds of millions of people in a testing environment.
Without realistic load tests, it's impossible to know if infrastructure will stand up to the pressure.

Instead of simulating load, why not just deploy the feature to see what happens without disrupting usability?
Facebook calls this a *dark launch* of the feature.

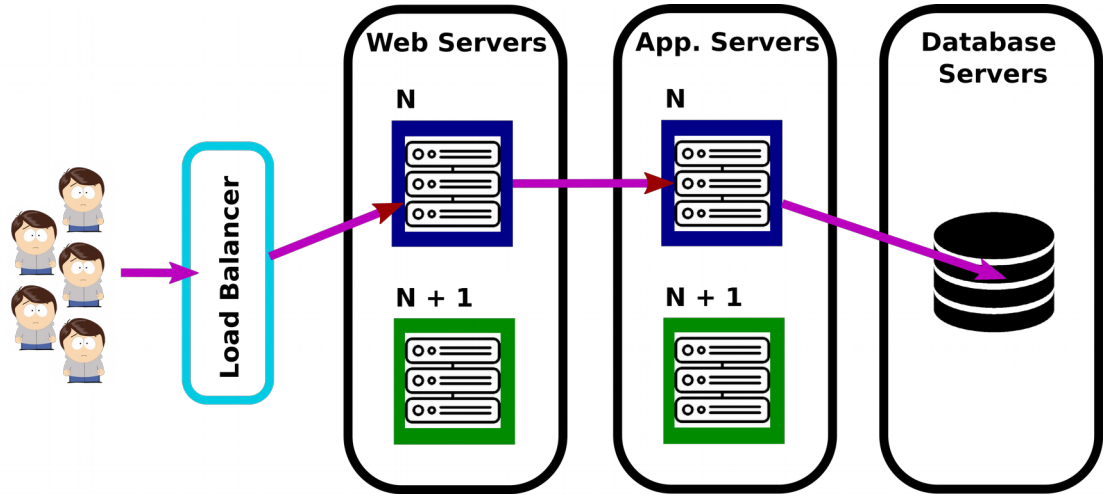
Let's say you want to turn a static search field used by 500 million people into an autocomplete field so your users don't have to wait as long for the search results. You built a web service for it and want to simulate all those people typing words at once and generating multiple requests to the web service.
The dark launch strategy is where you would augment the existing form with a hidden background process that sends the entered search keyword to the new autocomplete service multiple times.
If the web service explodes unexpectedly then no harm is done; the server errors

would just be ignored on the web page. But if it does explode then, great, you can tune and refine the service until it holds up.

There you have it, a real world load test.

Blue/Green Deployments

Blue/Green Deployments consists in building a second complete line of production for version N + 1. Both development and operation teams can peacefully build up version N + 1 on this second production line.
Whenever the version N + 1 is ready to be used, the configuration is changed on the load balancer and users are automatically and transparently redirected to the new version N + 1.
At this moment, the production line for version N is recovered and used to peacefully build version N + 2.
And so on.



(Source : Les Patterns des Géants du Web – Zero Downtime Deployment - http://blog.octo.com/zero-downtime-deployment/)

This is quite effective and easy but the problem is that it requires to double the infrastructure, amount of servers, etc.
Imagine if Facebook had to maintain a complete second set of its hundreds of thousands of servers.

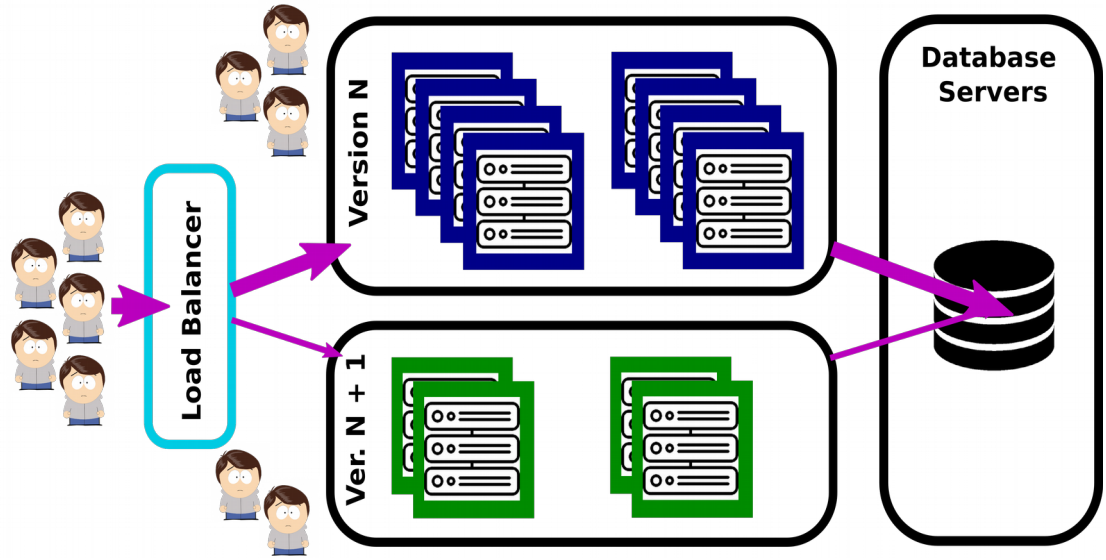
So there is some room for something better.

Canari release

Canari release is very similar in nature to *Blue/Green Deployments* but it addresses the problem to have multiple complete production lines.
The idea is to switch users to the new version in an incremental fashion : as more servers are migrated from the version N line to the version N + 1 line, an equivalent

proportion of users are migrated as well.
This way, the load on every production line matches the amount of servers.

At first, only a few servers are migrated to version N + 1 along with a small subset of the users. This also allows to test the new release without risking an impact on all users.
When all servers have eventually been migrated from line N to line N + 1, the release is finished and everything can start all over again for release N + 2.



(Source : Les Patterns des Géants du Web – Zero Downtime Deployment - http://blog.octo.com/zero-downtime-deployment/)

4.4 Collaboration

Agile software development has broken down some of the silos between requirements analysis, testing and development. Deployment, operations and maintenance are other activities which have suffered a similar separation from the rest of the software development process. The DevOps movement is aimed at removing these silos and encouraging collaboration between development and operations.
Even with the best tools, DevOps is just another buzzword if you don't have the right culture.

The primary characteristic of DevOps culture is increased collaboration between the roles of development and operations. There are some important cultural shifts, within teams and at an organizational level, that support this collaboration.

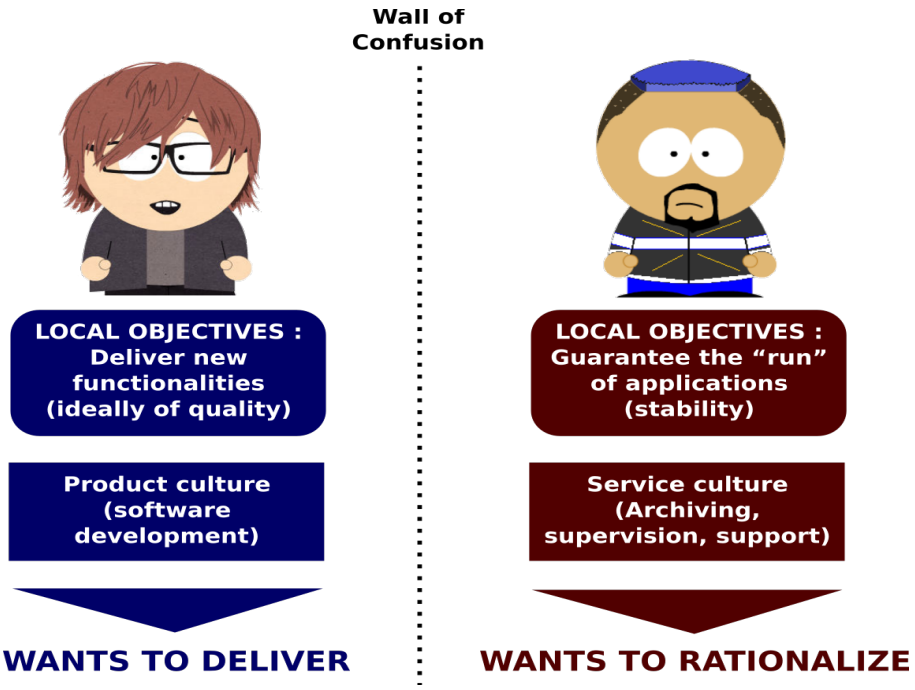
This addresses a very important problem that is best illustrated with the following meme:(Souce : DevOps Memes @ EMCworld 2015 - http://fr.slideshare.net/bgracely/devops-memes-emcworld-2015)

Team play is so important to DevOps that one could really sum up most of the methodology's goals for improvement with two C's: collaboration and communication. While it takes more than that to truly become a DevOps workplace, any company that has committed to those two concepts is well on its way.

But why is it so difficult ?

4.4.1 The wall of confusion

Because of the wall of confusion :



In a traditional development cycle, the development team kicks things off by "throwing" a software release "over the wall" to Operations.
Operations picks up the release artifacts and begins preparing for their deployment.
Operations manually hacks the deployment scripts provided by the developers or, most of the time, maintains their own scripts.
They also manually edit configuration files to reflect the production environment, which is significantly different than the Development or QA environments.
At best they are duplicating work that was already done in previous environments, at worst they are about to introduce or uncover new bugs.

The IT Operations team then embarks on what they understand to be the currently correct deployment process, which at this point is essentially being performed for the first time due to the script, configuration, process, and environment differences between Development and Operations.
Of course, somewhere along the way a problem occurs and the developers are called in to help troubleshoot. Operations claims that Development gave them faulty code.
Developers respond by pointing out that it worked just fine in their environments, so it must be the case that Operations did something wrong.
Developers are having a difficult time even diagnosing the problem because the configuration, file locations, and procedure used to get into this state is different then what they expect. Time is running out on the change window and, of course, there isn't a reliable way to roll the environment back to a previously known good state.

So what should have been an eventless deployment ended up being an all-hands-on-deck fire drill where a lot of trial and error finally hacked the production environment into a usable state.

It **always** happens this way, always.

Here comes DevOps

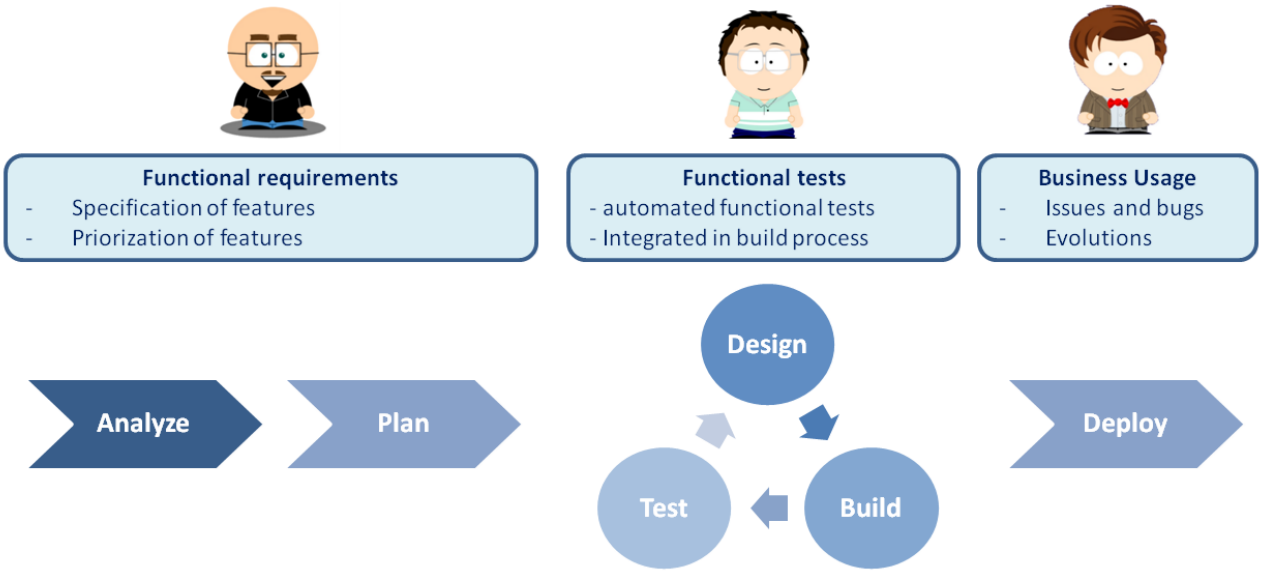
DevOps helps to enable IT alignment by aligning development and operations roles and processes in the context of shared business objectives. Both development and operations need to understand that they are part of a unified business process. DevOps thinking ensures that individual decisions and actions strive to support and improve that unified business process, regardless of organizational structure.

Even further, as Werner Vogel, CTO of Amazon, said in 2014 :

"You build it, you run it."

4.4.2 Software Development Process

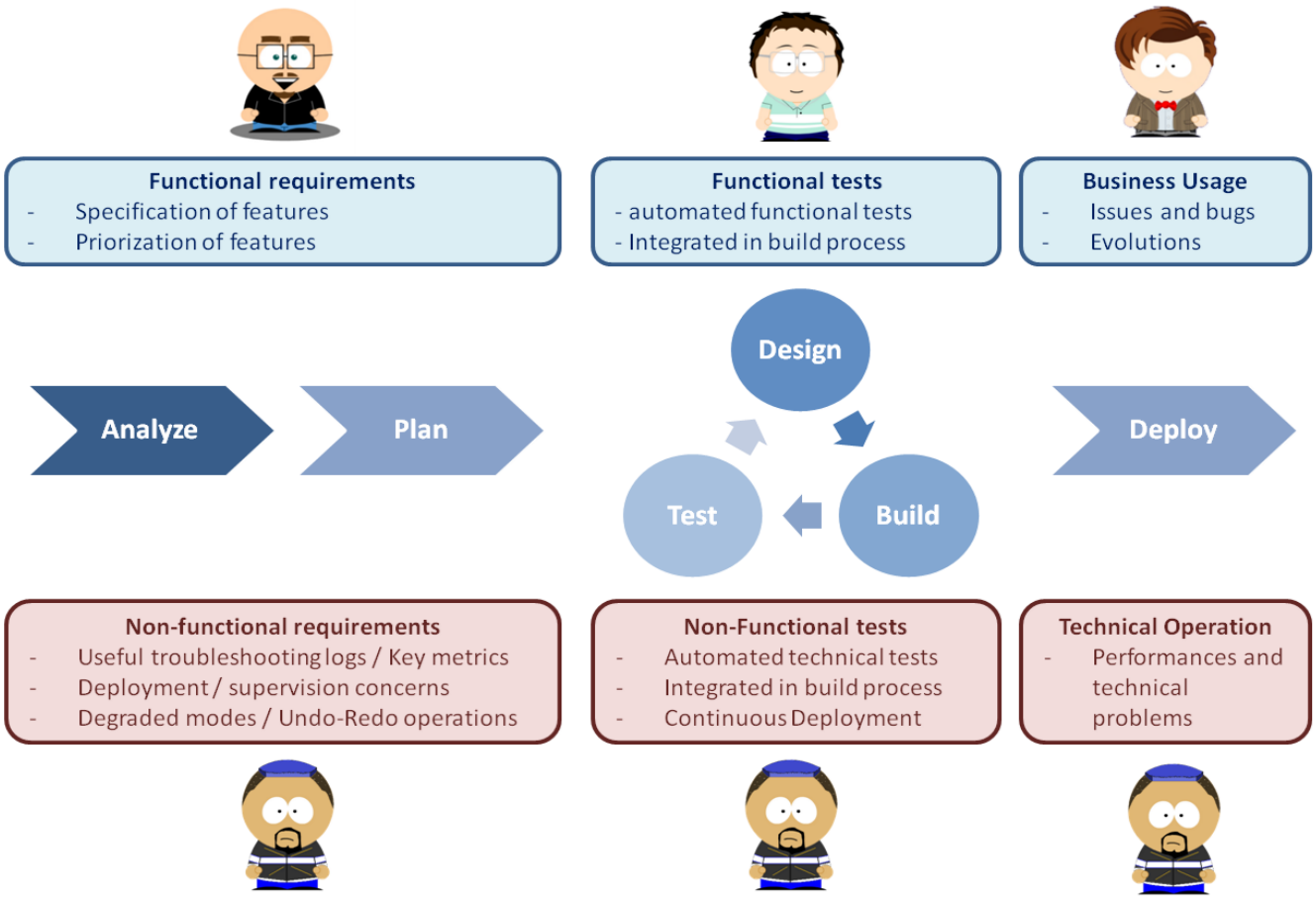
Below is a simplified view of how the Agile Software Development Process usually looks like.



Initially the business representatives work with the Product Owner and the Architecture Team to define the software, either through Story Mapping with User stories or with more complete specification.
Then the development team develops the software in short development sprints, shipping a production ready version of the software to the business users at the end

of every sprint in order to capture feedback and get directions as often and as much as possible.
Finally, after every new milestone, the software is deployed for wide usage to all business lines.

The big change introduced by DevOps is the understanding that **operators are the other users of the software !** and as such they should be fully integrated in the Software Development Process.
At specification time, operators should give their non-functional requirements just as business users give their functional requirement. Such non-functional requirements should be handled with same important and priority by the development team.
At implementation time, operators should provide feedback and non-functional tests specifications continuously just as business users provides feedback on functional features.
Finally, operators become users of the software just as business users.

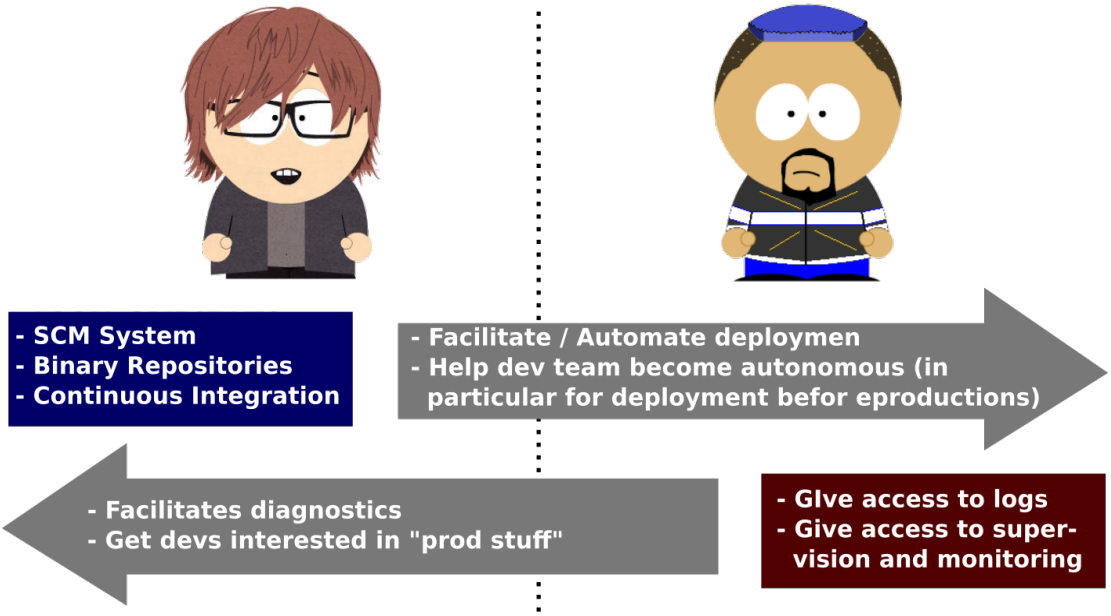


With DevOps, operators become fully integrated in the Software Development Process.

4.4.3 Share the Tools

In traditional corporations, teams of operators and teams of developers use specific, dedicated and well separated set of tools.
Operators usually don't want do know anything about the dev team SCM system as well as continuous integration environment. They perceive this as additional work and fear to be overwhelmed by developer requests if they put their hands on this systems as well. After all, they have well enough to do by taking care of production systems.
Developers, on their side, usually have no access to production system logs and monitoring tools, sometimes due to lack of will on their side, sometimes for regulation or security concerns.

This needs to change! DevOps is here for that.



(Source : Mathieu Despriee - OCTO Technology - Introduction to DevOps)

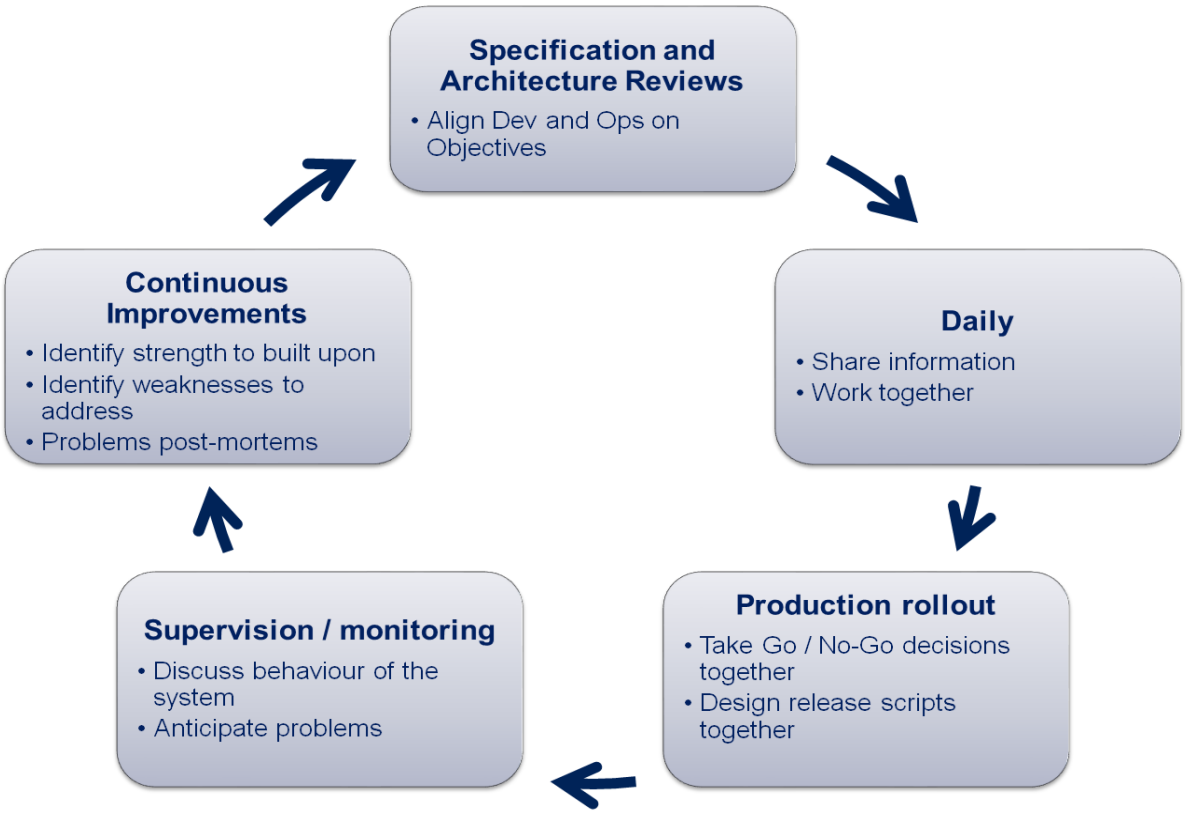
One should note that this can be difficult to achieve. For instance for regulation or security reasons, logs may need to be anonymized on the fly, supervision tools need to be secured to avoid an untrained and forbidden developer to actually change something in production, etc. This may take time and cost resources. But the gain in efficiency is way greater that the required investment, and the ROI of this approach for the whole company is striking.

4.4.4 Work Together

A fundamental philosophy of DevOps is that developers and operations staff must work closely together on a regular basis.

An implication is that they must see one other as important stakeholders and actively seek to work together.

Inspired from the XP practice "*onsite customer*", which motivates agile developers to work closely with the business, disciplined agilists take this one step further with the practice of active stakeholder participation, which says that developers should work closely with all of their stakeholders, **including operations and support staff**. This is a two-way street: operations and support staff must also be willing to work closely with developers.



In addition, other collaboration leads:

- Have operators taking part in Agile rituals (Daily scrum, sprint planning, sprint retro, etc.)
- Have devs taking part in production rollouts
- Share between Dev and Ops objectives of continuous improvement

4.5 Conclusion

DevOps is a revolution that aims at addressing the *wall of confusion* between development teams and operation teams in big corporations having large IT departments where these roles are traditionally well separated and isolated.

Again, I've spent two thirds of my fifteen years career working for such big institutions, mostly financial institutions, and I have been able to witness this wall of confusion on a daily basis. Some sample things I got to hear:

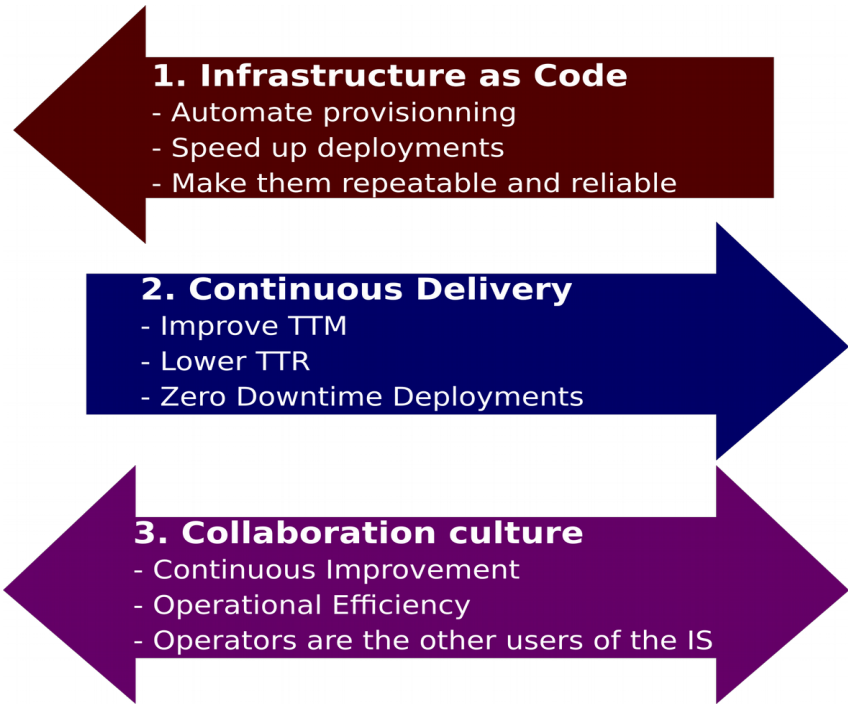
- "*It worked fine on my Tomcat. Sorry but I know nothing about your Websphere thing. I really can't help you.*" (a dev)
- "*No we cannot provide you with an extract of this table from the production database. It contains confidential customer-related data.*" (an ops)

And many more examples such as those every day .... every day!

Happily DevOps is several years old and increasingly even these very traditional corporations are moving in the right direction by adopting DevOps principles and practices. But a lot remains to be done.

Now what about smaller corporations that don't necessarily have split functions between developers and operators?
Adopting DevOps principles and practices, such as deployment automation, continuous delivery and feature flipping still brings a lot.

I would summarize DevOps principles this way:



DevOps is simply a step further towards Scaling Agility!

5. The Lean Startup - A focus on Practices

A few years ago, I worked intensively on a pet project : AirXCell.
What was at first some framework and tool I had to write to work on my Master Thesis dedicated to Quantitative Research in finance, became after a few months somewhat my most essential focus in life.
Initially it was really intended to be only a tool providing me with a way to have a Graphical User Interface on top of all these smart calculations I was doing in R. After my master thesis, I surprised myself to continue to work on it, improving it a little here and a little there. I kept on doing that until the moment I figured I was dedicated several hours to it every day after my day job.
Pretty soon, I figured I was really holding an interesting software and I became convinced I could make something out of it and eventually, why not, start a company.

And of course I did it all wrong.

Instead of **finding out first if there was a need and a market for it**, and then *what should I really build to answer this need*, I spent hours every day and most of my week-ends developing it further towards what I was convinced was the minimum set of feature it should hold before I actually try to meet some potential customers to tell them about it.
So I did that for more than a year and a half until I came close to burn-out and send it all to hell.

Now the project hasn't evolve for three years. The thing is that I just don't want to hear about it anymore. I burnt myself and I am just disgusted about it. Honestly it is pretty likely that at the time of reading this article, the link above is not even reachable anymore.
When I think of the amount of time I ~~invested~~ wasted in it, and the fact that even now, three years after, I still just don't want to hear anything about this project anymore, I feel so ashamed. Ashamed that I didn't take a leap backwards, read a few books about startup creation, and maybe, who knows, discover *The Lean Startup* movement before.
Even now, I still never met any potential customer, any market representative. Even worst: I'm still pretty convinced that there is a need and a market for such a tool. But I'll never know for sure.

Such stories, and even worst, stories of startups burning millions of dollars for nothing in the end, happen every day, still today.

Some years ago, Eric Ries, Steve Blank and others initiated ***The Lean Startup*** movement. *The Lean Startup* is a movement, an inspiration, a set of principles and

practices that any entrepreneur initiating a startup would be well advised to follow. Projecting myself into it, I think that if I had read Ries' book before, or even better Blank's book, I would maybe own my own company today, around AirXCell or another product, instead of being disgusted and honestly not considering it for the near future.
In addition to giving a pretty important set of principles when it comes to creating and running a startup, *The Lean Startup* also implies an extended set of Engineering practices, especially software engineering practices.

This article focuses on presenting and detailing these **Software Engineering Practices from the Lean Startup Movement** since, in the end, I believe they can benefit from any kind company, from initiating startup to well established companies with Software Development Activities.
By Software Engineering practices, I mean software development practices of course but not only. Engineering is also about analyzing the features to be implemented, understanding the customer need and building a successful product, not just writing code.

5.1. The Lean Startup

The Lean Startup is today a movement, initiated and supported by some key people that I'll present below.
But it's also a framework, an inspiration, an approach, a methodology with a set of fundamental principles and practices for helping entrepreneurs increase their odds of building a successful startup.
Lean Startup cannot be thought as a set of tactics or steps. Don't expect any checklist (well, at least not only checklists) or any recipe to be applied blindly.

The approach is built around two main objectives:

1. Teaching entrepreneurs how to drive a startup through the process of steering (*Build-Measure-Learn* feedback loop).
2. Enabling entrepreneurs to scale and grow the business with maximum acceleration

Lean Startup Practices

The Lean Startup methodology can be divided in two sets of practices:

1. The **steering practices** : designed to minimize the total time through the Build-Measure-Learn feedback loop and
2. The **acceleration practices** : which allow Lean Startups to grow without sacrificing the startup's speed and agility

This is developed further in 5.2 The four steps to the Epiphany.

5.1.1 Origins

The Lean Movement

Lean thinking is a **business methodology** that aims to provide a new way to think about how to organize human activities to deliver more benefits to society and value to individuals while **eliminating waste**.
Lean thinking is a **new way of thinking any activity** and seeing the waste inadvertently generated by the way the process is organized

The aim of lean thinking is to create a **lean enterprise**, one that **sustains growth** by aligning customer satisfaction with employee satisfaction, and that **offers innovative products** or services profitably while **minimizing unnecessary over-costs** to customers, suppliers and the environment.

The Lean Movement finds its roots in Toyotism and values **performance** and **continuous improvement**. The Lean Movement really rose in the early 90's and the lean tradition has adopted a number of practices from Toyota's own learning curve.
Some worth to mention:

- **Kaizen** (Continuous Improvement) : is a strategy where employees at all levels of a company work together pro-actively to achieve regular, incremental improvements to the manufacturing process. The point of Kaizen is that improvement is a normal part of the job, not something to be done "when there is time left after having done everything else", that should involve the company as a whole, from the CEO to the assembly line workers.
- **Kanban** (Visual Billboard) : is a scheduling system and visual management tool used in *Lean Manufacturing* to signal steps in their manufacturing process. The system's highly visual nature allows teams to communicate more easily on what work needed to be done and when. It also standardizes cues and refines processes, which helps to reduce waste and maximize value.

Plus strong emphasizes on *Autonomation*, *Visualization*, etc.

The Lean Startup

The author, I should say *initial author*, of the Lean Startup methodology, Eric Ries, explains in his book "*The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*", that traditional management practices and ideas are not adequate to tackle the entrepreneurial challenges of startups.

By exploring and studying new and existing approaches, Ries found that adapting *Lean thinking* to the context of entrepreneurship would allow to discern between *value-creating activities* versus *waste*.

Thus, Ries, decided to apply lean thinking to the process of innovation. After its initial development and some refinement, as he states, **the Lean Startup** represents a new approach to creating continuous innovation that builds on many previous management and product development ideas, including lean manufacturing, design thinking, customer development, and agile development.

5.1.2 The movement

I would highly recommend this enlightening article - The History Of Lean Startup - that does a pretty great job explaining how and why the following guys got together and initiated the *Lean Startup Movement* (aside from a few things I do not agree with).

Blank, Ries, Osterwalder and Maurya are the founders or initiators of the *Lean Startup Movement*. Eric Ries is considered as the leader of the movement, while Steve Blank considers himself as its godfather.
Osterwalder and Maurya's work on business models is considered to fill a gap in Ries and Blank's work on processes, principles and practices. In Steve Blank's "The four Steps the the Epiphany", the business model section is a vague single page.
Furtherly, Maurya's "*Running Lean*" magnificently completes Blank's work on *Customer Development*. We'll get to that.

5.1.3 Principles

In my opinion, the most fundamental aspect of Lean Startup is the *Build-Measure-Learn* loop, or, in the context of the *Customer Development Process*, the *Customer Discovery - Customer Validation - Re-adapt the product* loop.

The idea is to be able to loop in laboratory mode, mostly with prototypes and interviews, in an iterative research process, with as little costs as possible, about the product to be developed. A startup should spend as little investment as possible in terms of product development as long as it has no certainty in regards to the customer needs, the right product to be developed, the potential market, etc.
This is really key, before hiring employees and starting to develop a product, the entrepreneur should have certainty about the product to be developed and its market.
Premature scaling is the immediate cause of the Death Spiral.

Before digging any further into this, below are the essential principles that characterize *The Lean Startup* approach, as reported by Eric Ries' book.

Entrepreneurs are everywhere

You don't have to work in a garage to be in a startup. The concept of entrepreneurship includes anyone who works within Eric Ries' definition of a startup, which I like very much BTW.
His definition is as follows :

A startup is a human institution designed to create new products and services under conditions of extreme uncertainty.

That means entrepreneurs are everywhere and the Lean Startup approach can work in any size company, even a very large enterprise, in any sector or industry.

Entrepreneurship is management

A startup is an institution, not just a product, and so it requires a new kind of management specifically geared to its context of extreme uncertainty.
In fact, Ries believes "*entrepreneur*" should be considered a job title in all modern companies that depend on innovation for their future growth

Validated learnings

Startups exist not just to make stuff, make money, or even serve customers. They exist to learn how to build a sustainable business. This learning can be validated scientifically by running frequent experiments that allow entrepreneurs to test each element of their vision.

Innovation accounting

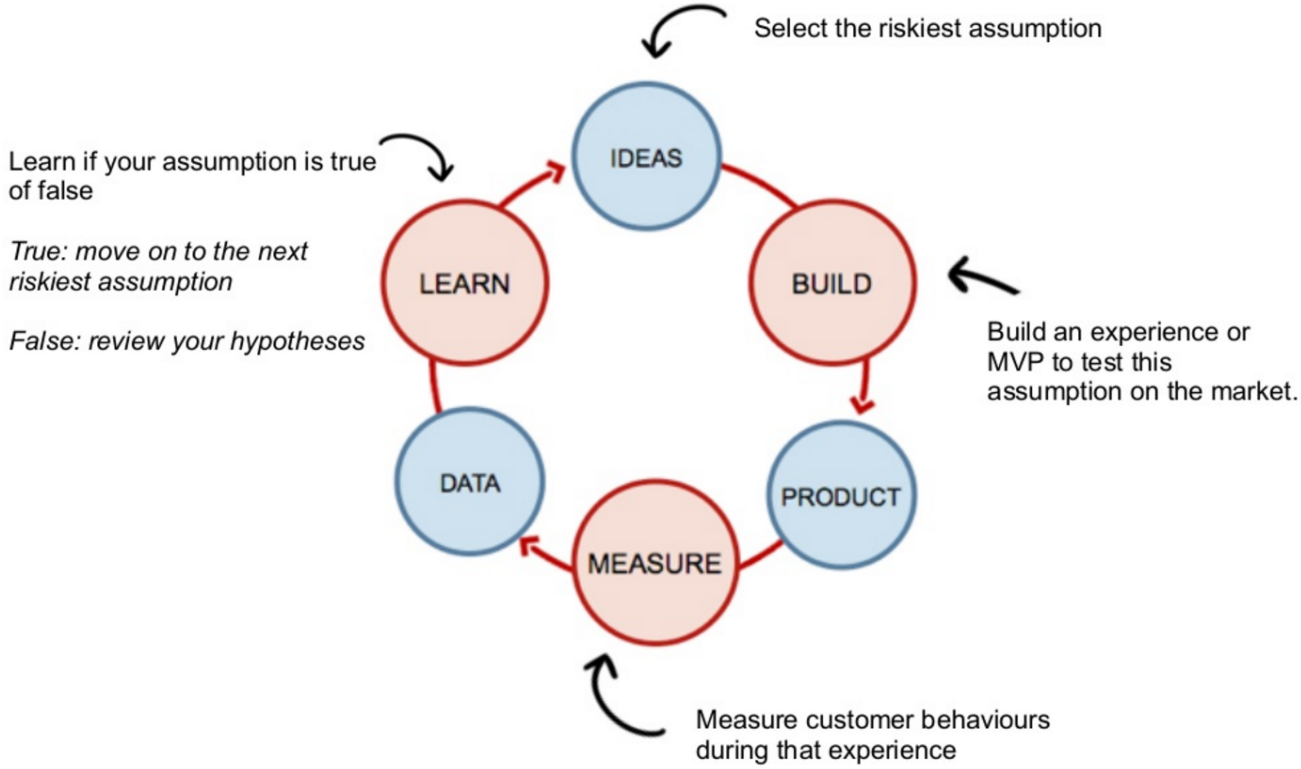
To improve entrepreneurial outcomes and hold innovators accountable, we need to focus on the boring stuff: how to measure progress, how to set up milestones, and how to prioritize work. This requires a new kind of accounting designed for startups- and the people who hold them accountable.

Build-Measure-Learn

The fundamental activity of a startup is to turn ideas into products, measure how customers respond, and then learn whether to **pivot or persevere**. All successful startup processes should be geared to accelerate that **feedback loop**.

5.1.4 The Feedback Loop

The feedback loop is represented as below.
The five-part version of the *Build-Measure-Learn* schema helps us see that the real intent of building is to test "*ideas*" - not just to build blindly without an objective. The need for "*data*" indicates that after we measure our experiments we'll use the data to further refine our learning. And the new learning will influence our next ideas. So we can see that the goal of Build-Measure-Learn isn't just to build things, the goal is to build things to validate or invalidate the initial idea.



Again, the goal of *Build-Measure-Learn* is not to build a final product to ship or even to build a prototype of a product, but to **maximize learning** through incremental and iterative engineering.
In this case, learning can be about product features, customer needs, distribution channels, the right pricing strategy, etc.
The "*build*" step refers to building an 5.3.2.1 MVP (Minimal Viable Product).
It's critical here to understand that an MVP does not mean *the product with fewer features*. Instead, an MVP should be seen as the simplest thing that you can show to

customers to get the most learning at that point in time. Early on in a startup, an MVP could well simply be a set of Powerpoint slides with some fancy animations, or whatever is sufficient to demonstrate a set of features to customers and get feedback from it. Each time one builds an MVP one should also define precisely what one wants to test/measure.
Later, as more is learned, the MVP goes from low-fidelity to higher fidelity, but the goal continues to be to maximize learning not to build a beta/fully featured prototype of a product or a feature.

In the end, the *Build-Measure-Learn* framework lets startups be fast, agile and efficient.

5.1.5 Business Model Canvas and Lean Canvas

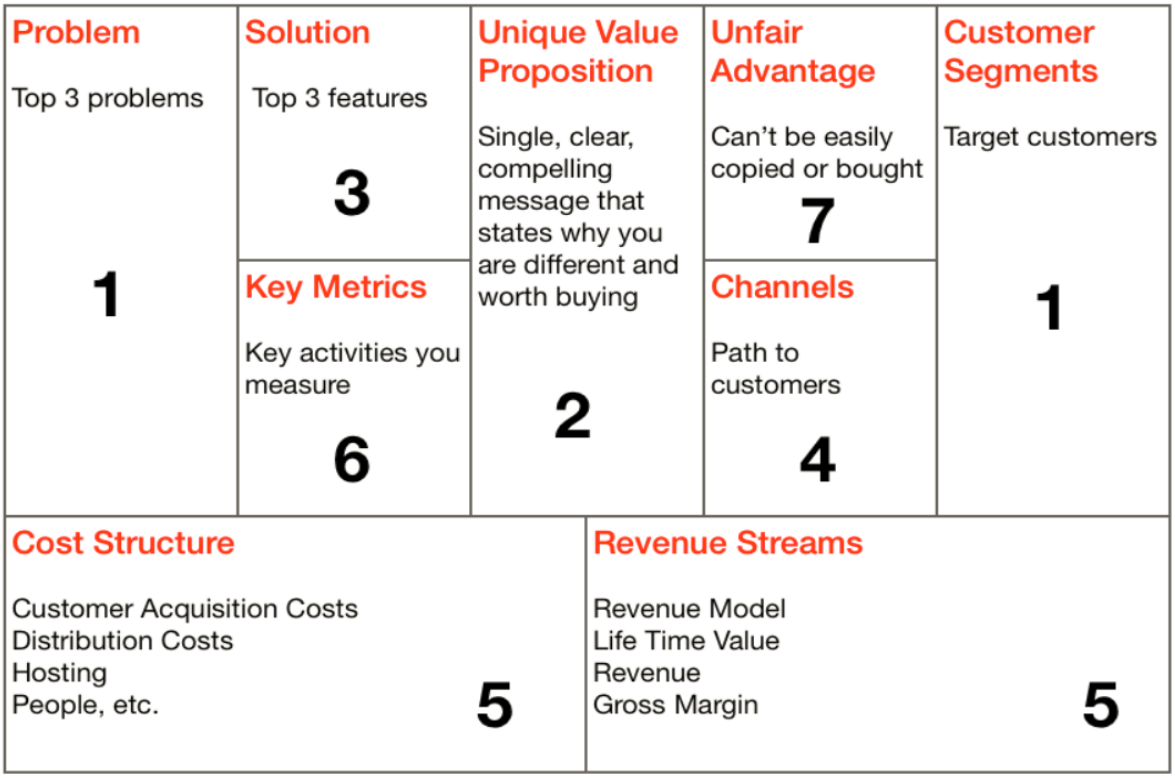
Evolution on Business Models and the relative processes were surprisingly missing or poorly addressed from Ries' and Blank's initial work.
Fortunately, Osterwalder and Maurya caught up and filled the gap.

Business Model Canvas

The Business Model Canvas is a strategic management template invented by Alexander Osterwalder and Yves Pigneur for developing new business models or documenting existing ones.
It is a visual chart with elements describing a company's value proposition, infrastructure, customers, and finances. It assists companies in aligning their activities by illustrating potential trade-offs.

Lean Canvas

The Lean Canvas is a version of the Business Model Canvas adapted by Ash Maurya specifically for startups. The Lean Canvas focuses on addressing broad customer problems and solutions and delivering them to customer segments through a unique value proposition.



So how should one use the Lean Canvas?

1. **Customer Segment and Problem**
   Both Customer Segment and Problem sections should be filled in together. Fill in the list of potential *customers* and *users* of your product, distinguish customers (willing to pay) clearly from users, then refine each and every identified customer segment. Be careful not no try to focus on a too broad segment at first, think of Facebook whose first segment was only Harvard students.
   Fill in carefully the problem encountered by your identified customers.
2. **UVP - Unique Value Proposition**
   The UVP is the unique characteristic of your product or your service making it different from what is already available on the market an that makes it worth the consideration of your customers. Focus on the main problem you are solving and what makes your solution different.
3. **Solution**
   Filling this is initially is tricky, since knowing about the solution for real requires trial and error, build-measure-learn loop, etc. In an initial stage one shouldn't try to be to precise here and keep things pretty open.
4. **Channels**
   This consists in answering: how should you get in touch with your users and

customers ? How do you get them to know about your product ? Indicate clearly your communication channels.

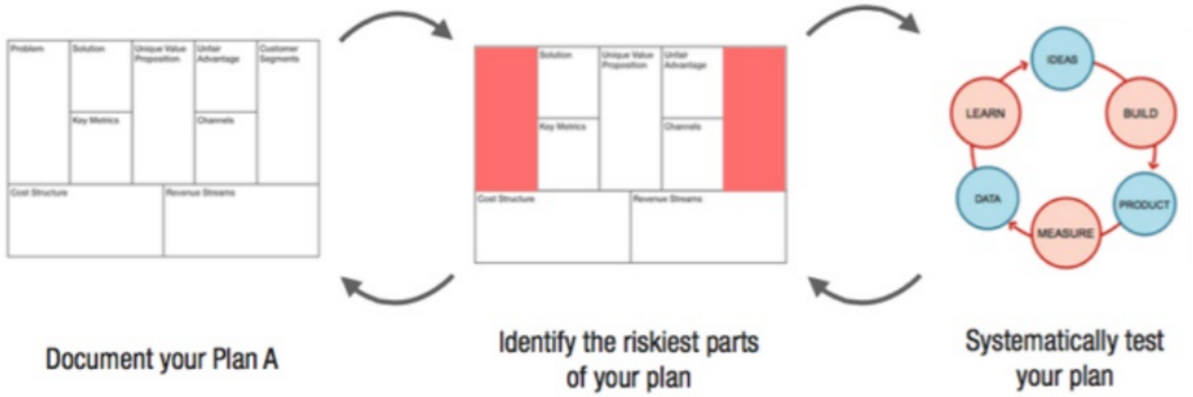
5. **Revenue Stream and Cost Structure**
   Both these sections should also be filled in together.
   At first, at the time of the initial stage of the startup, this should really be focused on the costs and revenues related to launching the MVP (how to interview 50 customers ? Whats the initial burn rate ? etc.)
   Later this should evolve towards an initial startup structure and focus on identifying the *break-even* point by answering the question : how many customers are required to cover my costs ?

6. **Key Metrics**
   Ash Maurya refers to Dave McClure Pirate Metrics to identify the relevant KPIs to be followed :
   Acquisition - How do user find you ?
   Activation - Do user have a great first experience ?
   Retention - Do users come back ?
   Revenue - How do you make money ?
   Referral - Do users tell others ?

7. **Unfair Advantage**
   This consists in indicating the adoption barriers as well as the competitive advantages of your solution. An *unfair advantage* is defined as something that cannot be copied easily neither bought.

Lean Startup : test your plan !

Using the new "Build - Measure - Learn" diagram, the question then becomes, "What hypotheses should I test?". This is precisely the purpose of the initial Lean Canvas,



And it brings us to another definition of a startup:

A startup is a temporary organization designed to *search* for a repeatable and scalable business model.

And once these hypotheses fill the Lean Canvas (Or Business Model Canvas), the key approach is to **run experiments**. This leads us to the next section.

5.1.6 Customer Development

The Customer Development process is a simple methodology for taking new venture hypotheses and getting out of the building to test them. *Customer discovery* (see below) captures the founders' vision and turns it into a series of business model hypotheses. Then it develops a series of experiments to test customer reactions to those hypotheses and turn them into facts. The experiments can be a series of questions you ask customers. Though, most often an MVP to help potential customers understand your solution accompanies the questions.

Startups are building an MVP to learn the most they can, not to get a prototype!

The goal of designing these experiments and minimal viable products is not to get data. The data is not the endpoint. Anyone can collect data. **The goal is to get insight**. The entire point of getting out of the building is to inform the founder's vision.
The insight may come from analyzing customer answers, but it also may come from interpreting the data in a new way or completely ignoring it when realizing that the idea is related to a completely new and disruptive market that even doesn't exist yet.

Customer Development instead of Product Development

More startup fail from a *lack of customers* rather than from a failure of Product Development.

The Customer Development model delineates all the customer-related activities in the early stage of a company into their own processes and groups them into four easy-to-understand steps: *Customer Discovery*, *Customer Validation*, *Customer Creation*, and *Company Building*.
These steps mesh seamlessly and support a startup's ongoing product development activities. Each step results in specific deliverables and involves specific practices.

As its name should communicate, the Customer Development model focuses on developing customers for the product or service your startup is building. Customer Development is really about finding a market for your product. It is built upon the idea that the founder has an idea but he doesn't know if the clients he imagines will buy. He needs to check this point and it is better if he does it soon.

5.2 The four steps to the Epiphany

Shortly put, Steve Blank proposes that companies need a **Customer Development process** that complements, or even in large portions replaces, their *Product Development Process*. The *Customer Development process* goes directly to the theory of Product/Market Fit.
In "*The four steps to the Epiphany*", Steve Blank provides a roadmap for how to get to Product/Market Fit.

5.2.1 Overview

The Path to Disaster: The Product Development Model

The traditional product development model has four stages:

1. concept/seed,
2. product development,
3. beta test,
4. and launch.

That product development model, when applied to startups, suffers from a lot of flaws. They basically boil down to:

- Customers were nowhere in that flow chart
- The flow chart was strictly linear
- Emphasis on execution over learning
- Lack of meaningful milestones for sales/marketing
- Treating all startups alike

What's the alternative? Before we get to that, one final topic is the *technology life cycle adoption curve*, i.e. adoption happens in phases of early adopters (tech enthusiasts, visionaries), mainstream (pragmatists, conservatives), and skeptics. Between each category is a *chasm*, the largest is between the early adopters and the mainstream.
Crossing the chasm is a success problem. But you're not there yet, "customer development" lives in the realm of the early adopter.

The Path to Epiphany: The Customer Development Model

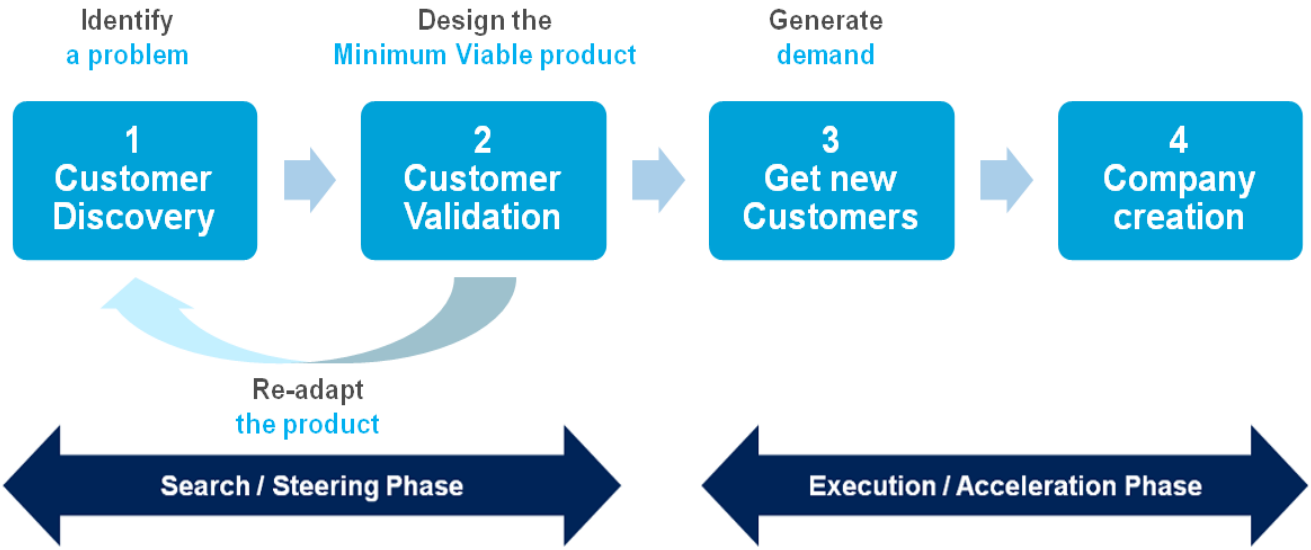
Most startups lack a process for discovering their markets, locating their first customers, validating their assumptions, and growing their business.
The **Customer Development Model** creates the process for these goals.

5.2.2 A 4 steps process

The four stages the *Customer Development Model* are: customer discovery, customer validation, customer creation, and company creation.

1. **Customer discovery**: understanding customer problems and needs
2. **Customer validation**: developing a sales model that can be replicated
3. **Customer creation / Get new Customers**: creating and driving end user demand
4. **Customer building / Company Creation**: transitioning from learning to executing

We can represent them as follows:



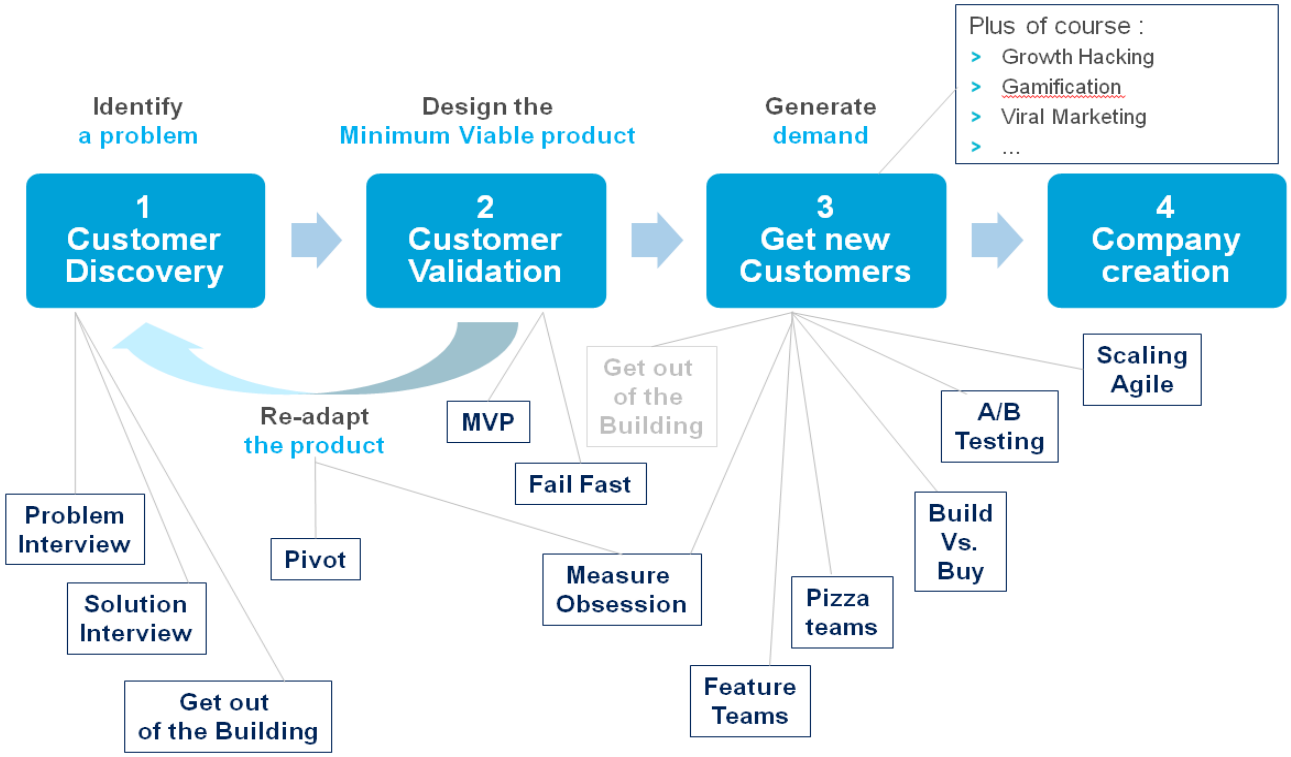
I won't go any further in this article in describing these steps, their purpose and reasons.
To be honest Blank's book is pretty heavy and not very accessible. Happily Blank's did a lot of presentations around his book that one can find on youtube or elsewhere. In addition, there are a lot of excellent summaries and text explanations available online on Blank's book and I let the reader refer to this material should he want more information.

Instead, I want to focus in this article on the **Software Engineering Practices** inferred from The Lean Startup approach, since, again, I believe they are very important for any kind of corporation with an important Software Development activity.
And yet again, Software Engineering practices go beyond solely Software

Development practices, but cover every activity in the company aimed at identifying and developing the product.

5.3 Lean startup practices

So I want to present the most essentials principles and practices introduced and discussed by *the Lean Startup* approach.
These principles and practices are presented on the following schema attached to the stages of the *Customer Development* process where I think they make more sense:



Important notes

- I attached the practices to the step where I think they make more sense, where I think they bring the most added value or should be introduced. But bear in mind that such a *categorization* is highly subjective and questionable. If you yourself believe some practices should be attached to another step, well just leave a comment and move on.
- Also, there are other practices of course. I mention here and will be discussing below the ones that seem the most appealing to me, myself and I. Again my selection is highly subjective and personal. If you think I am missing something important, just leave a comment and move on.

The rest of this paper intends to describe all these engineering - mostly software engineering - practices since, again, at the end of the day, I strongly believe that

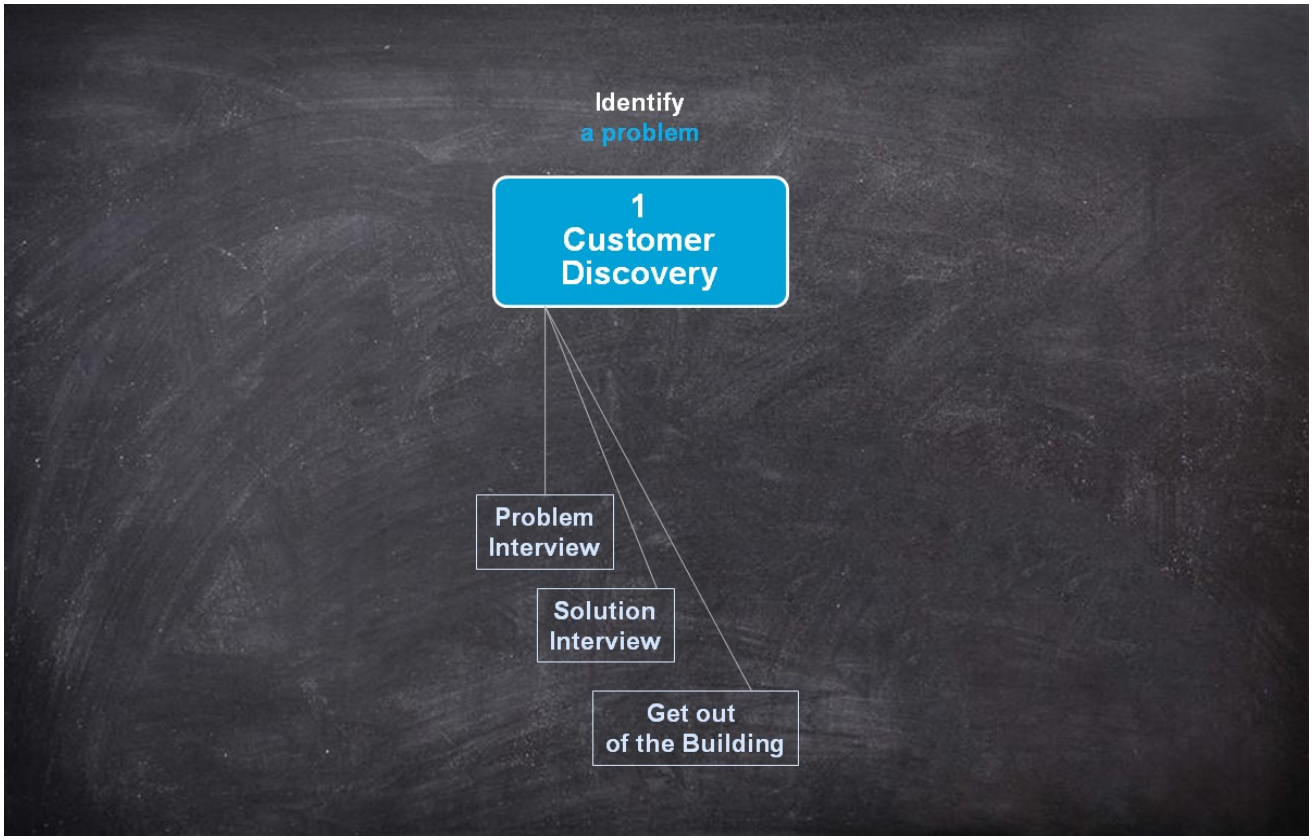
they form the most essential legacy of the Lean Startup movement and that they can benefit any kind of company, not only startups.

5.3.1 Customer Discovery

Customer Discovery, focuses on understanding customer problems and needs. Its really about searching for the *Product-Solution Fit*, turning the founders' initial hypotheses about their market and customers into facts.
The Problem-Solution Fit occurs when entrepreneurs identify relevant insights that can be addressed with a suggested solution. As Osterwalder describes it, this fit happens when there is evidence that customers care about certain problems that need to be solved or needs, and, there is a value proposition designed that addresses those needs.

In Customer Discovery the startup aims at understanding customer problems and needs and, also, to ideate potential solutions that could be valuable based on the findings. Similarly, Osterwalder calls these problems and needs as *jobs, pains and gains*.

The three practices I want to emphasize at this stage are as follows:



5.3.1.1 Get out of the building

If you're not Getting out of the Building, you're not doing Customer Development and Lean Startup.
There are no facts inside the building, only opinions.

If you aren't actually talking to your customers, you aren't doing Customer Development. And talking here is really speaking, with your mouth. Preferably in-person, but if not, a video call would work as well, messaging or emailing doesn't.

As Steve Blank said "*One good customer development interview is better for learning about your customers / product / problem / solution / market than five surveys with 10'000 statistically significant responses.*"

The problem here is that tech people, especially software engineers, try to avoid going out of the building as much as possible. But this is so important. Engineers need to fight against their nature and get out of the building and talk to customers as much as possible; find out who they are, how they work, what they need and what your startup needs to do, to build and then sell its solution.

In fact, so many engineers, just as myself, spent months of working on a prototype or even a complete solution, sometimes for several years, before actually meeting a first potential customer, and discovering the hard way that all this work has been for nothing.
As hard as it is, Engineers should not work one one single line of code, even not one single powerpoint presentation before having met at least a twenty potential customers or representatives and conducted formal 5.3.1.2 Problem interview.
After that, it's still not a question of writing lines of code, it's a question of investing a few hours - not more ! - in designing a demonstrable prototype for the next set of interviews, the 5.3.1.3 Solution interview. That prototype doesn't need to be actually

working, it should only be demonstrable. A powerpoint presentation with clickable animations works perfectly!

Again, getting out of the building is not getting in the parking lot, it's really about getting in front of the customer.
At the end of the day, it's about *Customer Discovery*. And *Customer Discovery* is not sales, it's a lot of listening, a lot of understanding, not a lot of talking.

A difficulty that people always imagine is that young entrepreneurs with an idea believe that they don't know anybody, so how to figure out who to talk to ?
But at the time of Linkedin, facebook, twitter, it's hard to believe one cannot find a hundred of people to have a conversation with.

And when having a conversation with one of them, whatever else one's asking (5.3.1.2 Problem interview, 5.3.1.3 Solution interview), one should ask two very important final questions:

1. "*Who else should I be talking to ?*"
   And because you're a pushy entrepreneur, when they give you those names, you should ask "*Do you mind if I sit here while you email them introducing me ?*"

2. "*What should I have really asked you ?*"
   And sometimes that gets into another half hour related to what the customer is *really* worried about, what's really the customer's problem.

Customer Discovery becomes really easy once you realize you don't need to get the world's best first interview.
In fact its the sum of these data points over time, it's not one's just going to be doing one and one wants to call on the highest level of the organization.
In fact you actually never want to call on the highest level of the organization because you're not selling yet, you don't know enough.
What one actually wants is to understand enough about the customers, their problems and how they're solving it today, and whether one's solution is something they would want to consider.

A few hints in regards to how to get out of the building:

5.3.1.2 Problem interview

Problem Interview is Ash Maurya's term for the interview you conduct to validate whether or not you have a real problem that your target audience has.

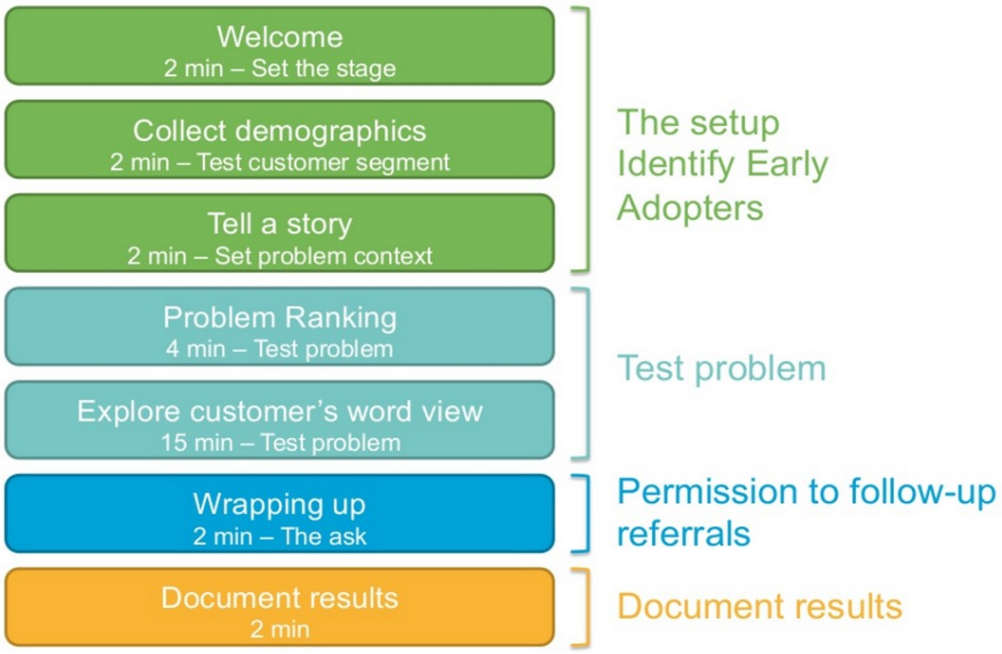
In the Problem Interview, you want to find out 3 things:

1. **Problem** - What are you solving? - How do customers rank the top 3 problems?
2. **Existing Alternatives** - Who is your competition? - How do customers solve these problems today?
3. **Customer Segments** - Who has the pain? - Is this a viable customer segment?

Talking to people is hard, and talking to people in person is even harder. The best way to do this is building a script and sticking to it. Also don't tweak your script until you've done enough interviews so that your responses are consistent.
The main point is to collect the information that you will need to validate your

problem, and to do it face-to-face, either in-person or by video call. It's actually important to see people and be able to study their body language as well.

The interview script - at least the initial you should follow until you have enough experience to build yours - is as follows:



If you have to remember just three rules for problem interviews here they are:

1. Do not talk about your business idea or product. You are here to understand a problem, not imagine or sell a solution yet.
2. Ask about past events and behaviours
3. No leading question, learn from the customer

After every interview, take a leap backwards, analyze the answers, make sure you understand everything correctly and synthesize the results.
After a few dozen of interviews, you should be a able to make yourself a clear understanding of the problem and initiate a few ideas regarding the solution to it. Finding and validating your solution brings us to the next topic: the *Solution Interview*.

And what if a customer tells you that the issues you thought are important really aren't? Learn that you have gained important data.

5.3.1.3 Solution interview

In the Solution Interview, you want to find out three things:

1. **Early Adopters** - Who has this problem? - How do we identify an early adopter?
2. **Solution** - How will you solve the problems? - What features do you need to build?
3. **Pricing/Revenue** - What is the pricing model? - Will customers pay for it?

The key point here is to understand how to come up with a solution fitting the problem, step by step getting to the right track with your prototype and also understanding what could be a pricing model.

A *demo* is actually important. Many products are too hard to understand without some kind of demo. If a picture is worth a thousand words, a demonstration is probably worth a million.

Identifying early adopters is also key.
Think of something: if one of the guys you meet tells you that you definitely hold something, ask him if he would want to buy it. If he says he would definitely buy it when it's ready and available, ask him if he would commit to this. If he says he commits to this, ask him if he would be ready to pay half of it now and have it when its ready, thus becoming a partner or an investor.
If you find ten persons committing on already paying for the solution you draw, you may not even need to search for investors, you already have them. And that is the very best proof you can find that your solution is actually something.
And customers or partners are actually the best possible type of investors.

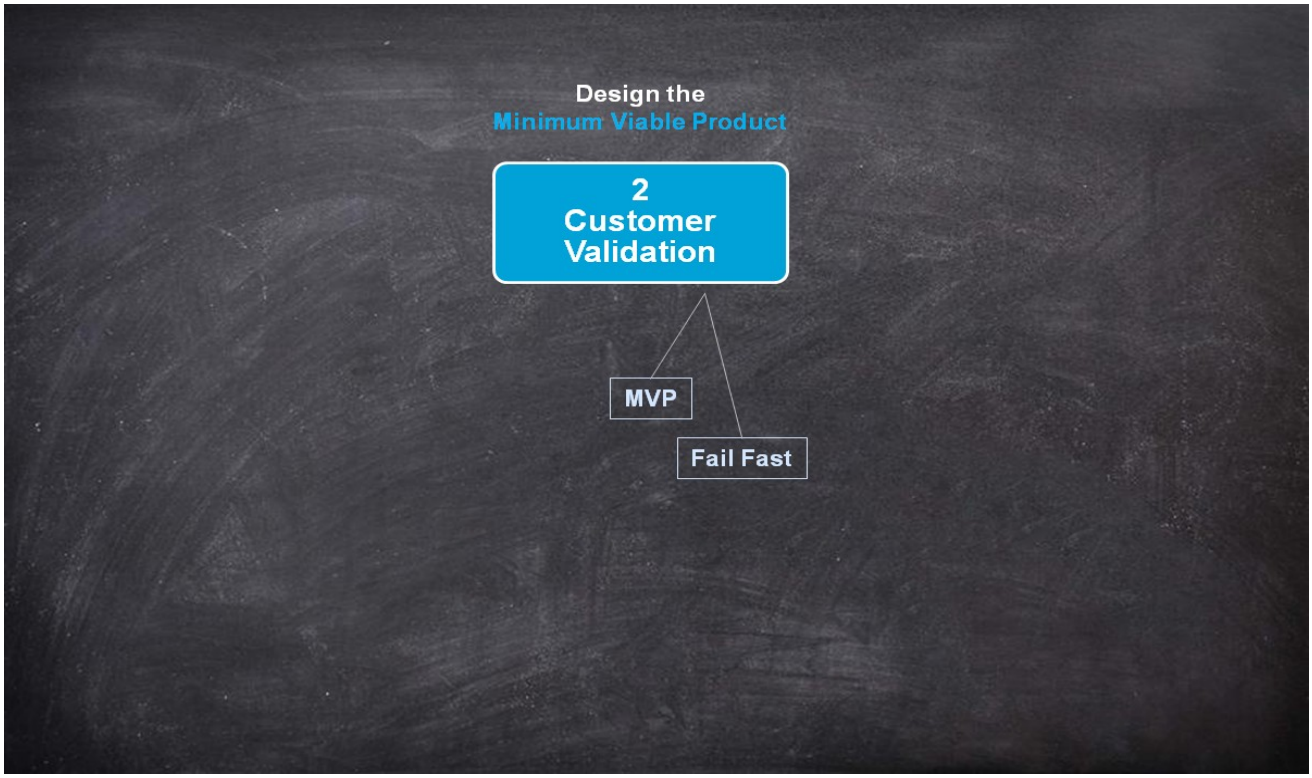
5.3.2 Customer Validation

The second step of the Customer Development model, *Customer Validation*, focuses on developing a sales model that can be replicated. The sales model is validated by running experiments to test if customers value how the startup's products and services are responding to the customer problems and needs identified during the previous step.
If customers show no interest, then the startup can 5.3.3.2 Pivot to search for a better business model.

Customer Validation needs to happen to validate if the customers really care about the products and services that could be valuable to them. This second step is hence really about *Product-Market Fit* which occurs when there is a sales model that works, when customers think the proposed solution is valuable to them. This should be proven by evidence that customers care about the products and services that conform the value proposition.

Blank believes that *product-market fit* needs to happen before moving from Customer Validation to Customer Creation (or the *Search Phase* to the *Execution Phase*).

The two practices I want to emphasize at this stage are as follows:



5.3.2.1 MVP

The **Minimum Viable Product** is an engineering product with just the set of features required to gather *validated learnings* about it - or some of its features - and its continuous development.
This notion of *Minimum Feature Set* is key in the MVP approach.

The key idea is that it makes really no sense developing a full and finalized product without actually knowing what will be the market reception and if all of it is actually worth the development costs.
Gathering insights and directions from an MVP avoids investing too much in a product based on wrong assumptions. Even further, The *Lean Startup* methodology seeks to avoid assumptions at all costs, see 5.1.4 The Feedback Loop and 5.3.3.1 Metrics Obsession.

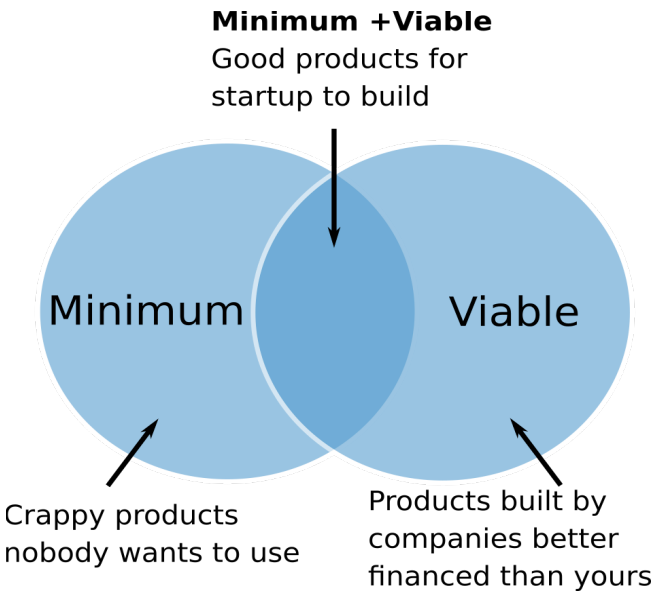
The *Minimum Viable Product* should have just that set of initial features strictly required to have a valid product, usable for its very initial intent, and nothing more. In addition these features should be as minimalist as possible but without compromising the overall *User Experience*. A car should move, a balloon should be round and bounce, etc.
when adopting an MVP approach, the MVP is typically put at disposal at first only to *early adopters*, these customers that may be somewhat forgiving for the "naked" aspect of the product and more importantly that would be willing to give feedback and help steer the product development further.

Eric Ries defines the MVP as:

"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."

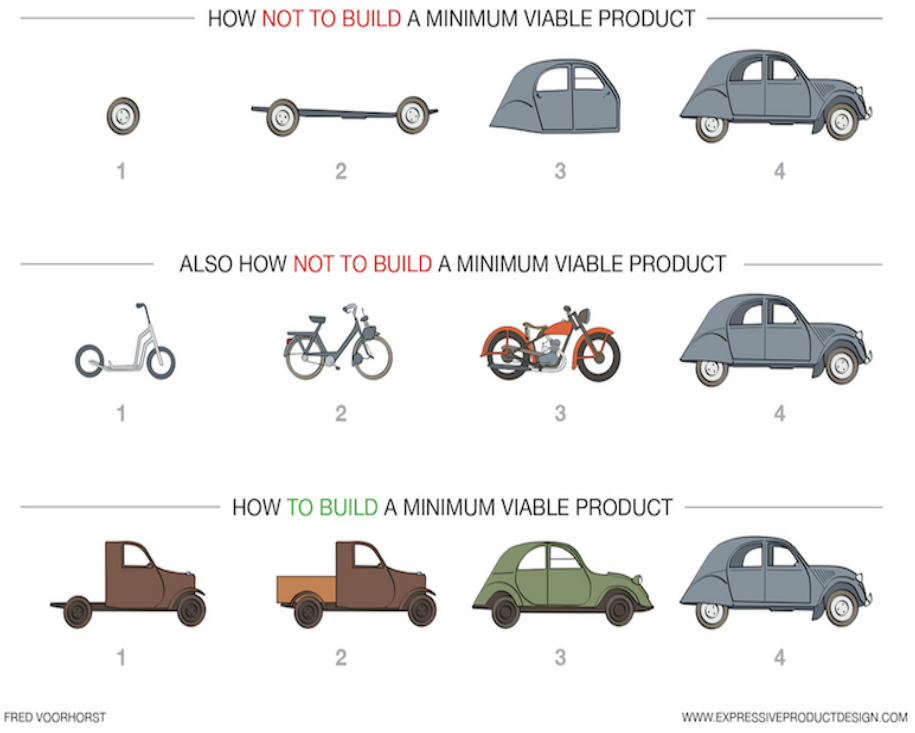
The definition's use of the words *maximum* and *minimum* means it is really not formulaic. In practice, it requires a lot of judgment and experience to figure out, for any given context, what MVP makes sense.

The following chart is pretty helpful in understanding why both terms *minimum* and *viable* are equally important and why designing an MVP is actually difficult:



When applied to a new feature of any existing product instead of a brand new product, the MVP approach is in my opinion somewhat different. It consists of implementing the feature itself not completely; rather, a mock-up or even some animation simulating the new feature should be provided.
The mock-up or links should be properly instrumented so that all user reactions are recorded and measured in order to get insights on the actual demand of the feature and the best form it should take (5.3.3.1 Metrics Obsession),
This is called a **deploy first, code later** method.

Fred Voorhorst' work does a pretty good job in explaining what an MVP is:



(Fred Voorhorst - Expressive Product Design - http://www.expressiveproductdesign.com/minimal-viable-product-mvp/)

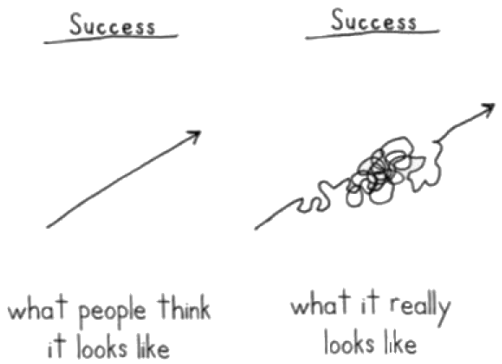
Developing an MVP is most definitely not the same as developing a sequence of elements which maybe, eventually combine into a product. A single wheel is not of much interest to a user wanting a personal transporter like a car, as illustrated by the first line.
Instead, developing an MVP is about developing the vision. This is not the same as developing a sequence of intermediate visions, especially not, if these are valuable products by themselves. As an example, a skateboard will likely neither interest someone in search for a car, as illustrated by the second line.

Developing an MVP means developing a sequence of prototypes through which you explore what is key for your product idea and what can be omitted.

5.3.2.2 Fail Fast

The key point of the "**fail fast**" principle is to quickly abandon ideas that aren't working. And the big difficulty of course is not giving up too soon on an idea that could potentially be working. should one find the right channel, the right approach. Fail fast means getting out of planning mode and into testing mode, eventually for every component, every single feature, every idea around your product or model of change. *Customer development* is the process that embodies this principle and helps you determine which hypotheses to start with and which are the most critical for your new idea.

It really is OK to fail if one knows the reason of the failure, and that is where most people go wrong. Once a site or a product fails then one needs to analyze why it bombed. It's only then that one can learn from it.
The key aspect here is really learning. And learning comes from experimenting, **trying things, measuring their success and adapting**.
An entrepreneur should really be a pathologist investigating a death and finding the cause of the failure. Understanding the cause of a failure can only work if the appropriate measures and metrics around the experiment are in place.



Now failing is OK as long as we learn from it and as long as we **fail as fast as possible**. Again, the whole *lean* idea is to avoid waste as much as possible and there's no greater waste than keeping investing on something that can ultimately not work. Failing as fast as possible, adapting the product, pivoting the startup towards its next approach as soon as possible is key.
But then again, the big difficulty is not to give up too soon on something that could possible work.

Fail fast,
Learn faster,
Succeed sooner !

So how do you know when to turn, when to drop an approach and adapt your solution ? How can you know it's not too soon?

Measure, measure, measure of course!

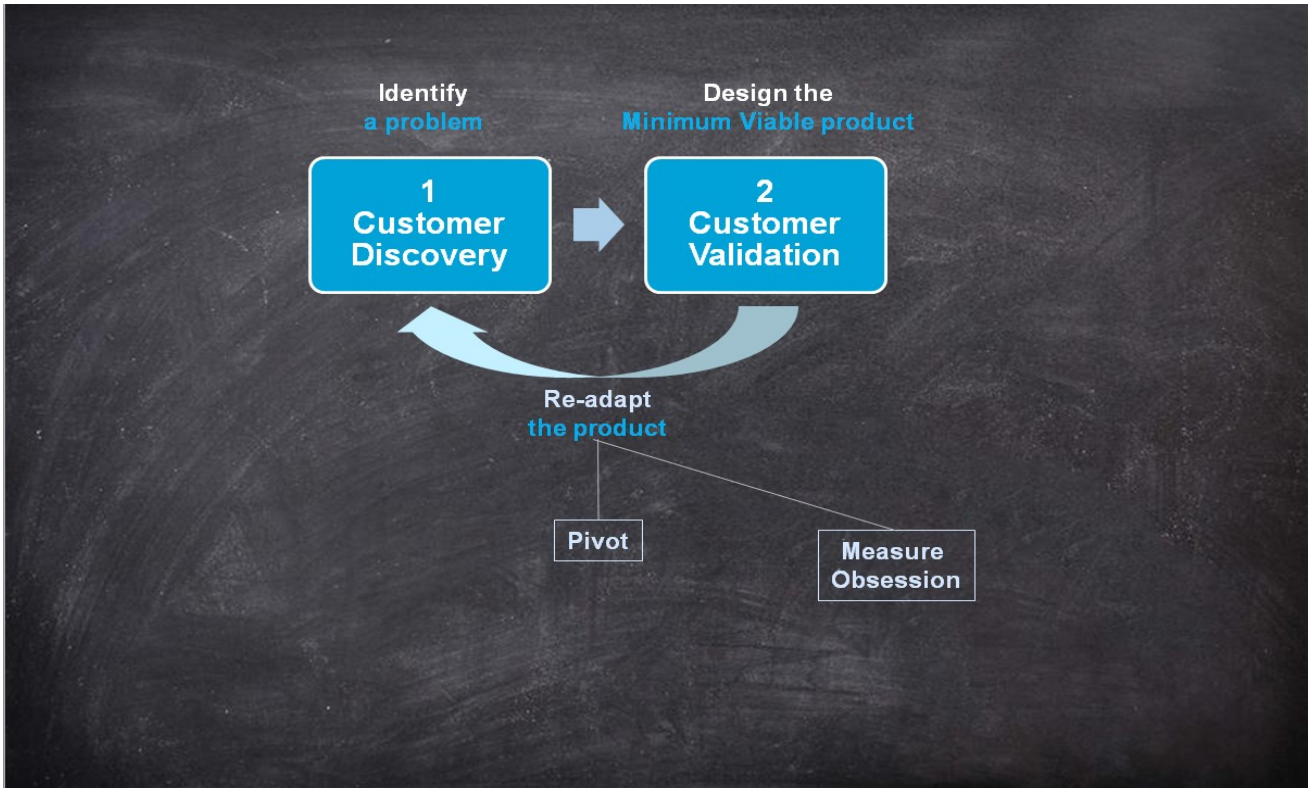
The testing of new concepts, failing, and building on failures are necessary when creating a great product.
The adage, "*If you can't measure it, you can't manage it*" is often used in management and is very important in *The Lean Startup* approach. By analyzing data, results can be measured, key lessons learned, and better initiatives employed.

5.3.3 Re-adapt the product

Customer development isn't predictable; you don't know what you're going to learn until you start. You'll need the ability to think on your feet and adapt as you uncover new information.
Adapting, in my opinion, is really re-adapting the product to the new situation, to the new knowledge you gained from the previous steps. And re-adapting the product, your solution, your approach is pivoting.

But I want to emphasize here that pivoting, or re-adapting the product, should only happen with the right data, the precise insights that give a clear new direction. Metrics and insight are essential.

The key practices here are as follows:



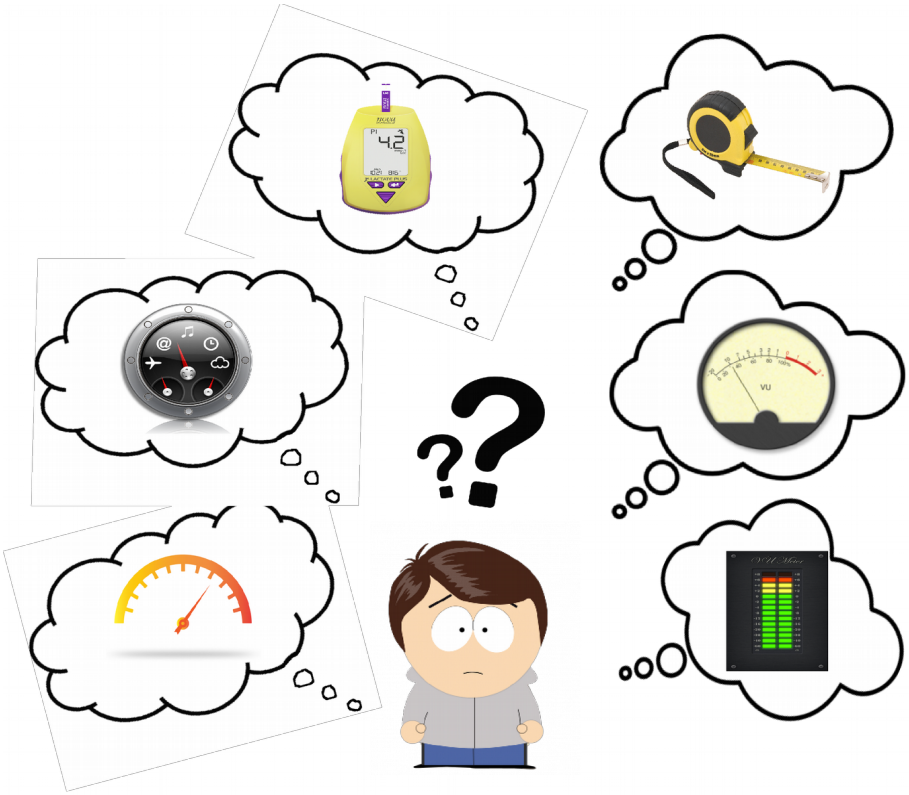
5.3.3.1 Metrics Obsession

In the *build-measure-learn* loop, there is measure ... *The Lean Startup* makes from measuring everything an actual obsession. And I believe that this is a damn' good thing.
Think of it: what if you have an idea regarding a new feature or an evolution of your product and you don't already have the metrics that can help you take a sound and enlightened decision? You'll need to introduce the new measure and wait until you get the data. Waiting is not good for startups.

This is why I like thinking of it as a **Metrics Obsession**. Measure everything, everything you can think of!

And repeat a hundred times:

I will never ever again think that
Instead I will *measure* that ...



Or as Edward Deming said :

"In god we trust, all others must bring data"

Imagine you work on a website. You should enhance your backend to measure, at least: amount of times a page has been displayed, count of users and different users displaying the pages, amount of times a link or button has been clicked, by who it has been clicked, how much time after the containing page has been displayed, what is the user think time between 2 actions, what is the path of navigation from each and every user (actually build the graph and the counts along the branches), etc.
Measure everything! Don't hesitate to measure something you do not see any use for now. Sooner or later you will find a usage for that metrics, and that day, you better have it.

How to choose good metrics ?

Honestly there is no magic silver bullet and it can in fact be pretty difficult to pick up the right metric that would be most helpful to validate a certain hypothesis.
However, metrics should at all cost respect the three A's. Good metrics

- are **actionable**,
- can be **audited**
- are **accessible**

An **actionable metric** is one that ties specific and repeatable actions to observed results. The *actionable* property of picked up metrics is important since it prevents the entrepreneur from distorting the reality to his own vision. We speak of *Actionable vs. Vanity* Metrics.
Meaningless metrics such as "How many visitors ?", "How many followers ?" are vanity metrics and are useless.

Ultimately, your metrics should be useful to **measure progress against your own questions**.

5.3.3.2 Pivot

In the process of learning by iterations, a startup can discover through field returns with real customers that its product is either not adapted to the identified need, that it does not meet that need.
However, during this learning process, the startup may have identified another need (often related to the first product) or another way to answer the original need.
When the startup changes its product to meet either this new need or the former need in a different way, it is said to have performed a **Pivot**.
A startup can *pivot* several times during its existence.

A *pivot* is ultimately a **change in strategy** without *a change in vision*.
It is defined as a structured course correction designed **to test a new fundamental hypothesis** about the product, business model and engine of growth.

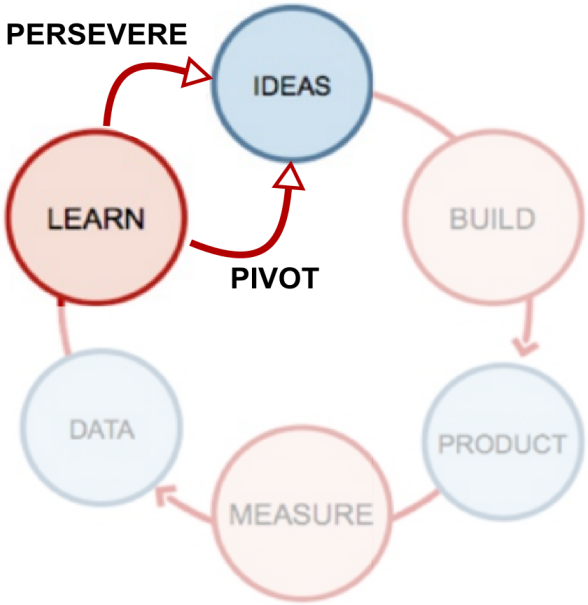
The vision is important. A startup is created because the founder has a vision and the startup is really built and organized around this vision. If the feedback from the field compromises the vision, the startup doesn't need to pivot, it needs to resign, cease its activities and another startup, another organization aligned to the new vision should perhaps be created.

There are various kind of pivots:

- **Zoom-In :** a single feature becomes the whole product
- **Zoom-Out :** the whole initial product becomes a feature of a new product
- **Customer segment :** Good product, bad customer segment
- **Customer need :** Repositioning, designing a completely new product (still sticking to the vision)
- **Platform :** Change from an application to a platform, or vice versa
- Many others ...

Pivot or Persevere

Since entrepreneurs are typically emotionally attached to their product ideas, there is a tendency to hang in there too long. This wastes time and money. The pivot or persevere process forces a non-emotional review of the hypothesis.



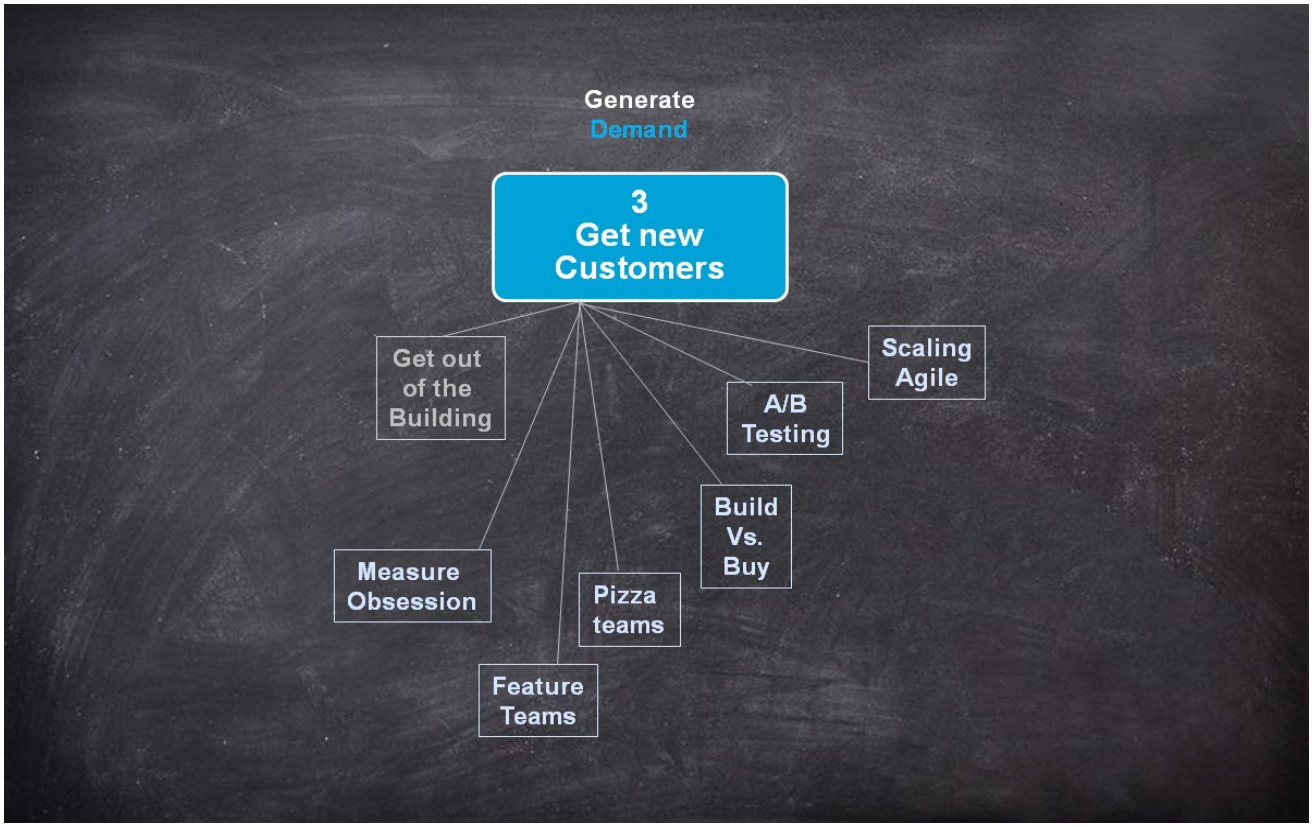
Unsurprisingly, knowing when to pivot is an art, not a science. It requires to be well thought through and can be pretty complicated to manage.
At the end of the day, knowing when to pivot or persevere requires experience and, more importantly, metrics: proper performance indicators giving the entrepreneur clear insights about the market reception of the product and the fitting of customer needs.

One thing seems pretty clear though, if it becomes clear to everyone in the company that another approach would better suit the customer needs, the startup needs to pivot, and fast.

5.3.4 Get new customers

The third step, the Customer Creation step, to "*start building end user demand to scale the business*", is the precursor to achieve *Business Model Fit*. Therefore, the Business Model Fit stage can be understood as validating the value for the company, where as the product-market fit focuses on validating the value for the customer.

The set of practices I deem important here are as follows:



Again, attaching some of these practices here or in the next and last step can be subjective. In my opinion, the startup needs to embrace this Lean and Agile principles and practices before it attempts to scale its organization, hence the reason why I consider these practices at this stage.

5.3.4.1 Pizza Teams

Jeff Bezos, Amazon's founder and CEO, always said that a team size shouldn't be larger than what two pizzas can feed, two american pizzas, not italian, needless to say.
This makes it 7 +/- 2 co-workers inside an Agile Team.

More communication isn't necessarily the solution to communication problems - it's how it is carried out. Compare the interactions at a small dinner - or pizza - party with a larger gathering like a wedding. As group size grows, you simply can't have as meaningful of a conversation with every person, which is why people start clumping off into smaller clusters to chat.
For Bezos, small teams make it easier to communicate more effectively rather than more, to stay decentralized and moving fast, and encourage high autonomy and innovation. Here's the science behind why the two-pizza team rule works.

As team size grows, **the amount of one-on-one communication channels tend to explode**, following the formula to compute number of links between people which is $n (n - 1) / 2$.
This is $O(n_2)$ (Hello Engineers) and is really a *combinatorial explosion*.
If you take a basic two-pizza team size of, say, 6. That's 15 links between everyone. Double that group for a team of 12. That shoots up to 66 links.
The cost of coordinating, communicating, and relating with each other explodes to such a degree that it lowers individual and team productivity.

Under five co-workers, the team becomes fragile to external events and lacks creativity.
Beyond ten, communication loses efficiency, cohesion diminishes, parasitism behaviors and power struggles appear, and the performance of the team decreases very rapidly with the number of members.

The right size for an Agile Team is 7 +/- 2 persons.

5.3.4.2 Feature Teams

Let's first have a look at what is the other model: *Component Teams*.

Component Teams

Components Teams are the usual, the legacy model. In large IT organizations, there is usually a development team dedicated to the front-end, the Graphical User Interface, another team dedicated to developing the Java (Or Cobol :-) backend, a team responsible to design and maintain the database, etc.
A Component Team is defined as a development Team whose primary area of concern is restricted to a specific component, or a set of components from a specific layer or tiers, of the system.
Prior to Agile, most large-scale systems were developed following the component team approach and the development teams were organized around components and subsystems.

The most essential drawback of *Component Teams* is obvious : most new features are spread among several components, creating dependencies that require cooperation between these teams. This is a continuing drag on velocity, as the

individual teams spend much of their time discussing dependencies between teams and testing, assessing, fixing behaviour across components rather than delivering end user value as efficiently as possible.
An important direct consequence of this dependency is that any given feature can only be delivered as fast as can be delivered the component changes by the slowest (or most overloaded) component team.

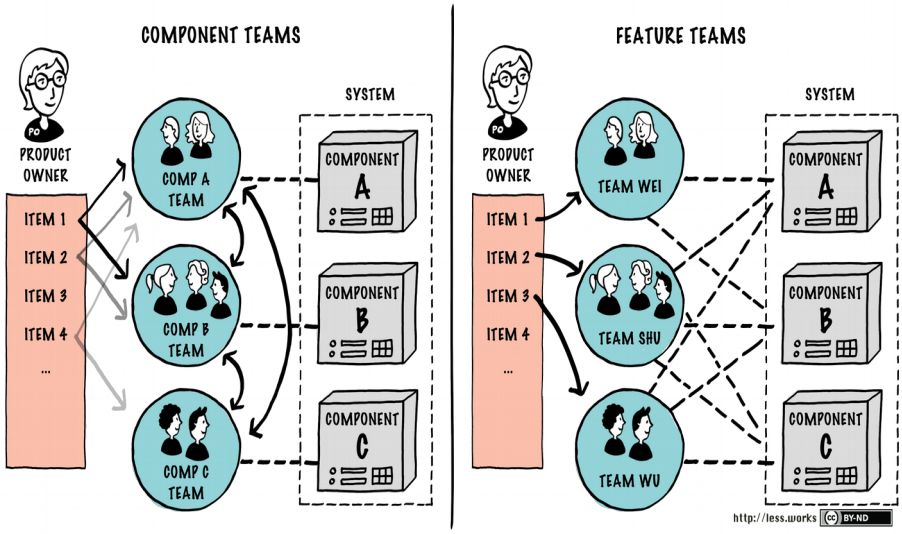
Feature Teams

As such, in an Agile Organization, where the whole company is organized around Feature backlogs or Kanban, it makes a lot more sense to organize the various development teams in **Feature Teams**.
Feature teams are organized around user-centered functionality. Each and every team, is capable of delivering end-to-end user value throughout the software stack. Feature teams operate primarily with user stories, refactors and spikes. However, technical stories may also occasionally occur in their backlog.
A feature team is defined as a long-lived, cross-functional, cross-component team that completes many end-to-end customer features, one by one.

More Information on Feature Teams:

- From SAFe - Scaled Agile Framework
- From LeSS - Large Scale Scrum framework

The difference between both models is well illustrated this way:



(Source : https://less.works/less/structure/feature-teams.html)

A pretty good summary of the most essential differences between both models is available on the LeSS web site:

component team	feature team
optimized for delivering the **maximum number of lines of code**	optimized for delivering the **maximum customer value**
focus on increased individual productivity by implementing 'easy' lower-value features	focus on high-value features and system productivity (value throughput)
responsible for only part of a customer-centric feature	responsible for complete customer-centric feature
traditional way of organizing teams - follows Conway's law	'modern' way of organizing teams - avoids Conway's law
leads to 'invented' work and a forever-growing organization	leads to customer focus, visibility, and smaller organizations
dependencies between teams leads to additional planning	**minimizes dependencies between teams to increase flexibility**
focus on single specialization	focus on multiple specializations
individual/team code ownership	**shared product code ownership**
clear individual responsibilities	**shared team responsibilities**
results in 'waterfall' development	**supports iterative development**
exploits existing expertise; lower level of learning new skills	exploits flexibility; continuous and broad learning
works with sloppy engineering practices-effects are localized	requires skilled engineering practices-effects are broadly visible
contrary to belief, often leads to low-quality code in component	**provides a motivation to make code easy to maintain and test**
seemingly easy to implement	seemingly difficult to implement

(Source : https://less.works/less/structure/feature-teams.html)

The Analogy with a Star Trek team makes surprisingly and funnily a lot of sense.

Think of a Star Trek spaceship. The crew is constituted by Commanding Officers, Medical Officers, Medical Staff, Engineering Officers, Engineering Staff, Science Officers, Scientists, etc.
These different functions, competencies and responsibilities are grouped together to work towards a common objective, its continuing mission: *to explore strange new worlds, to seek out new life and new civilizations, to boldly go where no one has gone before*.

Now imagine if Starfleet had instead put all the Commanding Officers in one ship, all medical staff in another ship, and so on. It would have been pretty difficult to make those ships actually do anything significant, don't you think ?
This is precisely the situation of *Component Teams*.
Just as with a Star Trek Ship, it makes a lot more sense to put all the required competencies together in a team (or ship) and assign them a clear objective, implementing that feature throughout the technology and software stack.

5.3.4.3 Build vs. Buy

This dilemma is as old as the world of computers: is it better to invest in developing a software that is best suited to your needs or should you rely on a software package or third party product that embed the capitalization and R&D of **another** software editor in order to - **apparently** - speed up your time to market ?

In order to be as efficient as possible on the build-measure-learn loop, it is essential to master your development process. For this reason, *tailor made* solutions are better because the adoption of a third party software package often requires to

invest a lot of resources not in the development of your product, but instead in the development of workarounds, hacks and patches to correct all the points on which the software package is poorly adapted to the specific and precise behavior required by your own product feature.

In the case of a startup, this aspect is catastrophic. Investing in the development of hacks and workarounds around a third party product, a product that one has in addition to pay for, sometimes depending on the number of machine or users, instead of developing the startup's core business, should just not happen.

This cost aspect is particularly critical of course when scaling the solution. When one multiplies the processors and the servers, the invoice climbs very quickly and not necessarily linearly, and the costs become very visible, no matter whether it is a business software package or an infrastructure brick.

This is precisely one of the arguments that led LinkedIn to gradually replace Oracle with a home solution: Voldemor.

Most technologies that make the buzz today in the world of high performance architectures are the result of developments made by the Web Giants that have been released as Open Source: Cassandra, developed by Facebook, Hadoop and HBase inspired by Google and developed at Yahoo, Voldemort by LinkedIn, etc.

Open-Source software is cool

Of course the cost problem doesn't apply to Open-Source and free to use software. In addition, instead of developing workarounds and patches around Open-Source Software, you can instead change its source, fork it and maintain your different baseline while still benefiting from the developments made on the official baseline by merging it frequently.

At the end of the day, integrating an Open-Source software, in contrary to Editor / Closed Source Software, is pretty closed to developing it on your own, as long as you have the competencies to maintain it on your own should you need to.
Open-Source software is cool, go for it!

5.3.4.4 A/B Testing

A/B testing is a marketing technique that consists in proposing several variants of the same object that differ according to a single criterion (for example, the color of a package) in order to determine the version which lead to the best appreciation and acceptance from consumers.
A / B testing is used to qualify all kinds of multivariate tests.

An A/B test evaluates the respective performance of one or more partially or totally different versions of the same product or functionality by comparing them to the original version. The test consists in creating modified versions of the functionality

by modifying as many elements as desired.
The idea is to split the visitors into two groups (hence the name A / B) and to present to each group a different version of the functionality or the product. Then, we should follow the path of the two groups, their appreciation of the functionality by means of ad'hoc metrics, and we consider which of the two variants gives the best result with respect to a given objective.

For instance, in order to tests if a *trial first approach* is more appealing and leads eventually to more sales than a mandatory buying:

The A/B test enables to validate very quickly the idea of introducing a trial period for a feature or a product.

5.3.4.5 Scaling Agile

Transforming a startup into a company, changing and scaling its organization is a unique, and yet challenging, opportunity to make it an agile organization keeping the *lean* genes on which it has been built.
The *agile* aspect here is essential and the approach here actually has a name: **Scaling Agile**.

Scrum and *Kanban* are two agile frameworks often used at the team level. Over the past decade, as they gained popularity, the industry has begun to adapt and use Agile in larger companies. Two methods (among others) emerged to facilitate this process: **LeSS** (Large Scale Scrum) and **SAFe** (Scaled Agile Framework). Both are excellent starting points for using Agile on a large scale within a company.

Both approaches differ a little but also have a lot in common: they consist of scaling agility first among multiple agile team within the R&D or Engineering department and then around it, by having the whole company organizing its activities in an agile

way and centered on the engineering team, the product development team.
I won't be describing these both approaches any further here and I let the reader refer to both links above.

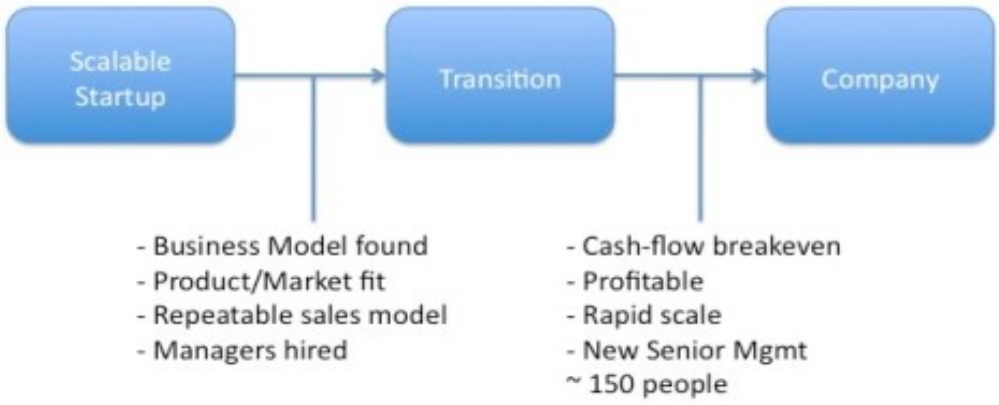
I just want to emphasize how important I believe that is. Scaling Agile is key in aligning business and IT engagement models.

5.3.5 Company creation

Company creation is the end phase, when all assumptions have been confirmed or adapted, when the product is build in an acceptable form, when the break-even point it reached, and the startup should evolve to a corporation. When that moment is reached, startups must begin the transition from the temporary organization designed to search a business model to a structure focused on executing a validated model.

Company creation happens at the moment the company can transition from its informal, learning and discovery-oriented Customer Development team (startup, temporary organization) into formal departments with VPs of Sales, Marketing and Business Development.
At that moment, these executives should focus on building mission-oriented departments that can exploit the company's early market success.

This is a change of bracket. We think of *Company Creation* since it is really a question of creating a company, from what was "only" a startup. The temporary organization should evolve towards a sustainable and viable organization.



Describing anything further in regards to *Company Creation* exceeds the scope of this article focused on *Lean Startup Practices*.
I can only recommend reading Steve Blank's article on the subject (or the big chapter in the "*Four Steps to the Epihpany*"):

- A Startup is Not a Smaller Version of a Large Company

- The Elves Leave Middle Earth - Sodas Are No Longer Free
- The Peter Pan Syndrome - The Startup to Company Transition
- What Do I Do Now? The Startup Lifecycle

5.4. Conclusions

The Lean Startup is not dogmatic. It is first and foremost a question of being aware that the market and the customer are not in the architecture meetings, marketing plans, sales projections or key feature discussions.

Bearing this in mind, you will see assumptions everywhere. The key approach then consists in putting in place a discipline of validation of the hypotheses while keeping as key principle to validate the minimum of functionalities at any given time.

Before doing any line of code, the main questions to ask revolve around the triplet : *Client / Problem / Solution*
Do you really have a problem that is worth resolving? Is your solution the right one for your customer? Is he likely to buy it? For how much ? All the means are good to remove these hypotheses: interviews, market studies, models, whatever you can think of.

The next step is to know if the model that you came up with and have been able to test on a smaller scale is really repeatable and extensible.
How to put a product they have never heard of in the hands of the customers ? Will they understand it as well with its use and benefits ?

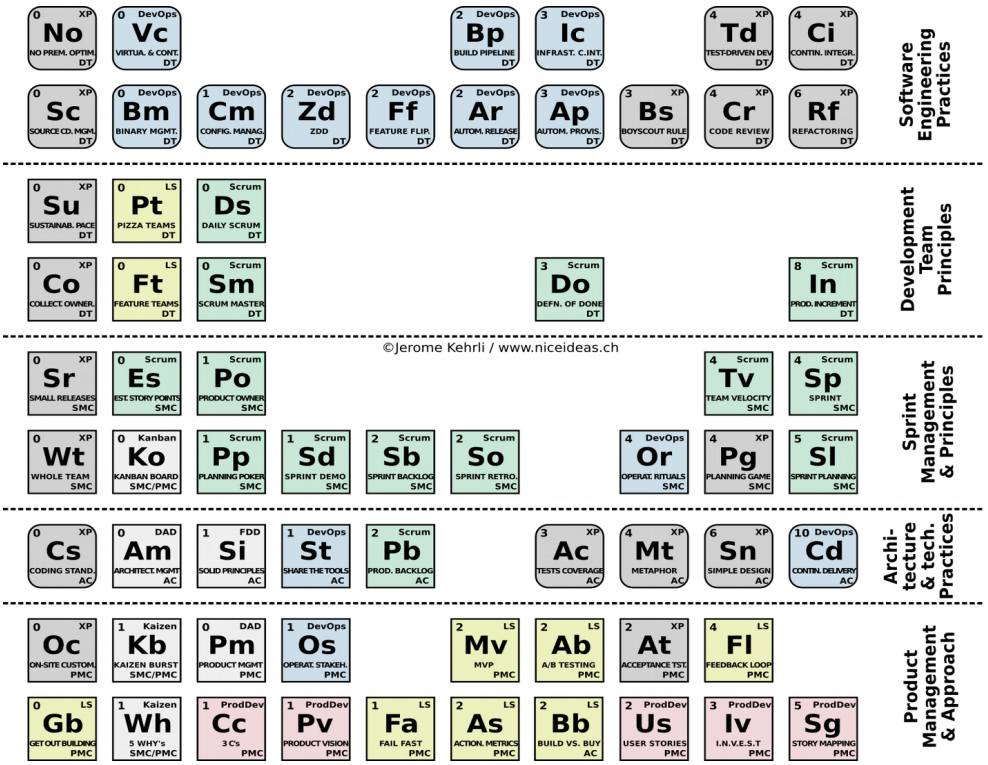
The Lean Startup is not an approach to be reserved only to mainstream websites or fancy internet products. Innovating by validating hypotheses as quickly as possible and limiting financial investment is obviously a logic that can be transposed to any type of engineering project, even if it is internal.
I am convinced that the practices and principles from **the Lean Startup** approach should be more widely used to avoid so many projects burning so much money and effort before being simply dropped.

6. Periodic Table of Agile Principles and Practices

After writing my previous article, I wondered how I could represent on a single schematic all the *Agile Principles and Practices* from the methods I am following, XP, Scrum, Lean Startup, DevOps and others.
I found that the approach I used in in a former schematic - a graph of relationship between practices - is not optimal. It already looks ugly with only a few practices and using the same approach for the whole set of them would make it nothing but a mess.

So I had to come up with something else, something better.
Recently I fell by chance on the Periodic Table of the Elements... Long time no see...
Remembering my physics lessons in University, I always loved that table. I remembered spending hours understanding the layout and admiring the beauty of its natural simplicity.
So I had the idea of trying the same layout, not the same approach since both are not comparable, really only the same layout for *Agile Principles and Practices*.
The result is hereunder.

6.1 The Periodic Table of Agile Principles and Practices

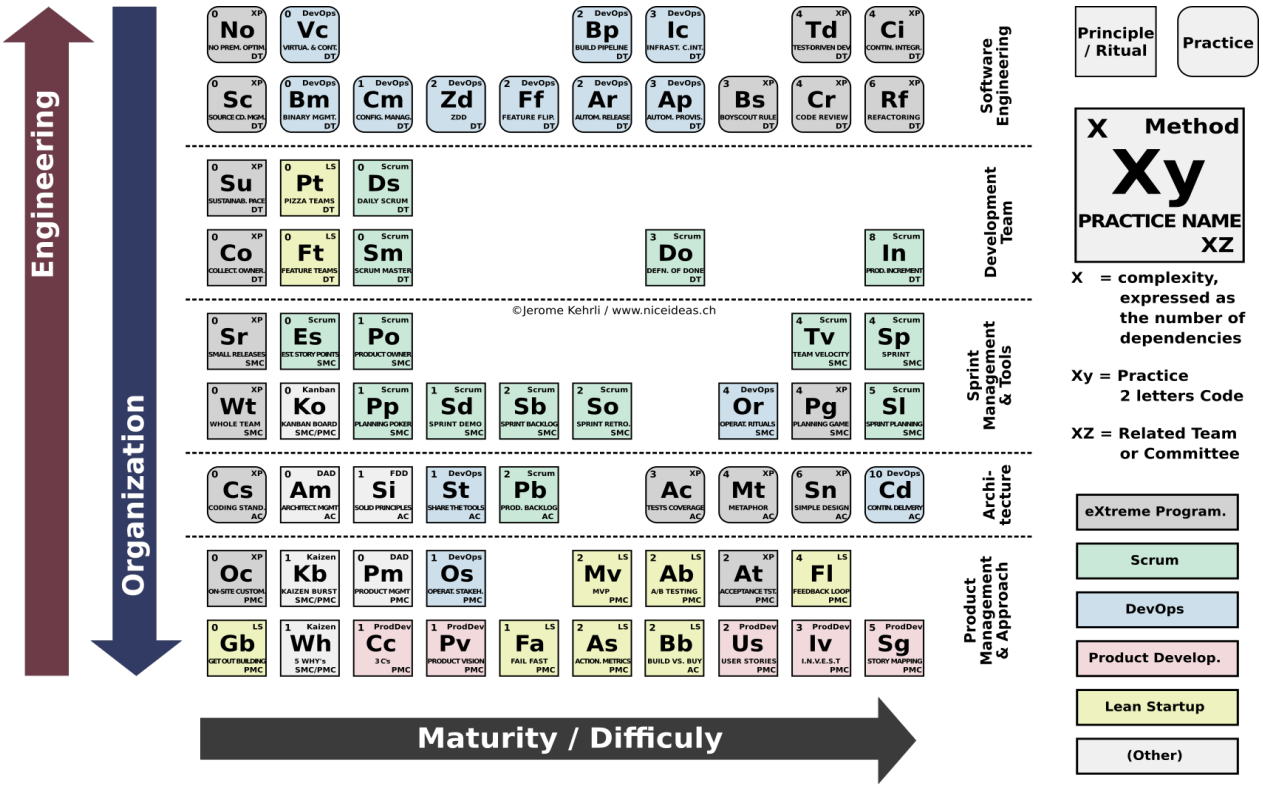


The layout principle is and the description of the principles and practices is explained hereafter.

6.2. Layout Principle

- The **origin Method** such as XP, Scrum, DevOps, etc is indicated by the color as well as the name of the method on the top-right corner.
- The **category**, *Principle* or *Practice* is indicated by the shape: rectangle or round corners.
- The number represents the **complexity** expressed as the number of dependencies.
- The **team or committee** concerned with the principle or practice is indicated as note on the bottom-right corner.
- The **horizontal dimension** is related to the complexity. The more on the right is an element, the higher its complexity.
- The **vertical dimension** is related to classifying principles and practices more organization or more related to engineering, in specific layers related to the category or team they apply to.

This is best presented as follows:



6.3. Remarks

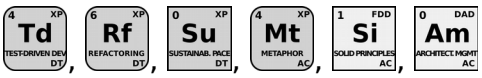
- Interestingly, but not surprisingly, scrum is really in the middle of the schematic, underlying the fact that it impacts as well development principles and the development team organization.
- XP is really everywhere down the line.
- Product Development is really about Product Management in the Agile world.
- DevOps is more related to development practices than everything else.

The next part of this article describes each and every principle and practice.

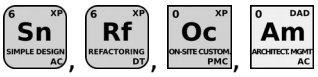
6.4. Principles and Practices

6.4.1 XP

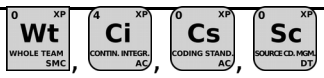
Sn : Simple Design
A simple design always takes less time to finish than a complex one. So always do the simplest thing that could possibly work next. If you find something that is complex replace it with something simple. It's always faster and cheaper to replace complex code now, before you waste a lot more time on it.

Depends on 

Mt : Metaphor
System Metaphor is itself a metaphor for a simple design with certain qualities. The most important quality is being able to explain the system design to new people without resorting to dumping huge documents on them. A design should have a structure that helps new people begin contributing quickly. The second quality is a design that makes naming classes and methods consistent.

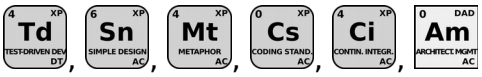
Depends on 

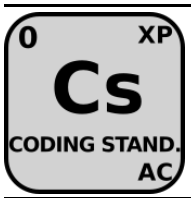
Td : TDD = Test Driven Development
Test-driven development is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements.

Depends on 

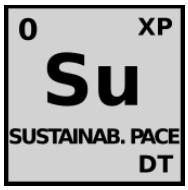
Oc : Onsite Customer
One of the few requirements of extreme programming (XP) is to have the customer available. Not only to help the development team, but to be a part of it as well. All phases of an XP project require communication with the customer, preferably face to face, on site. It's best to simply assign one or more customers to the development team.

Rf : Refactoring
We computer programmers hold onto our software designs long after they have become unwieldy. We continue to use and reuse code that is no longer maintainable because it still works in some way and we are afraid to modify it. But is it really cost effective to do so? Extreme Programming (XP) takes the stance that it is not. When we remove redundancy, eliminate unused functionality, and rejuvenate obsolete designs we are refactoring. Refactoring throughout the entire project life cycle saves time and increases quality.
Refactor mercilessly to keep the design simple as you go and to avoid needless clutter and complexity. Keep your code clean and concise so it is easier to understand, modify, and extend

Depends on 



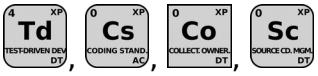
Cs : Coding Standards
Code must be formatted to agreed coding standards. Coding standards keep the code consistent and easy for the entire team to read and refactor. Code that looks the same encourages collective ownership.



Su : Sustainable Pace
To set your pace you need to take your iteration ends seriously. You want the most completed, tested, integrated, production ready software you can get each iteration. Incomplete or buggy software represents an unknown amount of future effort, so you can't measure it. If it looks like you will not be able to get everything finished by iteration end have an iteration planning meeting and re-scope the iteration to maximize your project velocity. Even if there is only one day left in the iteration it is better to get the entire team re-focused on a single completed task than many incomplete ones.

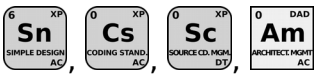
Wt : Whole Team
All the contributors to an XP project sit together, members of a whole team. The team shares the project goals and the responsibility for achieving them. This team must include a business representative, the "Customer" who provides the requirements, sets the priorities, and steers the project

Ci : Continuous Integration
Developers should be integrating and commiting code into the code repository every few hours, when ever possible. In any case never hold onto changes for more than a day. Continuous integration often avoids diverging or fragmented development efforts, where developers are not communicating with each other about what can be re-used, or what could be shared. Everyone needs to work with the latest version. Changes should not be made to obsolete code causing integration headaches.

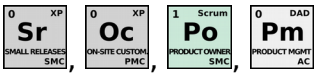
Depends on 

Co : Collective Ownership
Collective Ownership encourages everyone to contribute new ideas to all segments of the project. Any developer can change any line of code to add functionality, fix bugs, improve designs or refactor. No one person becomes a bottle neck for changes.

Cr : Code Review
Code review is increasingly favored over strict Pair Programming as initially requires by the XP Method. The problem with Pair programming is that it cannot fitr everybody.
Code reviews are considered important by many large-process gurus. They are intended to ensure conformance to standards, and more importantly, intended to ensure that the code is clear, efficient, works, and has QWAN. They also intended to help disseminate knowledge about the code to the rest of the team.

Depends on 

Pg : Planning Game
The main planning process within extreme programming is called the Planning Game. The game is a meeting that occurs once per iteration, typically once a week. The planning process is divided into two parts: Release Planning and Sprint Planning.

Depends on 

Sr : Small Releases
The development team needs to release iterative versions of the system to the customers often. Some teams deploy new software into production every day. At the very least you will want to get new software into production every week or two. At the end of every iteration you will have tested, working, production ready software to demonstrate to your customers. The decision to put it into production is theirs.

Sc : Source Code Management
A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

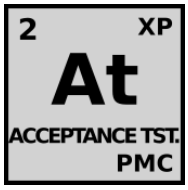
Bs : Boyscout Rule
The Boy Scouts have a rule: "Always leave the campground cleaner than you found it." If you find a mess on the ground, you clean it up regardless of who might have made the mess. You intentionally improve the environment for the next group of campers. Actually the original form of that rule, written by Robert Stephenson Smyth Baden-Powell, the father of scouting, was "Try and leave this world a little better than you found it."
What if we followed a similar rule in our code: "Always check a module in cleaner than when you checked it out." No matter who the original author was, what if we always made some effort, no matter how small, to improve the module. What would be the result?

Depends on 

, 

, 

No : No premature optimization
In Donald Knuth's paper "Structured Programming With GoTo Statements", he wrote: "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

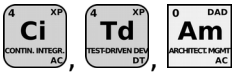


At : Acceptance testing
Acceptance tests are created from user stories. During an iteration the user stories selected during the iteration planning meeting will be translated into acceptance tests. The customer specifies scenarios to test when a user story has been correctly implemented. A story can have one or many acceptance tests, what ever it takes to ensure the functionality works.

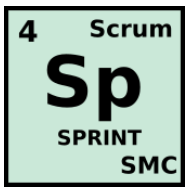
Depends on 

, 

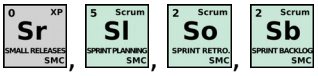
Ac : Automated Tests Coverage
Code Coverage is a measurement of how many lines/blocks/arcs of your code are executed while the automated tests are running.
Code coverage on every dimension should be above possible to 80% (the famous 80/20) rule and close to 100% (TDD).

Depends on 

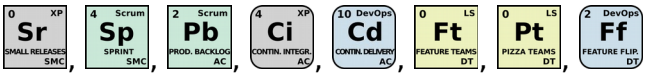
6.4.2 Scrum



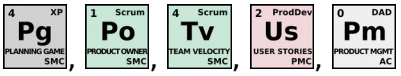
Sp : Sprint
A Sprint is a time-box of one month or less during which a "Done", useable, and potentially releasable product Increment is created. Sprints best have consistent durations throughout a development effort. A new Sprint starts immediately after the conclusion of the previous Sprint.

Depends on 

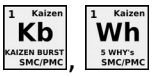
In : Product Increment (Shippable Product)
In Scrum, the Development Team delivers each Sprint a Product Increment.
The increment must consist of thoroughly tested code that has been built into an executable, and the user operation of the functionality is documented either in Help files or user documentation. These requirements are documented in the Definition of Done.
If everything works fine and the Development Team has estimated well, the Product Increment includes all items, which were planned in the Sprint Backlog, tested and documented.

Depends on 

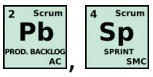
Sl : Sprint Planning
In Scrum, the sprint planning meeting is attended by the product owner, ScrumMaster and the entire Scrum team. Outside stakeholders may attend by invitation of the team, although this is rare in most companies.
During the sprint planning meeting, the product owner describes the highest priority features to the team. The team asks enough questions that they can turn a high-level user story of the product backlog into the more detailed tasks of the sprint backlog.

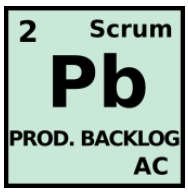
Depends on 

So : Sprint Retrospective
No matter how good a Scrum team is, there is always opportunity to improve. Although a good Scrum team will be constantly looking for improvement opportunities, the team should set aside a brief, dedicated period at the end of each sprint to deliberately reflect on how they are doing and to find ways to improve. This occurs during the sprint retrospective.
The sprint retrospective is usually the last thing done in a sprint. Many teams will do it immediately after the sprint review. The entire team, including both the ScrumMaster and the product owner should participate. You can schedule a scrum retrospective for up to an hour, which is usually quite sufficient. However, occasionally a hot topic will arise or a team conflict will escalate and the retrospective could take significantly longer.

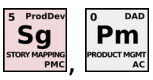
Depends on 

Sb : Sprint Backlog
The sprint backlog is a list of tasks identified by the Scrum team to be completed during the Scrum sprint. During the sprint planning meeting, the team selects some number of product backlog items, usually in the form of user stories, and identifies the tasks necessary to complete each user story. Most teams also estimate how many hours each task will take someone on the team to complete.

Depends on 



Pb : Product Backlog
The agile product backlog in Scrum is a prioritized features list, containing short descriptions of all functionality desired in the product. When applying Scrum, it's not necessary to start a project with a lengthy, upfront effort to document all requirements. Typically, a Scrum team and its product owner begin by writing down everything they can think of for agile backlog priorization. This agile product backlog is almost always more than enough for a first sprint. The Scrum product backlog is then allowed to grow and change as more is learned about the product and its customers.

Depends on 

Sd : Sprint Demo
In Scrum, each sprint is required to deliver a potentially shippable product increment. This means that at the end of each sprint, the team has produced a coded, tested and usable piece of software.
So at the end of each sprint, a sprint review meeting is held. During this meeting, the Scrum team shows what they accomplished during the

---

sprint. Typically this takes the form of a demo of the new features.

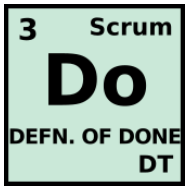
Depends on 

Po : Product Owner
The Scrum product owner is typically a project's key stakeholder. Part of the product owner responsibilities is to have a vision of what he or she wishes to build, and convey that vision to the scrum team. This is key to successfully starting any agile software development project. The agile product owner does this in part through the product backlog, which is a prioritized features list for the product.
The product owner is commonly a lead user of the system or someone from marketing, product management or anyone with a solid understanding of users, the market place, the competition and of future trends for the domain or type of system being developed.

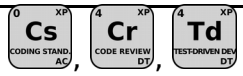
Depends on 

Ds : Daily Scrum
In Scrum, on each day of a sprint, the team holds a daily scrum meeting called the "daily scrum." Meetings are typically held in the same location and at the same time each day. Ideally, a daily scrum meeting is held in the morning, as it helps set the context for the coming day's work. These scrum meetings are strictly time-boxed to 15 minutes. This keeps the discussion brisk but relevant.

Sm : Scrum Master
What is a Scrum Master? The ScrumMaster is responsible for making sure a Scrum team lives by the values and practices of Scrum. The ScrumMaster is often considered a coach for the team, helping the team do the best work it possibly can. The ScrumMaster can also be thought of as a process owner for the team, creating a balance with the project's key stakeholder, who is referred to as the product owner.
The ScrumMaster does anything possible to help the team perform at their highest level. This involves removing any impediments to progress, facilitating meetings, and doing things like working with the product owner to make sure the product backlog is in good shape and ready for the next sprint. The ScrumMaster role is commonly filled by a former project manager or a technical team leader but can be anyone.



Do: Definition of Done
Definition of Done is a simple list of activities (writing code, coding comments, unit testing, integration testing, release notes, design documents, etc.) that add verifiable/demonstrable value to the product. Focusing on value-added steps allows the team to focus on what must be completed in order to build software while eliminating wasteful activities that only complicate software development efforts.

Depends on 

Pp : Planning Poker

Planning Poker is an agile estimating and planning technique that is consensus based. To start a poker planning session, the product owner or customer reads an agile user story or describes a feature to the estimators.

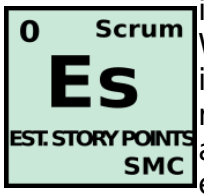
Each estimator is holding a deck of Planning Poker cards with values like 0, 1, 2, 3, 5, 8, 13, 20, 40 and 100, which is the sequence we recommend. The values represent the number of story points, ideal days, or other units in which the team estimates.

The estimators discuss the feature, asking questions of the product owner as needed. When the feature has been fully discussed, each estimator privately selects one card to represent his or her estimate. All cards are then revealed at the same time.

If all estimators selected the same value, that becomes the estimate. If not, the estimators discuss their estimates. The high and low estimators should especially share their reasons. After further discussion, each estimator reselects an estimate card, and all cards are again revealed at the same time.

The poker planning process is repeated until consensus is achieved or until the estimators decide that agile estimating and planning of a particular item needs to be deferred until additional information can be acquired.

Depends on 

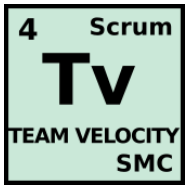


Es : Estimations in Story Points

Story points are a unit of measure for expressing an estimate of the overall effort that will be required to fully implement a product backlog item or any other piece of work.

When we estimate with story points, we assign a point value to each item. The raw values we assign are unimportant. What matters are the relative values. A story that is assigned a 2 should be twice as much as a story that is assigned a 1. It should also be two-thirds of a story that is estimated as 3 story points.

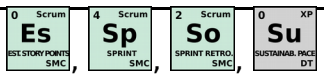
Instead of assigning 1, 2 and 3, that team could instead have assigned 100, 200 and 300. Or 1 million, 2 million and 3 million. It is the ratios that matter, not the actual numbers.



Tv : Team Velocity

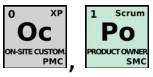
Velocity is simply a metric based on the completed items in a sprint by a single team. The metric is completely subjective to that specific team, and should never be extrapolated for any other comparison.

Velocity is a reflective metric gathered from the sprint throughput of a stable team. Usually, a velocity metric is not considered valid until several sprints have been completed.

Depends on 

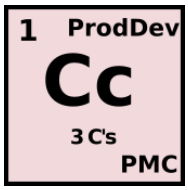
6.4.3 Product Development

Us : User Stories
In software development and product management, a user story is an informal, natural language description of one or more features of a software system. User stories are often written from the perspective of an end user or user of a system. They are often recorded on index cards, on Post-it notes, or in project management software. Depending on the project, user stories may be written by various stakeholders including clients, users, managers or development team members.

Depends on 

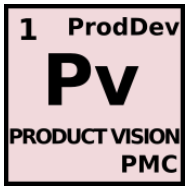
Sg : Story Mapping
Story mapping consists of ordering user stories along two independent dimensions. The "map" arranges user activities along the horizontal axis in rough order of priority (or "the order in which you would describe activities to explain the behaviour of the system"). Down the vertical axis, it represents increasing sophistication of the implementation. Given a story map so arranged, the first horizontal row represents a "walking skeleton", a barebones but usable version of the product. Working through successive rows fleshes out the product with additional functionality.

Depends on 



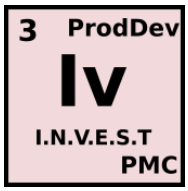
Cc : 3 C's - Card, conversation, confirmation
"Card, Conversation, Confirmation"; this formula (from Ron Jeffries) captures the components of a User Story:
a **"Card"** (or often a Post-It note), a physical token giving tangible and durable form to what would otherwise only be an abstraction;
a **"conversation"** taking place at different time and places during a project between the various people concerned by a given feature of a software product: customers, users, developers, testers; this conversation is largely verbal but most often supplemented by documentation;
the **"confirmation"**, finally, the more formal the better, that the objectives the conversation revolved around have been reached.

Depends on 



Pv : Product Vision (elevator Pitch)
Every Scrum project needs a product vision that acts as the project's true north, sets the direction and guides the Scrum team. It is the overarching goal everyone must share – Product Owner, ScrumMaster, team, management, customers and other stakeholders. As Ken Schwaber puts it: "The minimum plan necessary to start a Scrum project consists of a vision and a Product Backlog. The vision describes why the project is being undertaken and what the desired end state is."

Depends on 



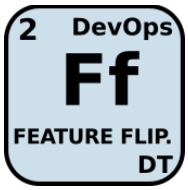
Iv : INVEST
The INVEST mnemonic for agile software projects was created by Bill Wake as a reminder of the characteristics of a good quality User Story:
Independent: The User Story should be self-contained, in a way that there is no inherent dependency on another PBI;
Negotiable: User Stories are no contracts and must leave space for discussion;
Valuable: A User Story must deliver value to the stakeholders;
Estimatable: You must always be able to estimate the size of a User Story;
Small: User Stories should not be so big as to become impossible to plan/task/prioritize with a certain level of accuracy;
TestableThe User Story or its related description must provide the necessary information to make test development possible.

Depends on 

, 

, 

6.4.4 DevOps



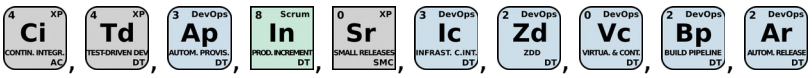
Ff : Feature Flipping
Feature flipping is a technique in software development that attempts to provide an alternative to maintaining multiple source-code branches (known as feature branches), such that the feature can be tested, even before it is completed and ready for release. Feature flipping is used to hide, enable or disable the features, during run time. For example, during the development process, the developer can enable the feature for testing and disable it for remaining users

Depends on 

, 

Cd : Continuous Delivery
Continuous delivery (CD) is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time. It aims at building, testing, and releasing software faster and more frequently. The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production. A

straightforward and repeatable deployment process is important for continuous delivery.

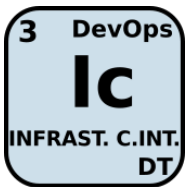
Depends on 

Ap : Automated Provisioning

(Infrastructure as Code) Server provisioning is a set of actions to prepare a server with appropriate systems, data and software, and make it ready for network operation. Typical tasks when provisioning a server are: select a server from a pool of available servers, load the appropriate software (operating system, device drivers, middleware, and applications), appropriately customize and configure the system and the software to create or change a boot image for this server, and then change its parameters, such as IP address, IP Gateway to find associated network and storage resources (sometimes separated as resource provisioning) to audit the system

With DevOps and Automated Provisioning, this whole configuration pipeline should be completely automated and executable in one-click, either automatically or on-demand.

Depends on 



Ic : Infrastructure Continuous Integration

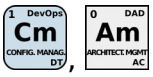
(Infrastructure as Code) Infrastructure Continuous Integration consists in leveraging Continuous Integration techniques to Infrastructure components.

The continuous integration system is necessarily complex, spanning the development, test and staging environments. The continuous integration build should continuously build and test the provisioning, configuring and maintaining of the various infrastructure components.

Depends on 

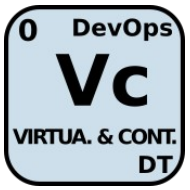
Zd : Zero Downtime Deployments

A Zero Downtime Deployment consists in redeploying (typically for a software upgrade) a production system without any downtime appearing to end users. To achieve such lofty goals, redundancy becomes a critical requirement at every level of your infrastructure. There are various techniques involved such a canari release or blue-green deployments.

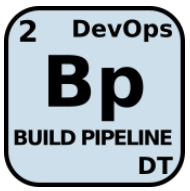
Depends on 

Cm : Configuration Management
Configuration management is a class of tool supporting the automation of the configuration of a system, platform or software. It typically consists in *define-with-code* the various config elements that prepare a provisioned compute resource (like a server or AWS Ec2 instance) for service (installing software, setting up users, configuring services, placing files with template-defined variables, defining external config resources like DNS records in a relevant zone).

Depends on 



Vc : Virtualization and Containers
Hardware virtualization or platform virtualization refers to the creation of a virtual machine that acts like a real computer with an operating system. Software executed on these virtual machines is separated from the underlying hardware resources.
Containerization - also called container-based virtualization and application containerization - is an OS-level virtualization method for deploying and running distributed applications without launching an entire VM for each application. Instead, multiple isolated systems, called containers, are run on a single control host and access a single kernel.



Bp : Build Pipelines
Build pipelines are integrated views of downstream and upstream build jobs on a build server. Build pipelines are requires to automated all the various tasks towards continuous delivery such as : provisionning of the environment, build of the various software (with compilation, tests, packaging, etc.), deployment of the software components, applying configuration and testing the deployed platform.

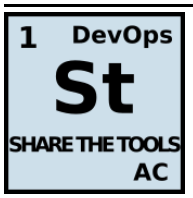
Depends on 

, 

Ar : Automated Releases
Release Automation consists in automating all the various steps required to release a new version of a software: building, testing, tagging, branching et deploying the binaries to a Binary management tools.

Depends on 

, 

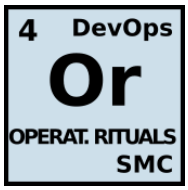


St : Share the tools
Share the tools is a DevOps principles aimed at leveraging both Dev and Ops tools and practices to the other side of the wall. Developers should leverage their automation and building tool to Infrastructure Automation, Provisioning and Testing. Ops should share the production monitoring concerns with developers.

Depends on 

Os : Operators are stakeholders
Operators as stakeholders is a DevOps principle stating that Operators should be considered the other users of the platform. They should be fully integrated in the Software Development Process.
At specification time, operators should give their non-functional requirements just as business users give their functional requirement. Such non-functional requirements should be handled with same important and priority by the development team.
At implementation time, operators should provide feedback and non-functional tests specifications continuously just as business users provides feedback on functional features.

Depends on 



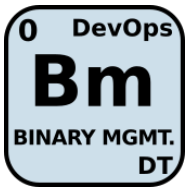
Or : Operators in Rituals
Operators in Rituals is a DevOps principle stating that operators should be integrated in the Development Team Rituals such as the Sprint Planning and Sprint Retrospective and represent non-functional constraints during these rituals just as the Product Owner represents the functional interests.

Depends on 

, 

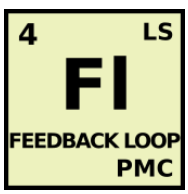
, 

, 



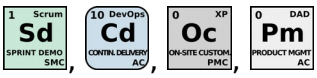
Bm : Binaries Management
A binary repository manager is a software tool designed to optimize the download and storage of binary files used and produced in software development. It centralizes the management of all the binary artifacts generated and used by the organization to overcome the complexity arising from the diversity of binary artifact types, their position in the overall workflow and the dependencies between them.
A binary repository is a software repository for packages, artifacts and their corresponding metadata. It can be used to store binary files produced by an organization itself, such as product releases and nightly product builds, or for third party binaries which must be treated differently for both technical and legal reasons.

6.4.5 Lean Startup



Fl : Feedback Loop
The *Build-Measure-Learn* feedback loop is one of the central principles of Lean Startup Method.
A startup is to find a successful revenue model that can be developed with further investment. Build-Measure-Learn is a framework for establishing – and continuously improving – the effectiveness of new products, services and ideas quickly and cost-effectively.
In practice, the model involves a cycle of creating and testing

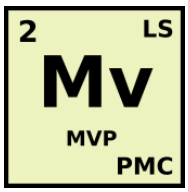
hypotheses by building something small for potential customers to try, measuring their reactions, and learning from the results.

Depends on 

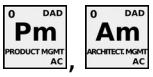
Ft : feature Teams
A *feature team* is a long-lived, cross-functional, cross-component team that completes many end-to-end customer features—one by one. It is opposed to the traditional approach of *Component Team* where a team is specialized on an individual software components and maintains it over several projects at the same time.
The Feature team approach seeks to avoid the bottlenecks usually appearing with Component Teams.

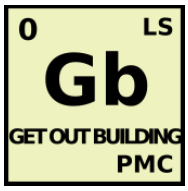
Fa : Fail Fast
Fail fast means getting out of planning mode and into testing mode, eventually for every critical component of your model of change.
Customer development is the process that embodies this principle and helps you determine which hypotheses to start with and which are the most critical for your new idea.
An important goal of the philosophy is to cut losses when testing reveals something isn't working and quickly try something else, a concept known as pivoting.

Depends on 



Mv : MVP
In product development, the minimum viable product (MVP) is a product with just enough features to satisfy early customers, and to provide feedback for future development

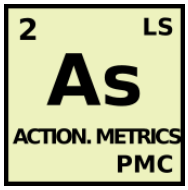
Depends on 



Gb : Get Out of the building
If you are pre-Product/Market Fit and you aren't actually "Getting out of the Building" (actually talking to your customers), you aren't doing Customer Development, and your startup isn't a Lean Startup.
Again: If you aren't actually talking to your customers, you aren't doing Customer Development.

Pt : Pizza Teams
The idea of a "two pizza team" was coined by Jeff Bezos, founder of Amazon.com. If you can't feed a team with two pizzas, it's too large. That limits a task force to five to seven people, depending on their appetites."
The underlying idea is that as a team's size grows, the amount of one-on-one communication channels tend to explode.
Beyond ten, communication loses efficiency, cohesion diminishes, parasitism behaviors and power struggles appear, and the performance

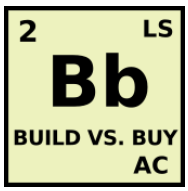
of the team decreases very rapidly with the number of members.



As : Actionable Metrics
The only metrics that entrepreneurs should invest energy in collecting are those that help them make decisions. Actionable Metrics are opposed to Vanity Metrics.
This is a precision of another fundamental Lean Startup practice wich is "Obsession of Measure" stating that everything should be measured and no decision should be taken in the company if it is not supported by a Key Process Indicator or a Key Risk Indicator.

Depends on 

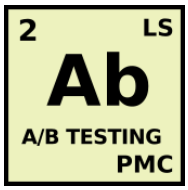
, 



Bb : Build vs. Buy
This is a fundamental principle of the Lean Startup and the web giants : favor as much as possible building your own software, your own feature instead of buying a third party software or library.
When initiating a startup, having to pay fees to third party corporations before reaching a sustainable growth is suicidal.

Depends on 

, 

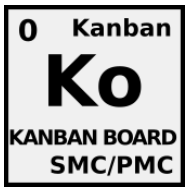


Ab : A/B Testing
In marketing and business intelligence, A/B testing is a term for a controlled experiment with two variants, A and B. It can be considered as a form of statistical hypothesis testing with two variants leading to the technical term, two-sample hypothesis testing, used in the field of statistics

Depends on 

, 

6.4.6 Kanban



Ko : Kanban Board
A Kanban board is a work and workflow visualization tool that enables you to optimize the flow of your work. Physical Kanban boards typically use sticky notes on a whiteboard to communicate status, progress, and issues.
An agile corporation should use a KanBan board to monitor all its processes.
A development team will typically use a Kanban board to monitor the Sprint backlog completion during a sprint.

6.4.7 Kaizen

Kb : Kaizen Burst
The Kaizen burst is a specific Kaizen process integrated the the development rituals. In Agile Software Development, it is really integrated in the Sprint Retrospective. This idea is to identify in a visual way (with a post-it on a board for instance) the weaknesses or problems in the development practices or processes. These boxes are called Kaizen burst.
Theses boxes are commented as actions are taken towards improvement and eventuelly removed when the weakness has been adressed or the problem solved.

Depends on 

Wh : 5 Why
5 Whys is an iterative interrogative technique used to explore the cause-and-effect relationships underlying a particular problem.
The primary goal of the technique is to determine the root cause of a defect or problem by repeating the question "Why?" Each answer forms the basis of the next question. The "5" in the name derives from an anecdotal observation on the number of iterations needed to resolve the problem.

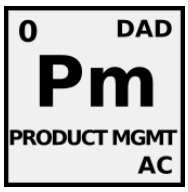
Depends on 

6.4.8 FDD (Feature Driven Development)

Si : SOLID principles
In computer programming, the term SOLID is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible and maintainable. The principles are a subset of many princples promoted by Robert C. Martin.
Though they apply to any object-oriented design, the SOLID principles can also form a core philosophy for methodologies such as agile development or Adaptive Software Development.
The 5 principles are as follows:
SRP : Single responsibility principle - a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
OCP : Open/closed principle - "software entities ... should be open for extension, but closed for modification."
LSP : Liskov substitution principle - "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."
ISP : Interface segregation principle - "many client-specific interfaces are better than one general-purpose interface."
DIP : Dependency inversion principle - one should "depend upon abstractions, not concretions."

Depends on 

6.4.9 DAD



Pm : Product Management Committee
The Product Management Committee is both a team and a ritual that enforces a smart approach to product management.
Product Management consists in identifying and evolving your organization’s business vision; identifying and prioritizing potential products/solutions to support that vision; identifying, prioritizing, and allocating features to products under development; managing functional dependencies between products; and marketing those products to their potential customers.
The Product Management Committee is the weekly (or bi-weekly) ritual enforcing and supporting this process with the required role attending the committee. It is led by the product Owner which has more a role of facilitator and arbitrator that a formal decision role. The Product Owner represents the PMC to the development team.

Am : Architecture Committee
The Architecture Committee is responsible to analyze user stories and define Development Tasks. Every story should be specified, designed and discussed. Screen mockups if applicable should be drawn, acceptance criteria agreed, etc.
Since the Architecture Committee is also responsible for estimating Stories, it's important that representatives of the Development Team, not only the Tech Leads and the Architects, but simple developers as well, take part in it.