Agile Planning : tools and processes

by Jerome Kehrli

Written in June, 2017

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 mentionned 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 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.

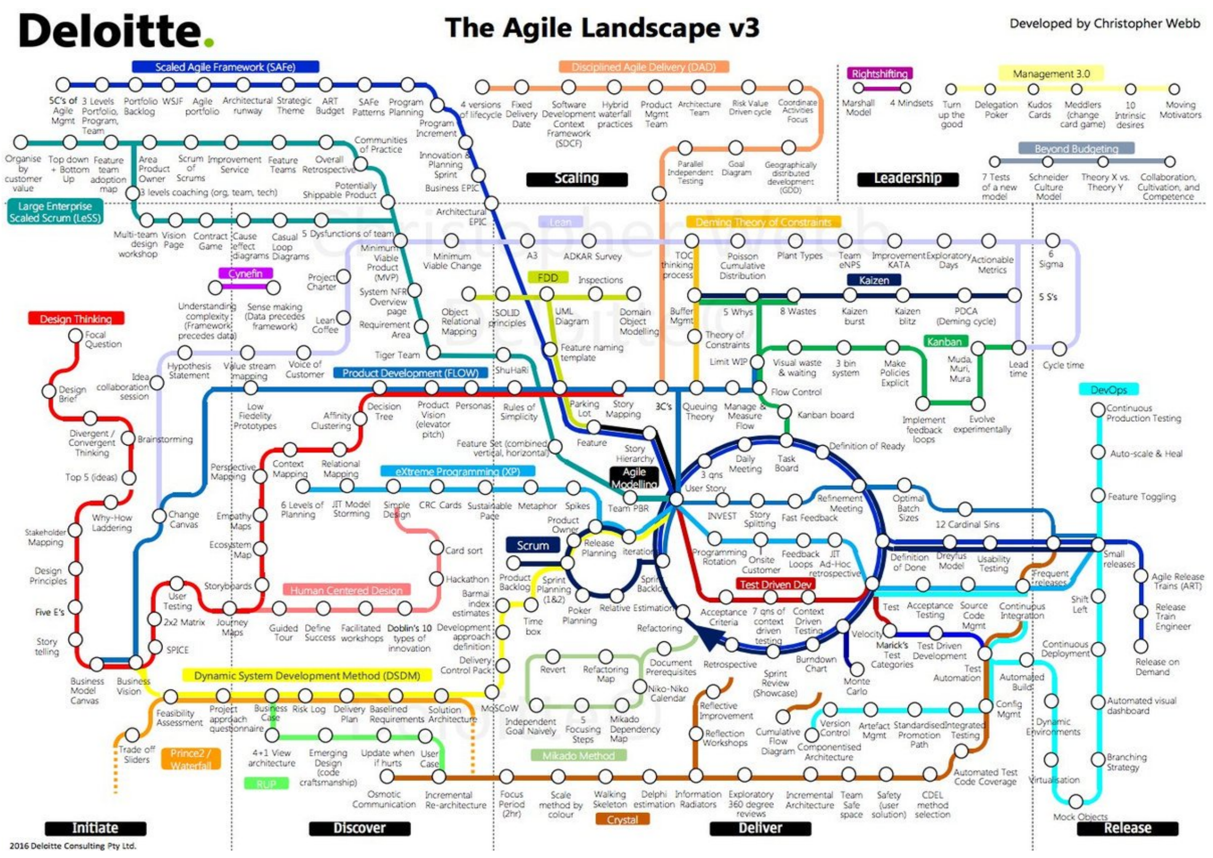
In this report, I intend to present the fundamentals, the roles, the processes, the rituals and the values that I believe a team would need to embrace to achieve success down the line in Agile Software Development Management - Product Management, Team Management and Project Management - with the ultimate 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.

Table of Contents

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:



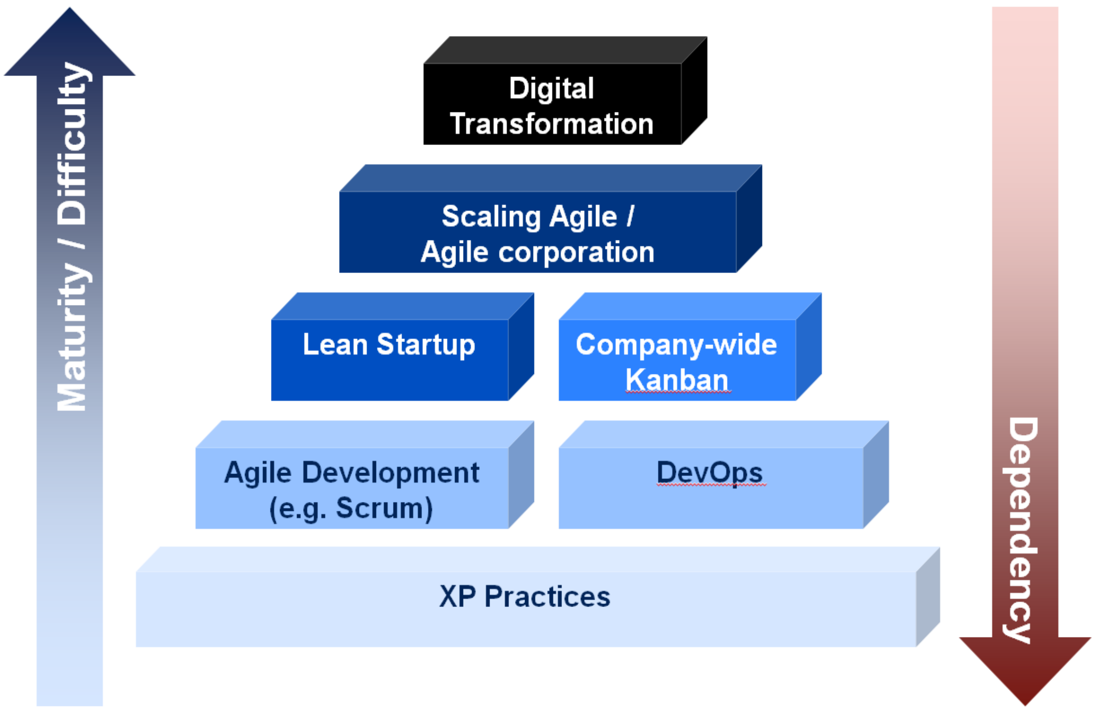
(Source : Christopher Webb - LAST Conference 2016 Agile Landscape - https://www.slideshare.net/ChrisWebb6/last-conference-2016-agile-landscape-presentation-v1)

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.

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.

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:

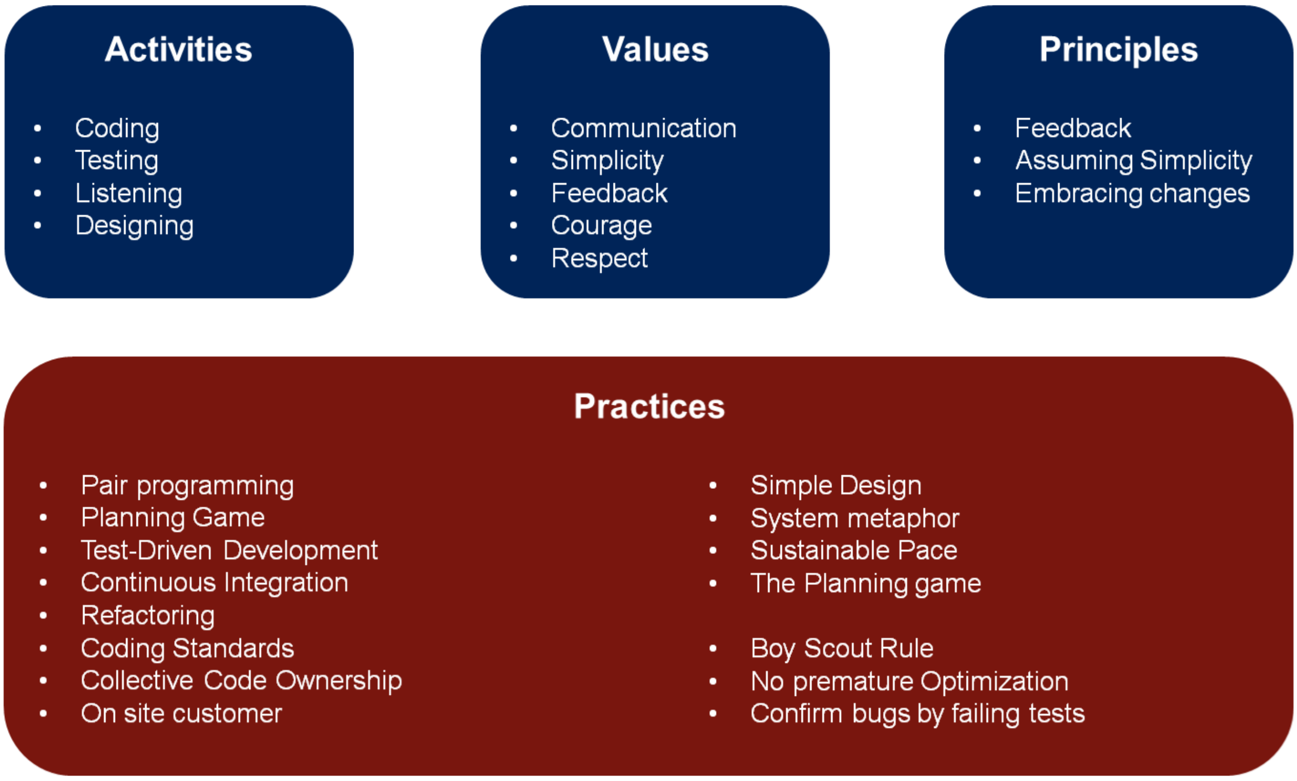
- An attempt to reconcile the human with productivity;
- A mechanism to facilitate social change;
- A way of improvement;
- A style of development;
- A discipline in the development of computer applications.

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

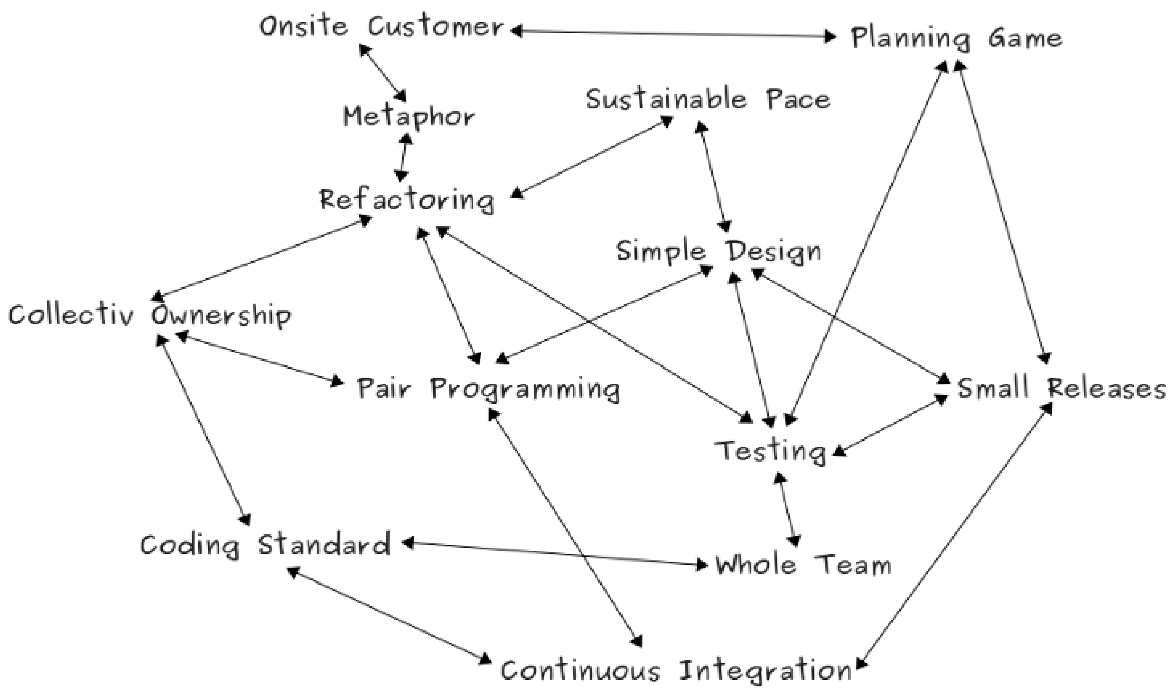
Practices

- 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:



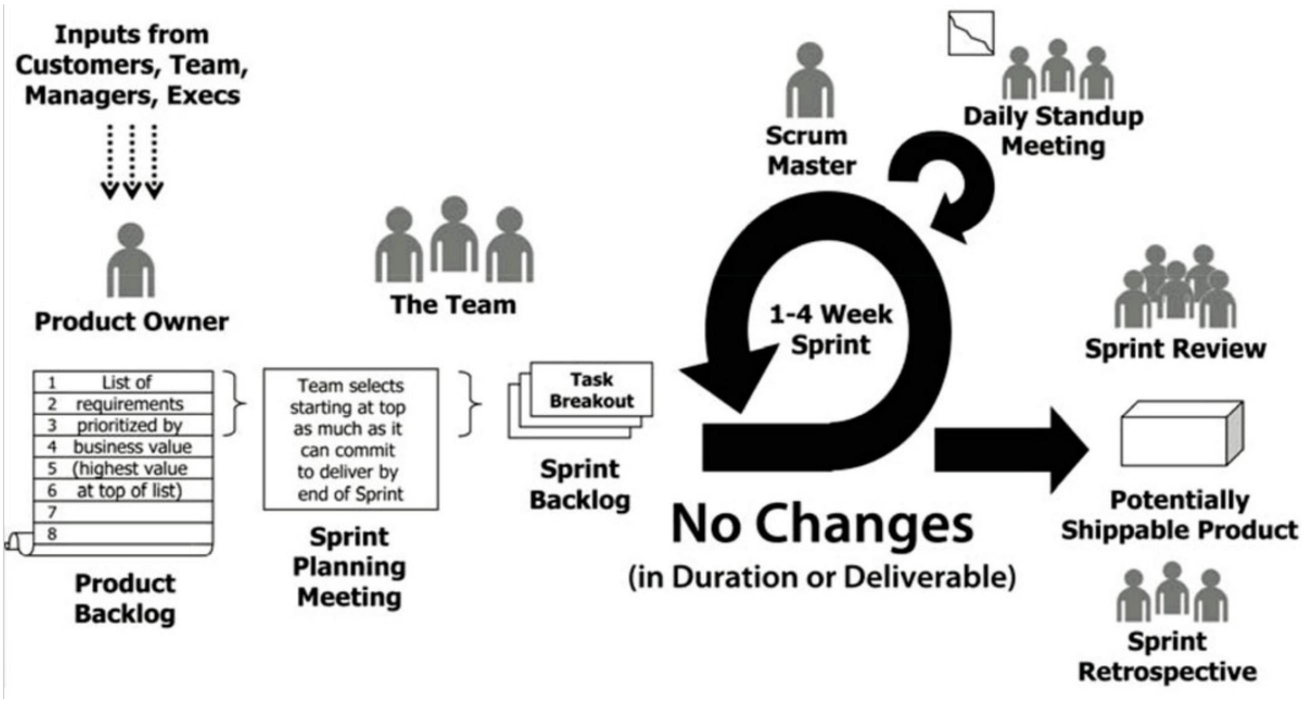
All of this makes a lot of sense if you think of it: doing refactorings without TDD would be suicidal, Continuous Integration without TDD as well, Testing without simple design is complicated, Simple Design is enforced by TDD, etc.

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 in a previous article here.

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*."

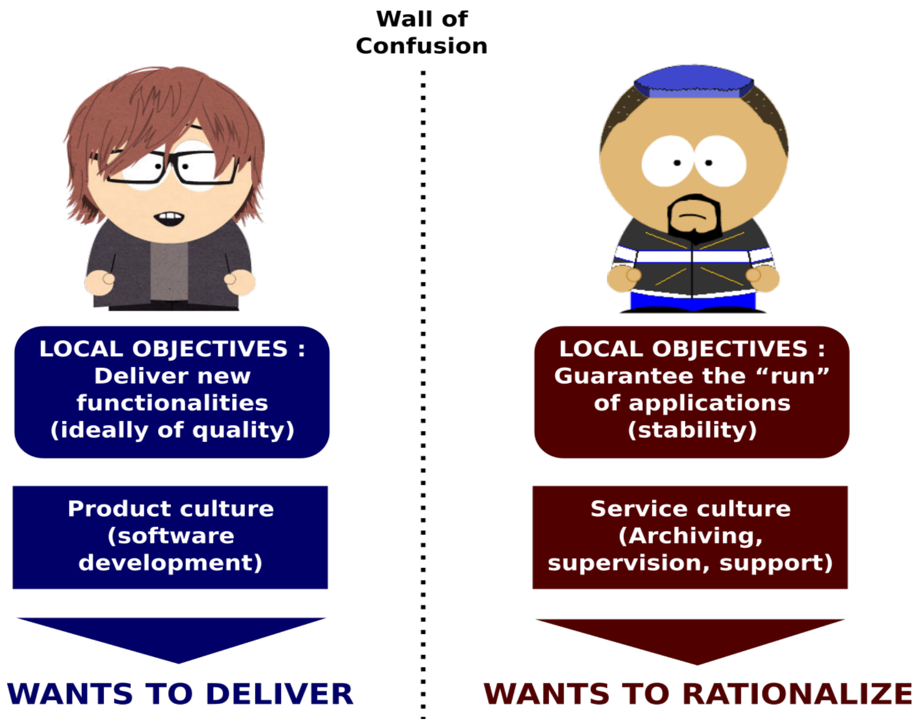
In Scrum these numbers are called **Story Points** - or **SP**.

2.3 DevOps

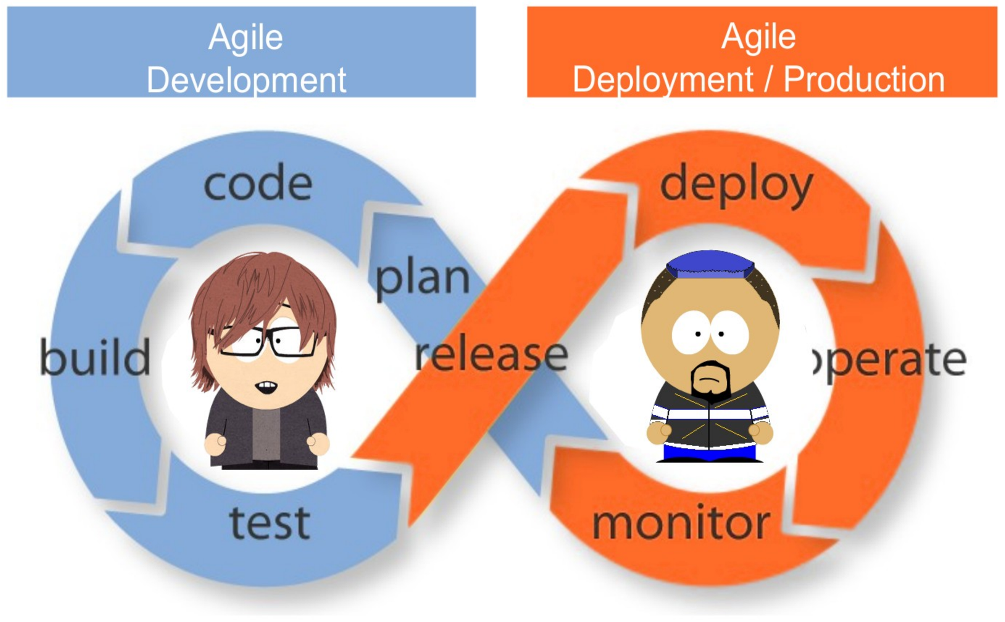
DevOps is a methodology capturing the practices adopted from the very start by the web giants who had a unique opportunity as well as a strong requirement to invent new ways of working due to the very nature of their business: the need to evolve their systems at an unprecedented pace as well as extend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


Wall of Confusion
LOCAL OBJECTIVES : Deliver new functionalities (ideally of quality)
Product culture (software development)
WANTS TO DELIVER
LOCAL OBJECTIVES : Guarantee the "run" of applications (stability)
Service culture (Archiving, supervision, support)
WANTS TO RATIONALIZE

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.



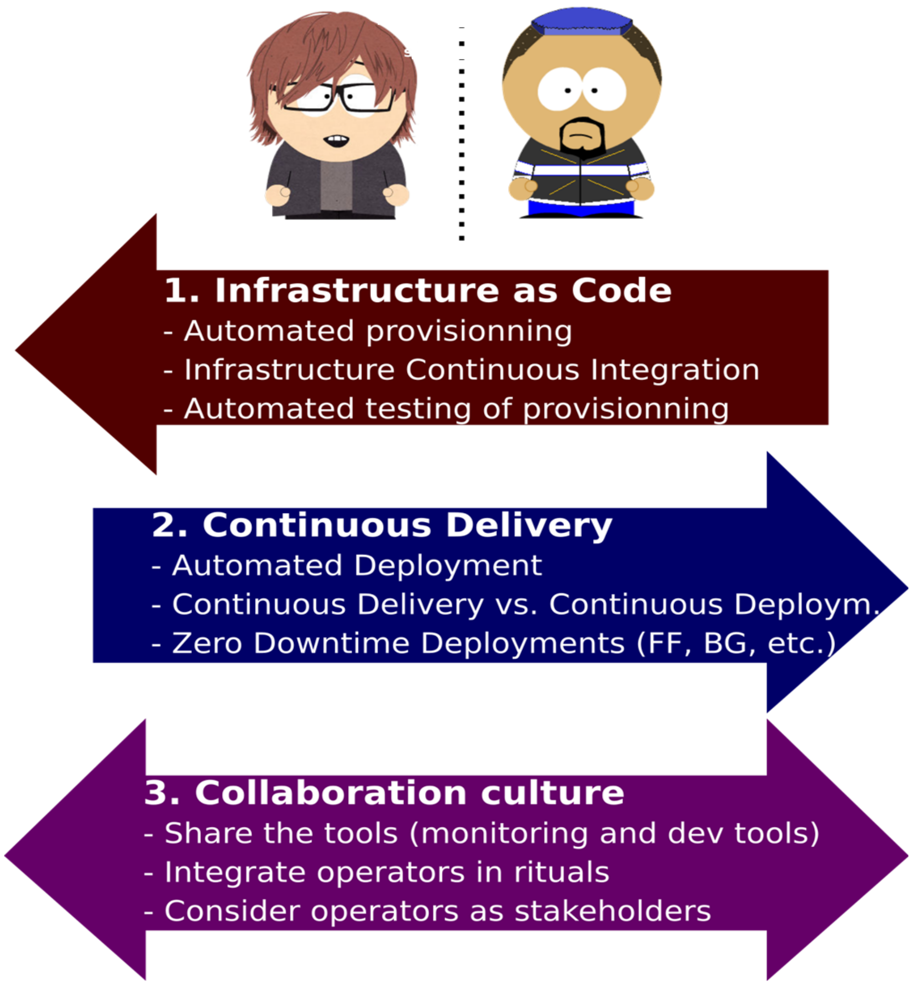
DevOps encourages **communication**, **collaboration**, **integration** and **automation** among software developers and IT operators in order to improve both the speed and quality of delivering software.

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 ?


1. Infrastructure as Code
- Automated provisionning
- Infrastructure Continuous Integration
- Automated testing of provisionning
2. Continuous Delivery
- Automated Deployment
- Continuous Delivery vs. Continuous Deploym.
- Zero Downtime Deployments (FF, BG, etc.)
3. Collaboration culture
- Share the tools (monitoring and dev tools)
- Integrate operators in rituals
- Consider operators as stakeholders

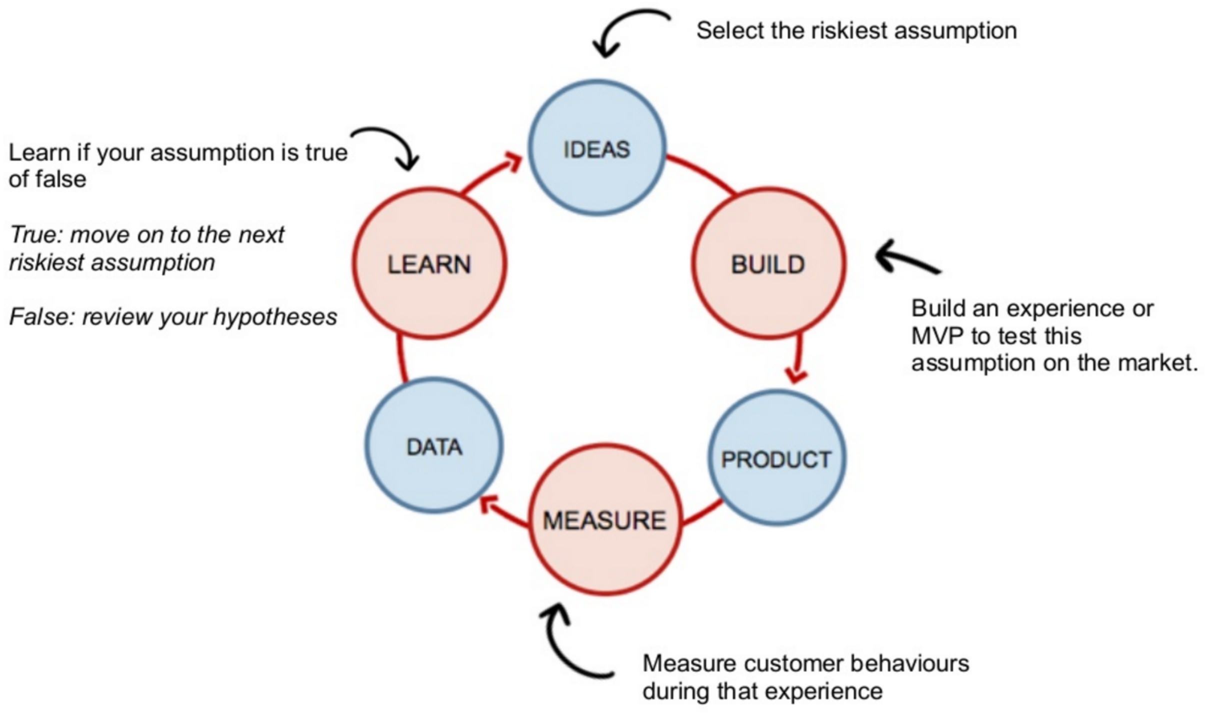
I presented these principles and practices in details in my previous article dedicated to devops..

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.

(2013)
The Four Steps to the Epiphany
Successful Strategies for Products that Win
Steven Gary Blank
Steve Blank
(2011)
Business Model Generation
Alex Osterwalder
THE LEAN STARTUP
ERIC RIES
Eric Ries
(2011)
RUNNING LEAN
ITERATE FROM PLAN A TO A PLAN THAT WORKS
Ash Maurya
(2012)

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.


Select the riskiest assumption
IDEAS
BUILD
Build an experience or MVP to test this assumption on the market.
PRODUCT
MEASURE
Measure customer behaviours during that experience
DATA
LEARN
Learn if your assumption is true of false
True: move on to the next riskiest assumption
False: review your hypotheses

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.

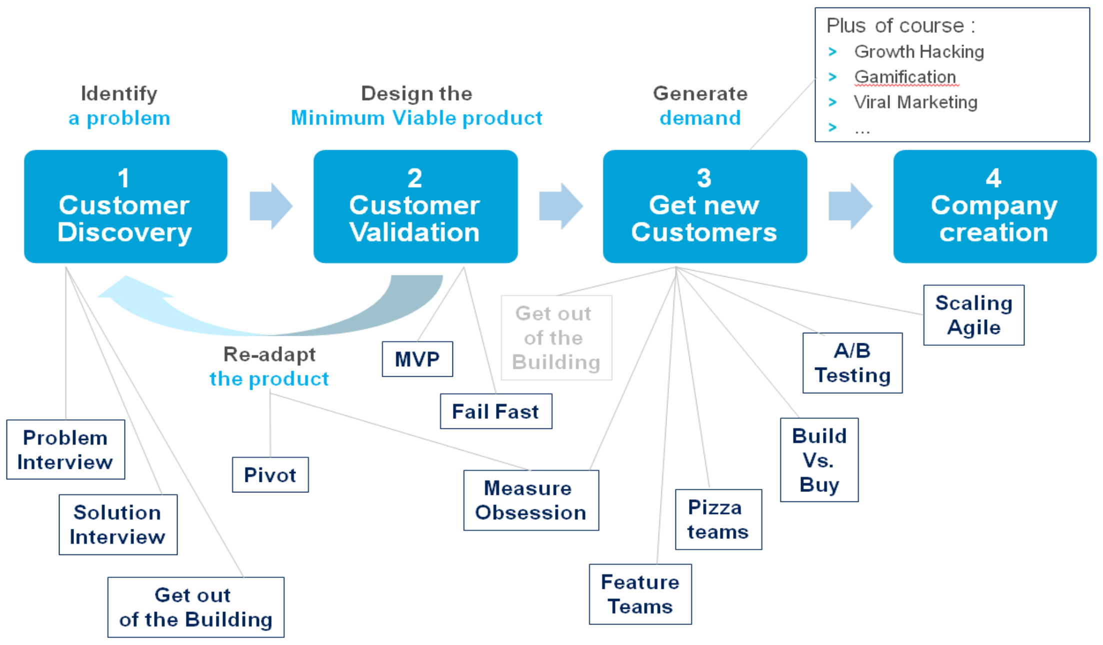
The four steps to the Epiphany

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:



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

I discussed these principles and practices in length in a previous article on this blog.

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

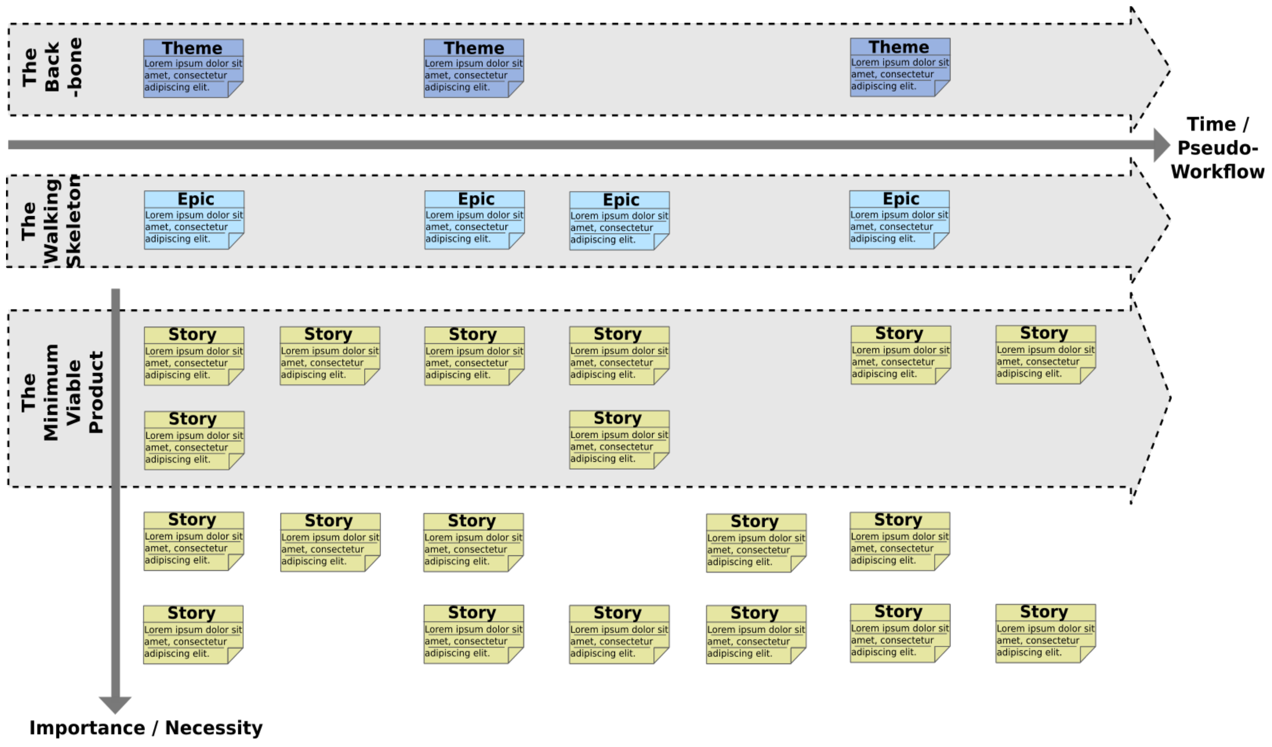
believe that Visual Management with sound tools and approaches is not optional.
At the end of the day, as we ill see, a good Project Management tool is a tool than enables anyone in the company to understand what is achievable in a given time or what time is required to deliver a given scope within a 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:

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:

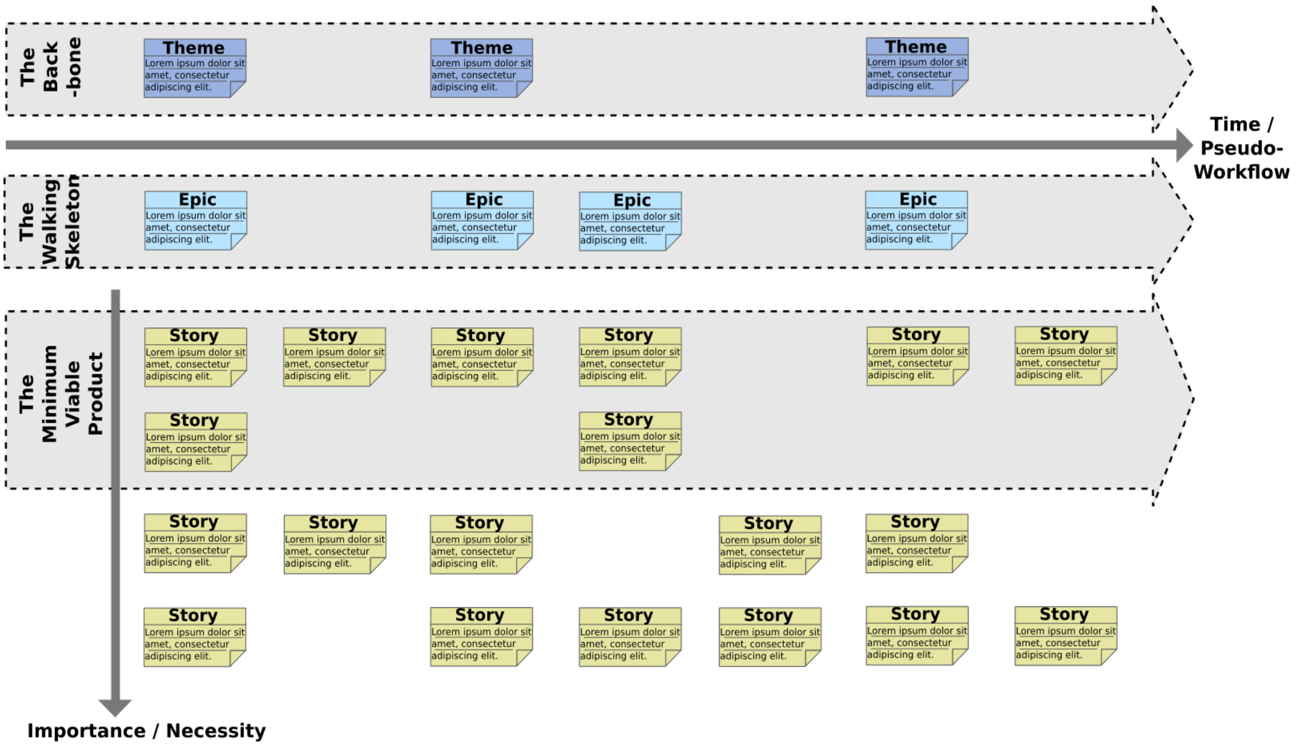


At the top of the map are "big stories." We call them **themes**. A theme is sort of a big thing that people do - something that has lots of steps, and doesn't always have a precise workflow. A theme is a big category containing actual user stories grouped in **Epics**.

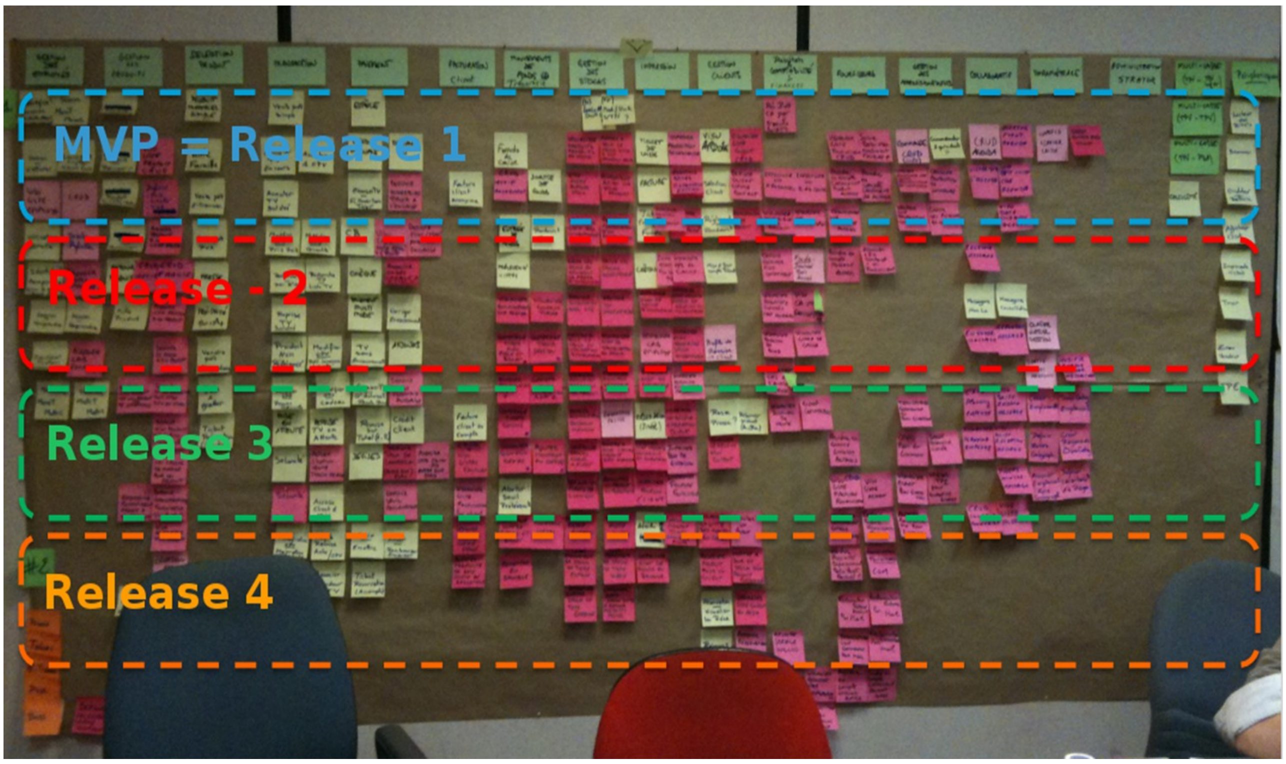
Epics are big user stories such as the one mentioned in example above. They usually involve a lot of development and cannot be considered as is in an actual product backlog. For this reason, Epics are split in a sub-set of **stories**, more precise and concrete that are candidate to be put in an actual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

I presented more information on Story Maps in a previous article here.
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:

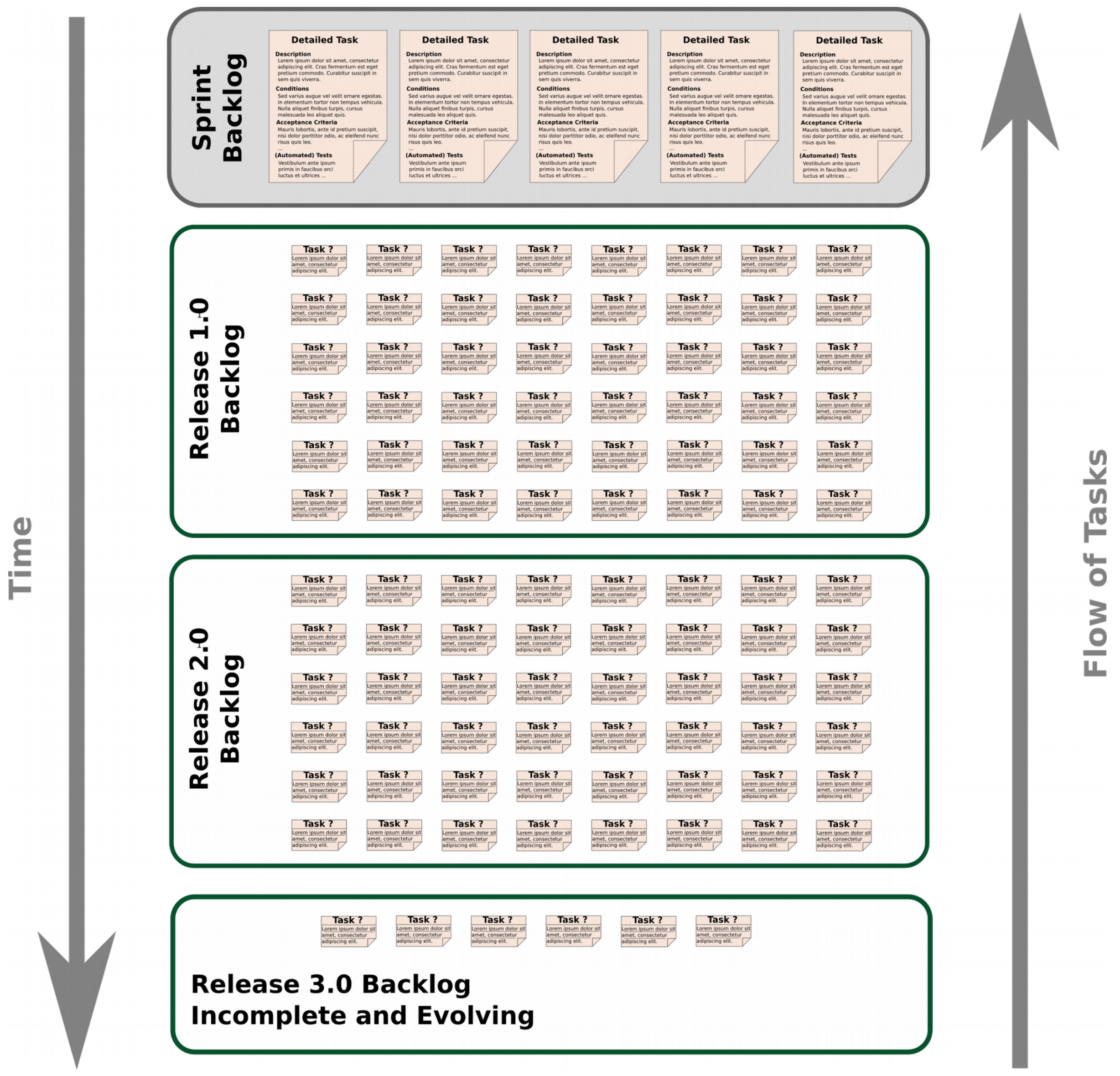


(Copyright OCTO Technology / *Unfortunately I haven't been able to recover the source*)

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.

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.

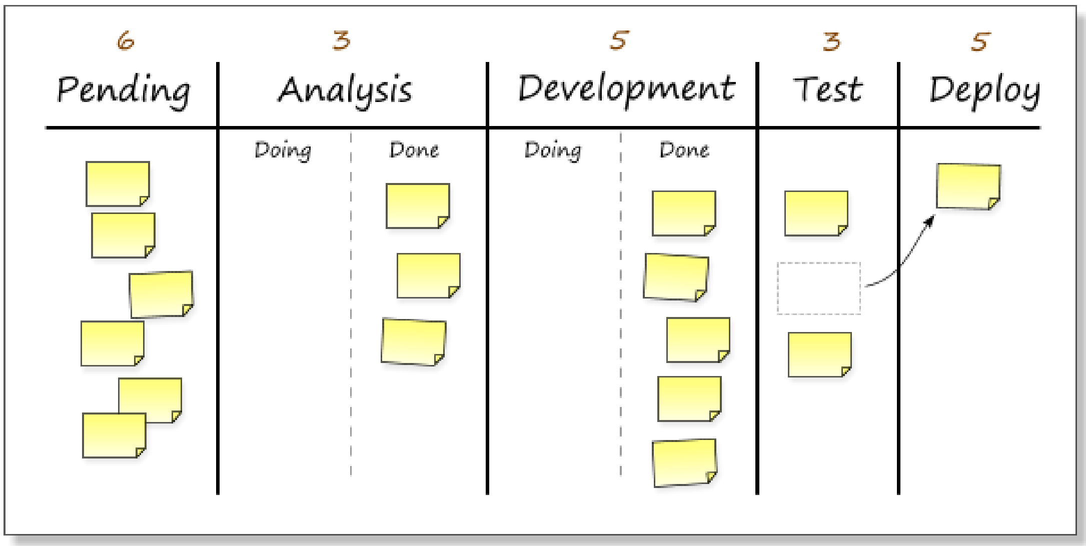
Some important constraints should be noted right away:

- 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.

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:

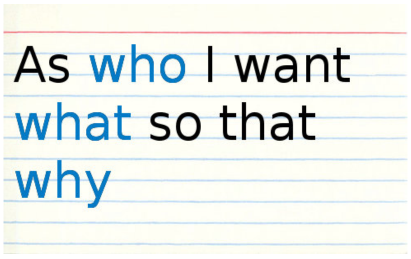
2.5.4 User Stories

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:

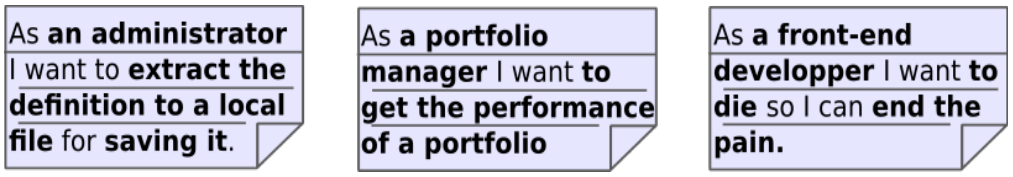
As a <type of user>, I want <some goal> so that <some reason>.

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:


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

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

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

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. Principles

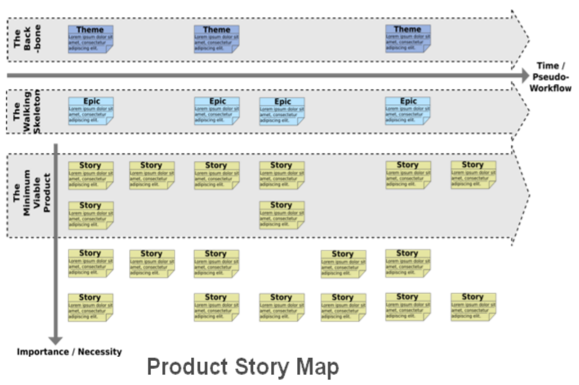
Having covered the fundamentals, we will now go through the principles required for **Agile Planning** and see how the principles and practices introduced in the previous section should be used to achieve *reliable forecasts and planning* with Agile methodologies.

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.

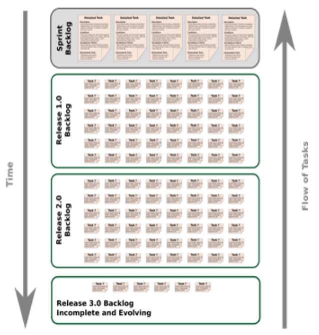
3.1 The tools

The tools that the organization should adopt are as follows:



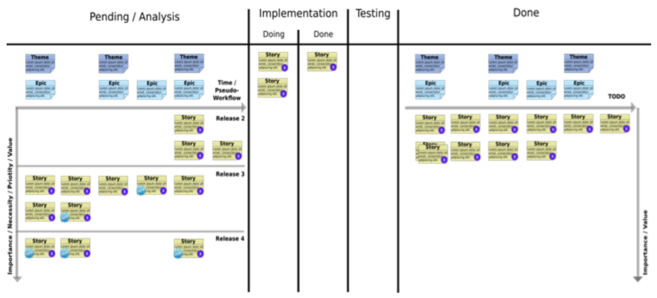
Product Story Map

- Product Management Tool
- Visual Tool



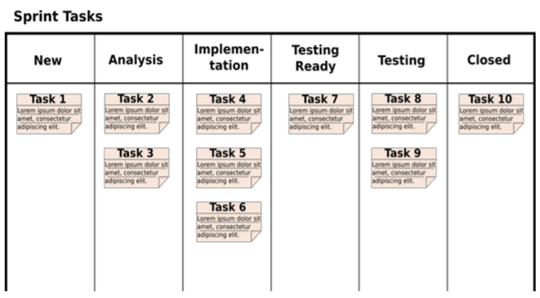
Product Backlog

- 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 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.

3.2 The Organization

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

3.2.1 Required roles

The required roles are as follows:

Team Leader

Architect

Tech Leads and Developers

Product Owner

Business Represent.

- **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 ?

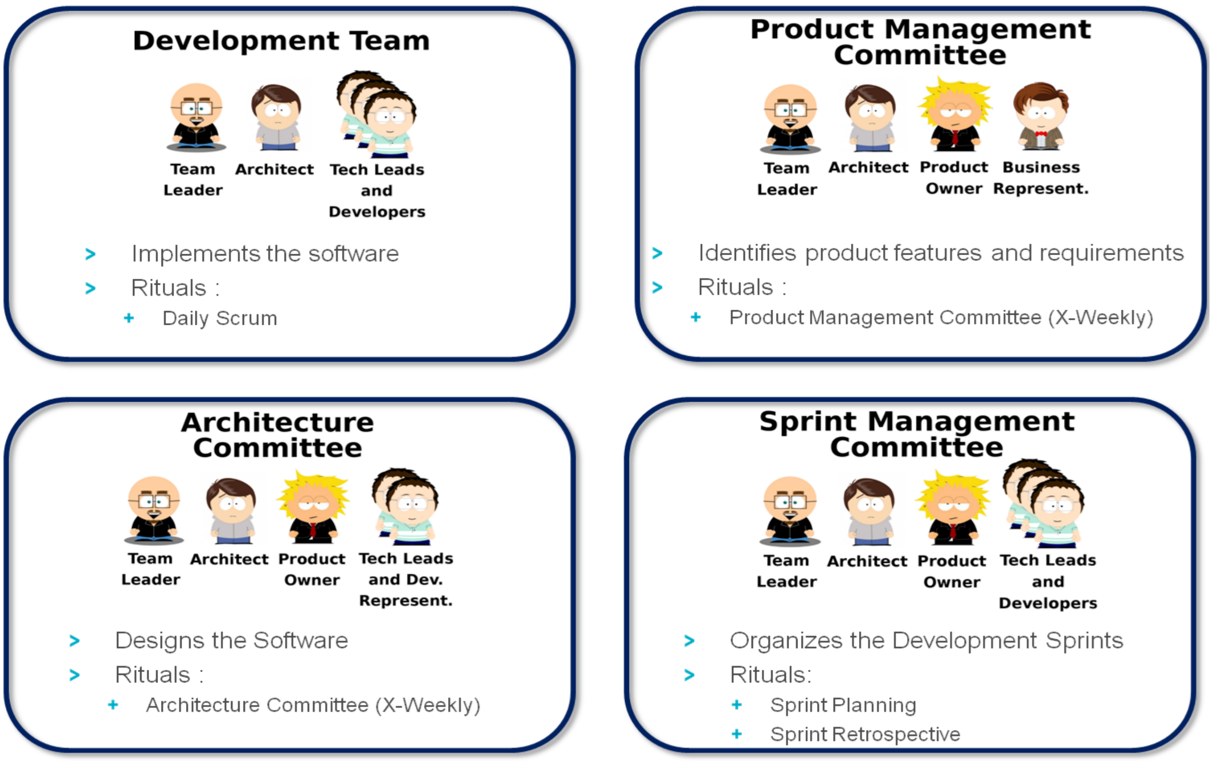
Roles are required mostly for two reasons : **efficiency** and **focus**:

- **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.

3.2.2 Required Committees and teams

Required committees and teams are as follows:


Development Team
Team Leader
Architect
Tech Leads and Developers
> Implements the software
> Rituals :
+ Daily Scrum
Product Management Committee
Team Leader
Architect
Product Owner
Business Represent.
> Identifies product features and requirements
> Rituals :
+ Product Management Committee (X-Weekly)
Architecture Committee
Team Leader
Architect
Product Owner
Tech Leads and Dev. Represent.
> Designs the Software
> Rituals :
+ Architecture Committee (X-Weekly)
Sprint Management Committee
Team Leader
Architect
Product Owner
Tech Leads and Developers
> Organizes the Development Sprints
> Rituals:
+ Sprint Planning
+ Sprint Retrospective

- **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

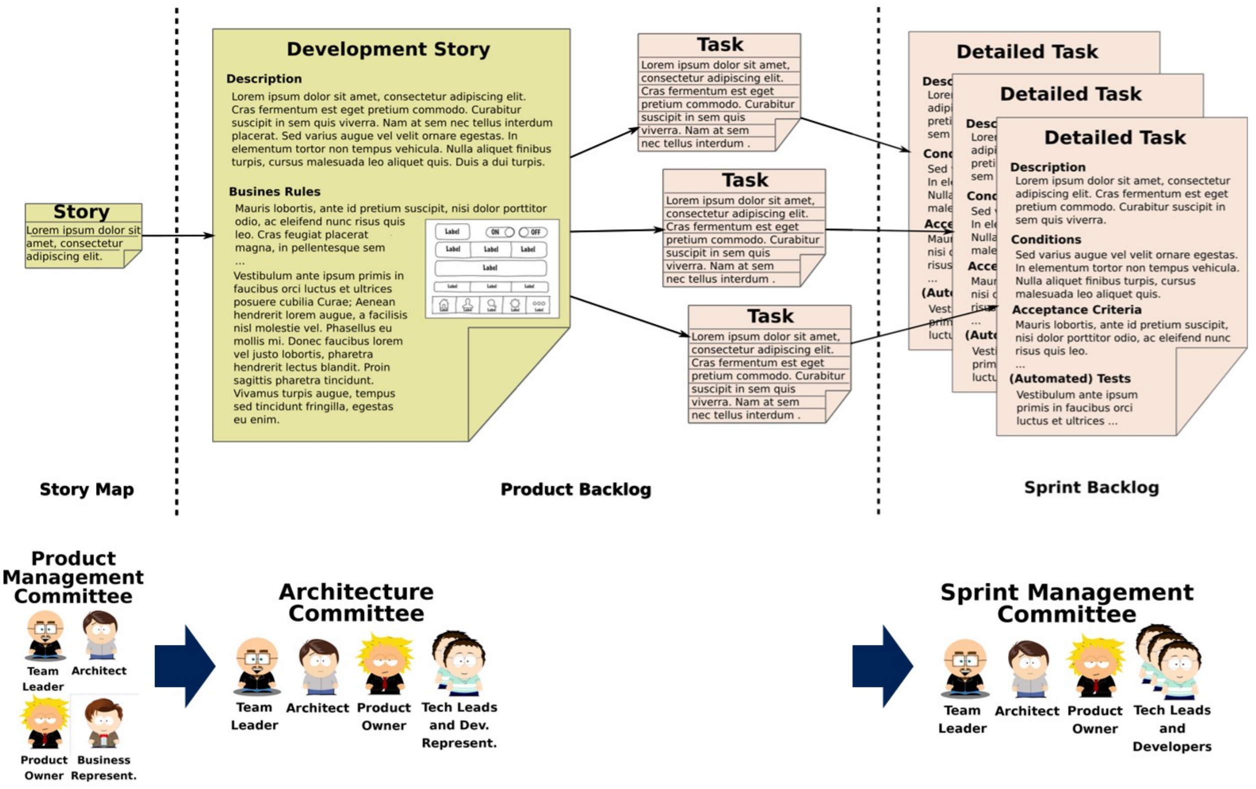
issues and drawbacks encountered during former sprint and agrees on an action plan to address them.

3.3 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.

3.3.1 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:

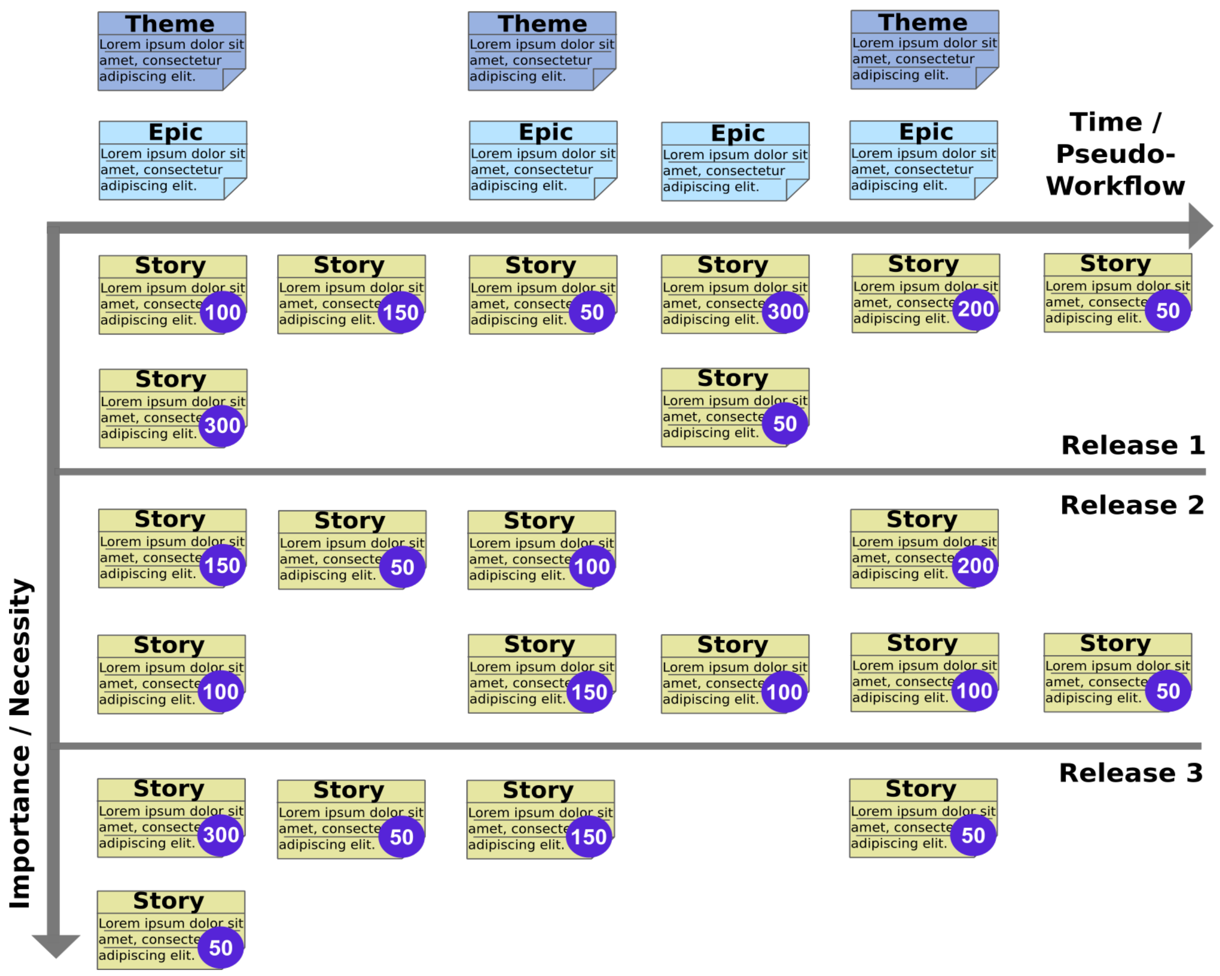
- Are all the information provided by the Architecture Committee clear enough or are some precisions required ?
- Is there any unforeseen impact on other parts of the software ?

- 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.

3.3.2 Estimation Process

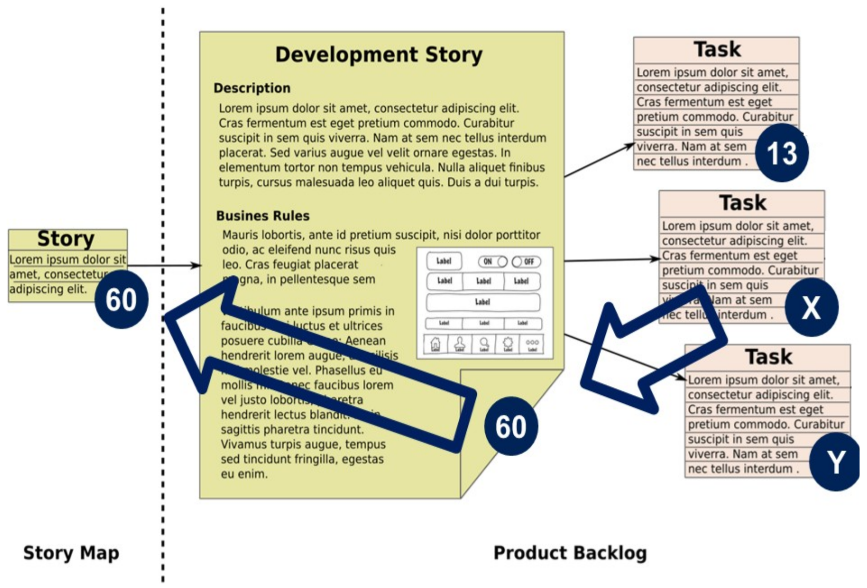
What we want eventually, is a Story Map containing estimations for all the Stories that have been analyzed by the Architecture Committee.
The result we want to achieve here can be represented as follows:



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

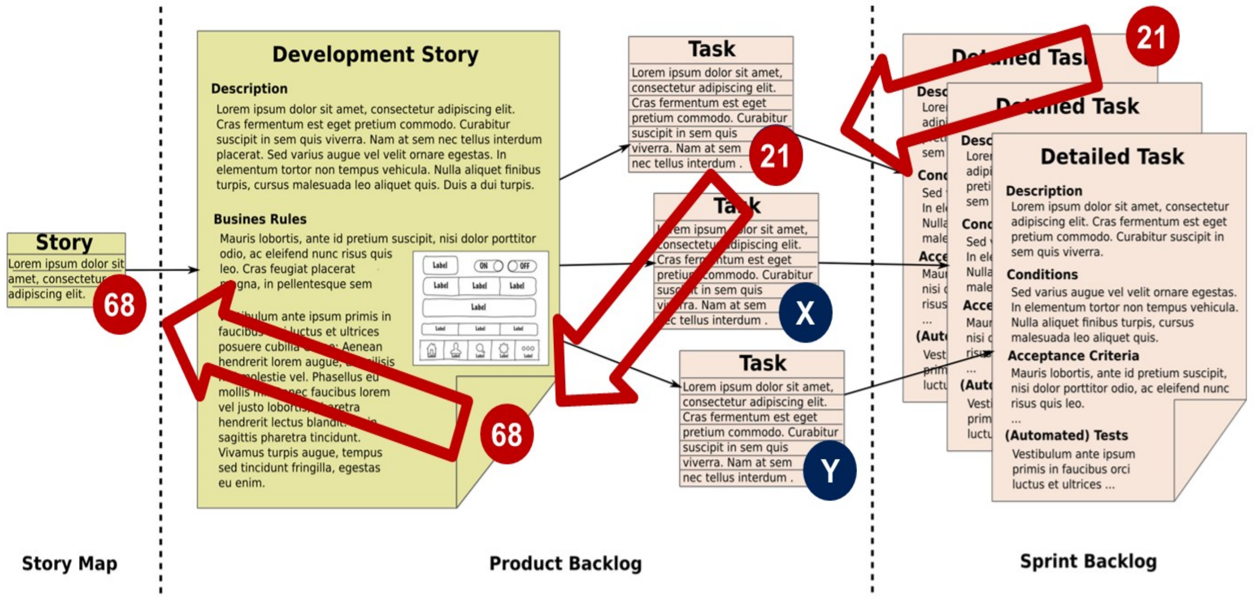
At this stage, The **Architecture Committee** is in charge of the Initial Estimations.
After a Story has been broken down in tasks, each and every of these tasks is estimated by the Committee using the *Planning Poker* approach.
The sum of the estimations of every individual tasks is reported on both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

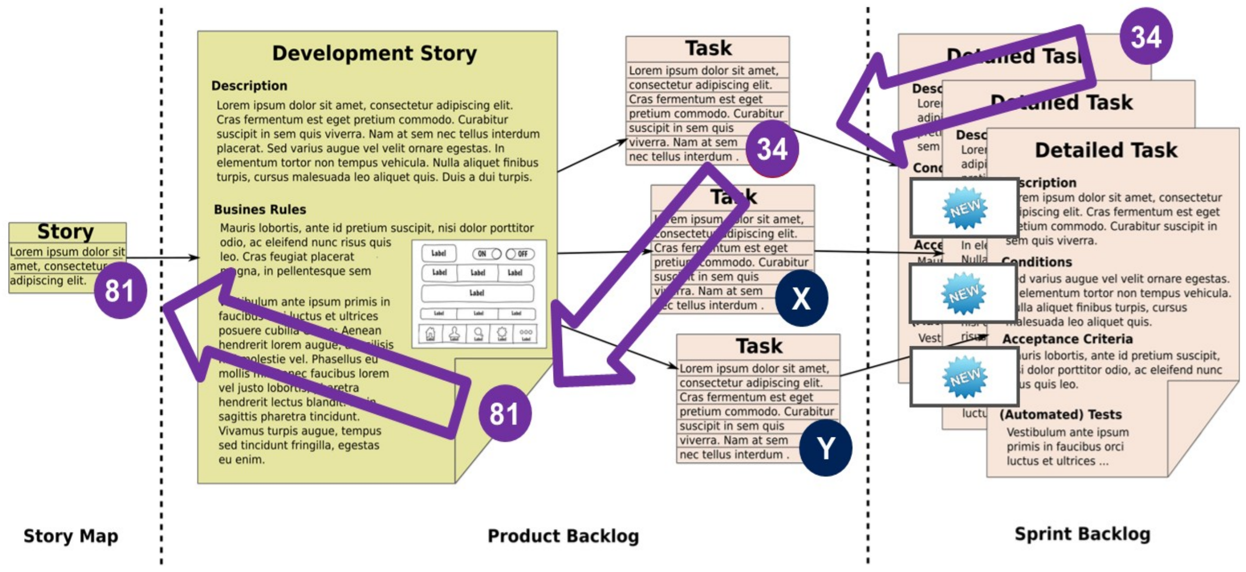
The Sprint Management Committee should challenge the initial estimations with their new findings and adapt the estimations accordingly.
Again, these new *Refined Estimations* should be reported on both the Development Story (Product Backlog) and the User Story (Story Map):



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.

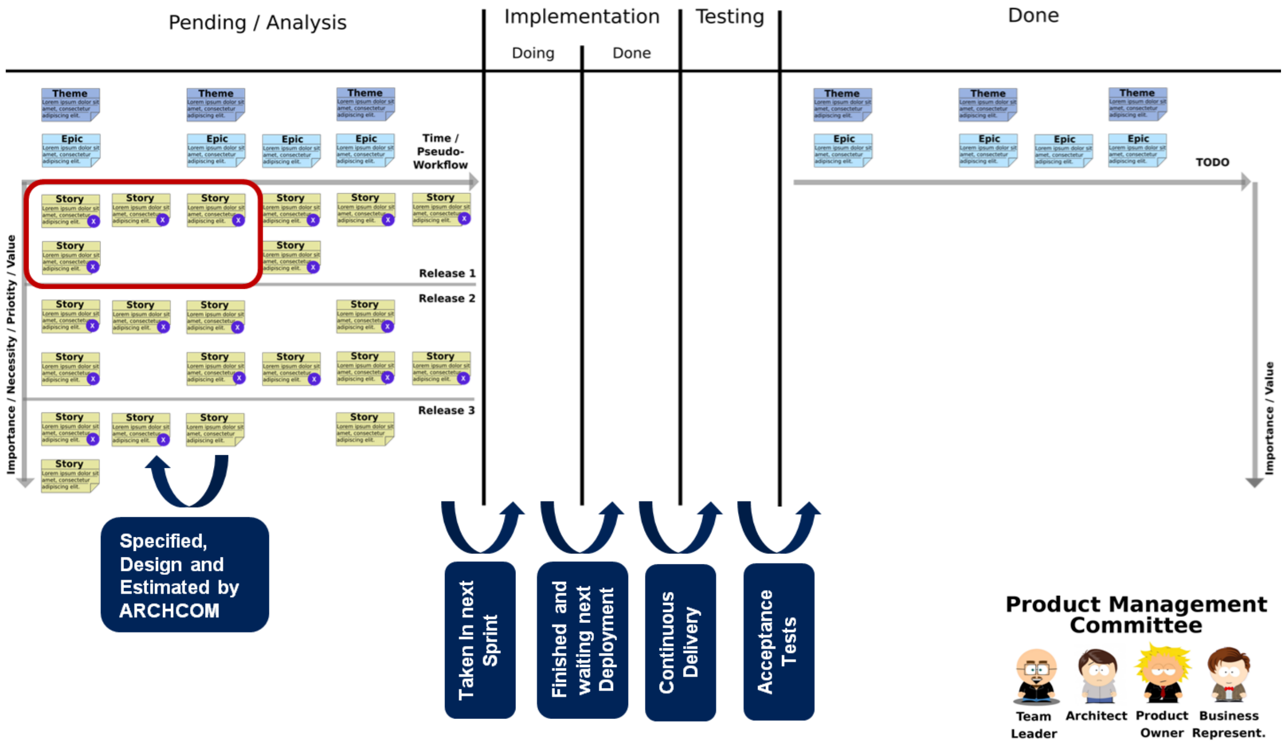
3.3.3 Product Kanban Board Maintenance Process

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**

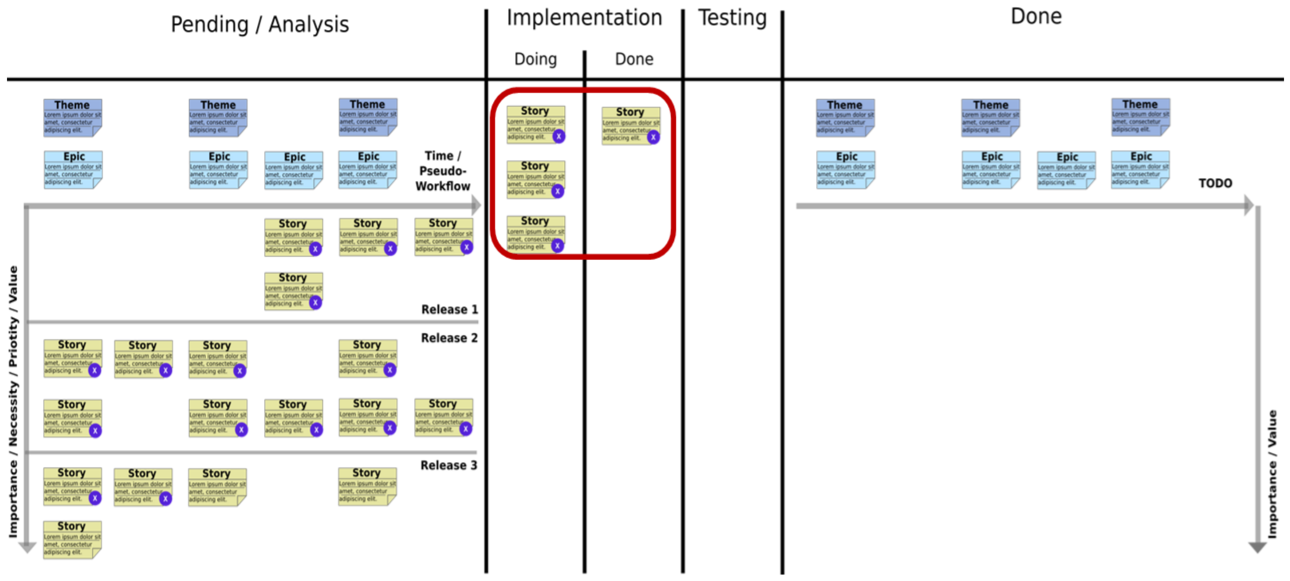
The Story Map on the left is a pretty standard Story Map, where releases are identified.

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.

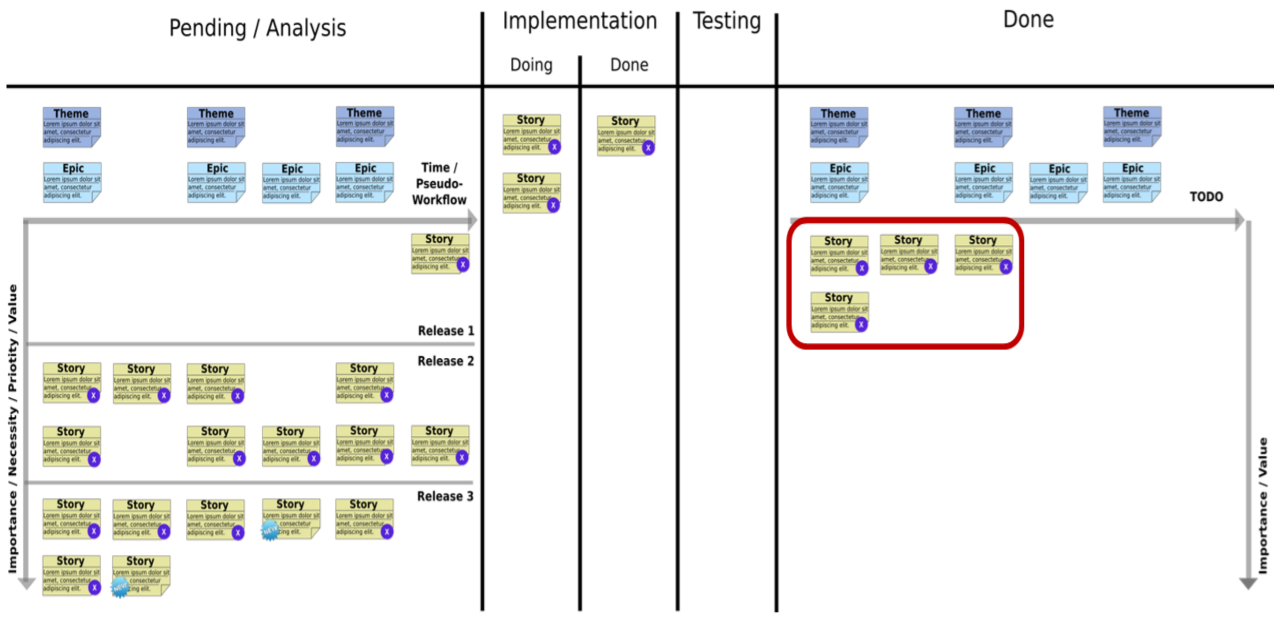
B. During the first sprint

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:



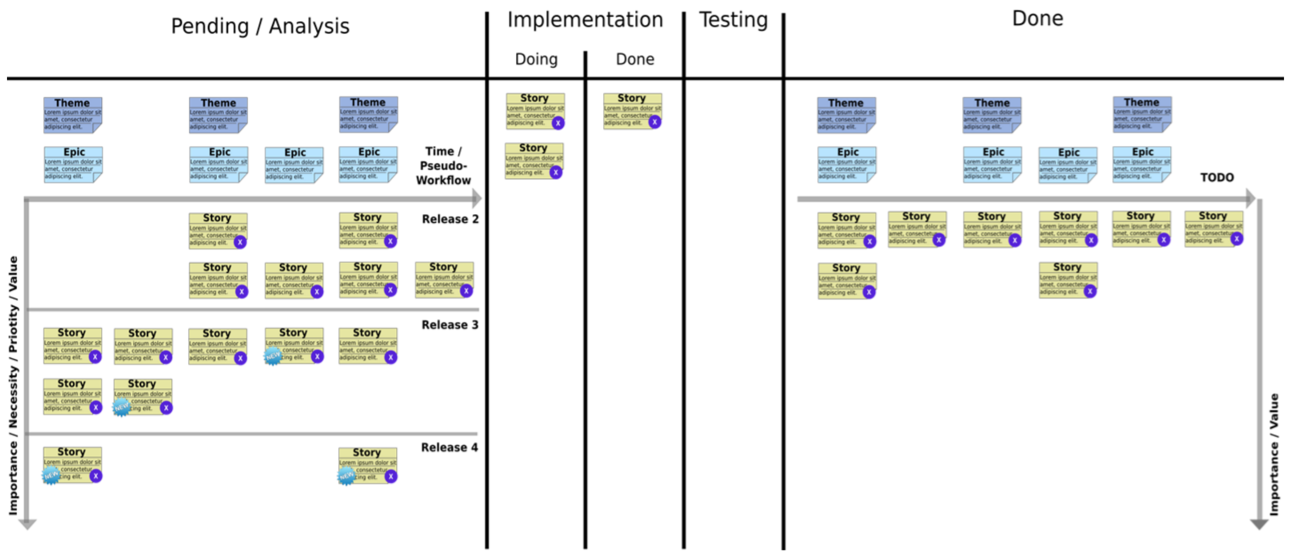
C. During the second sprint

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.



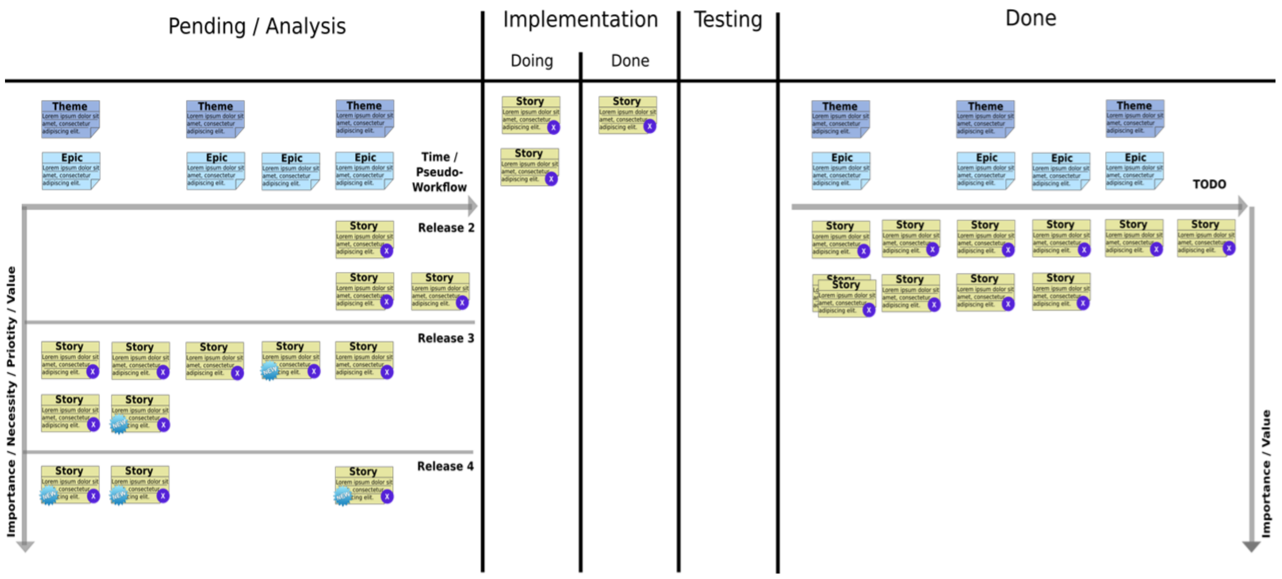
Notes:

- 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 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.

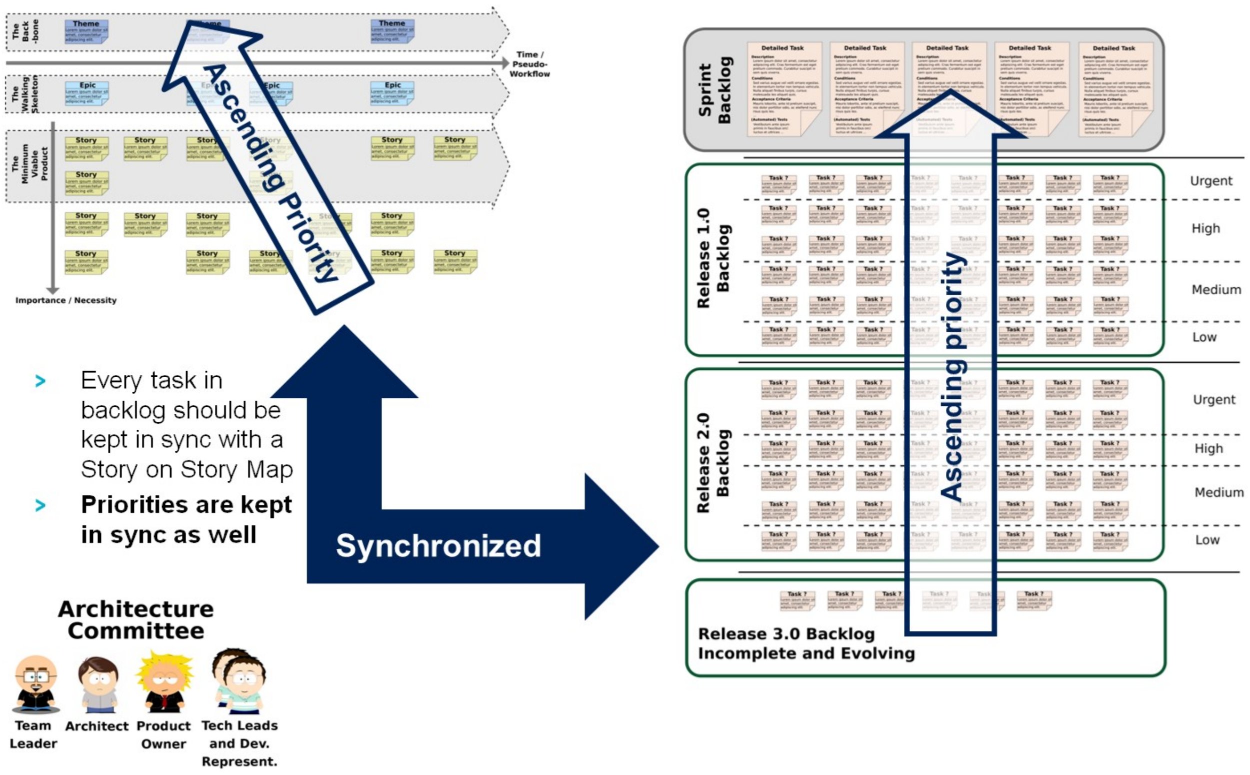


Also, User stories on the right may need to be merged whenever they relate to the same feature.

3.3.4 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.

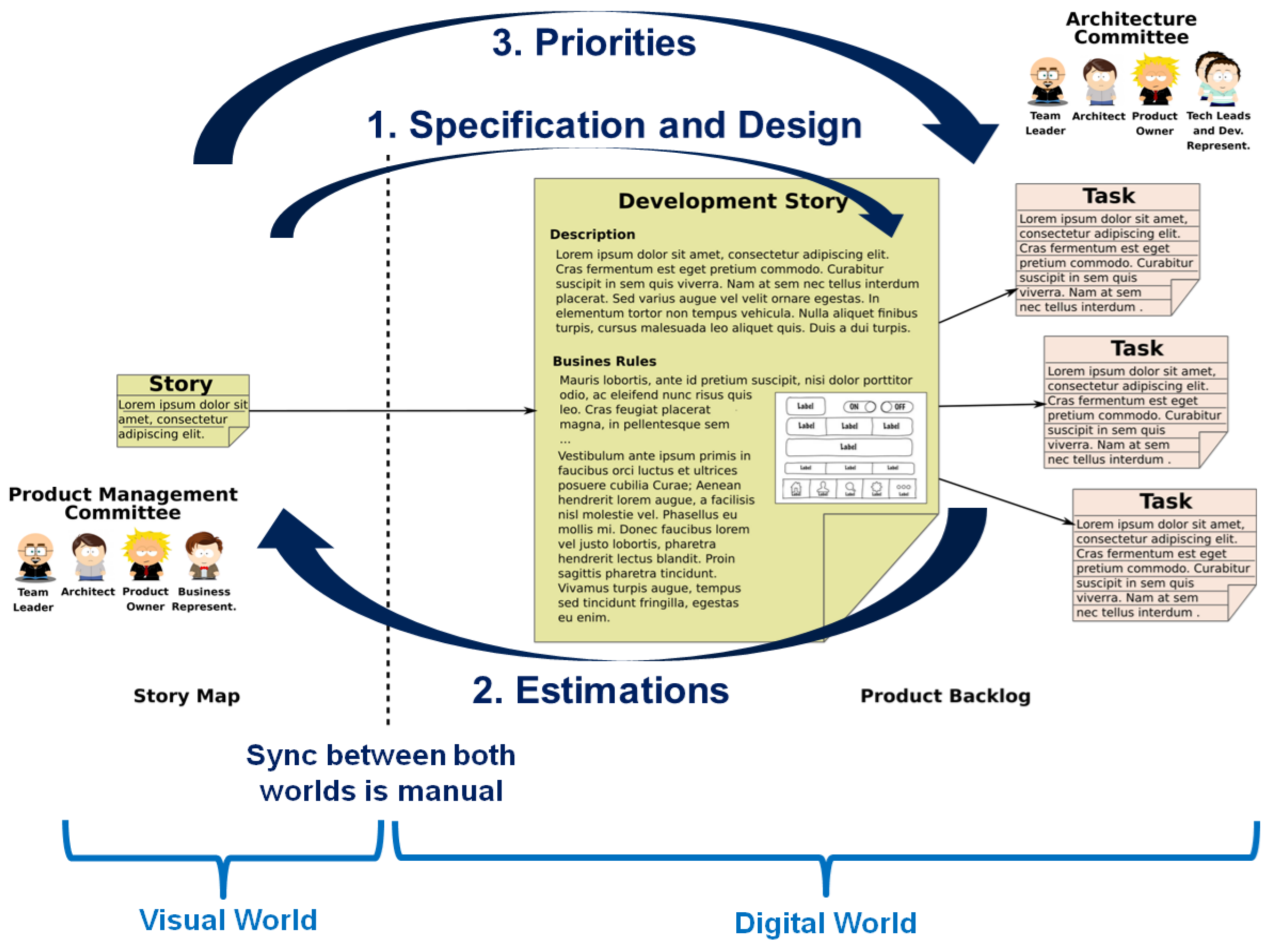
The principle is as follows:



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.

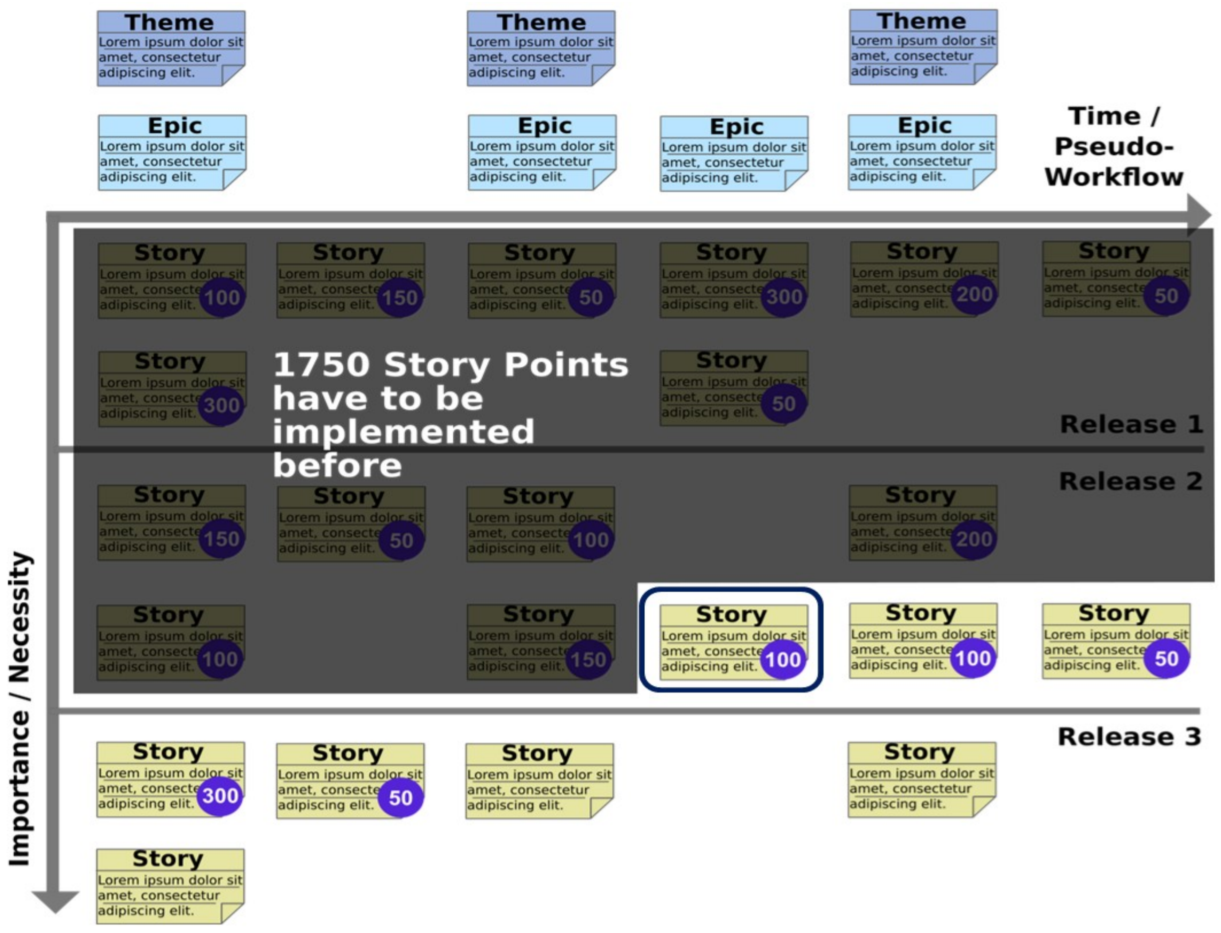
3.3.5 Forecasting

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 3.3.2 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.

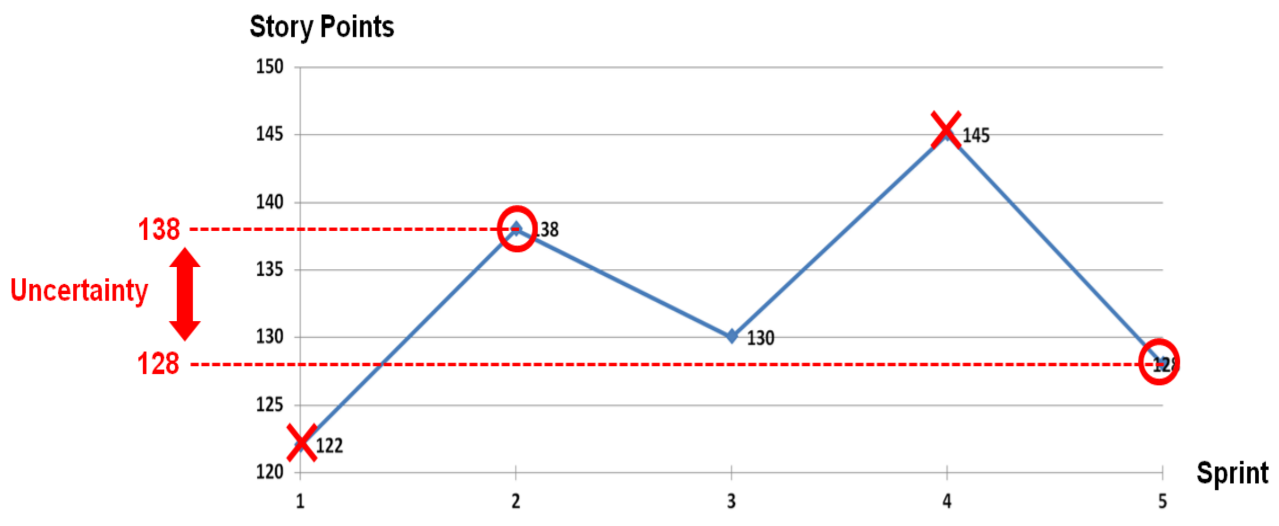
Base on this, we know that this story will be **delivered once all the stories before it will be implemented plus this story as well**, hence 1750 + 100 SP = 1850 SP.

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.

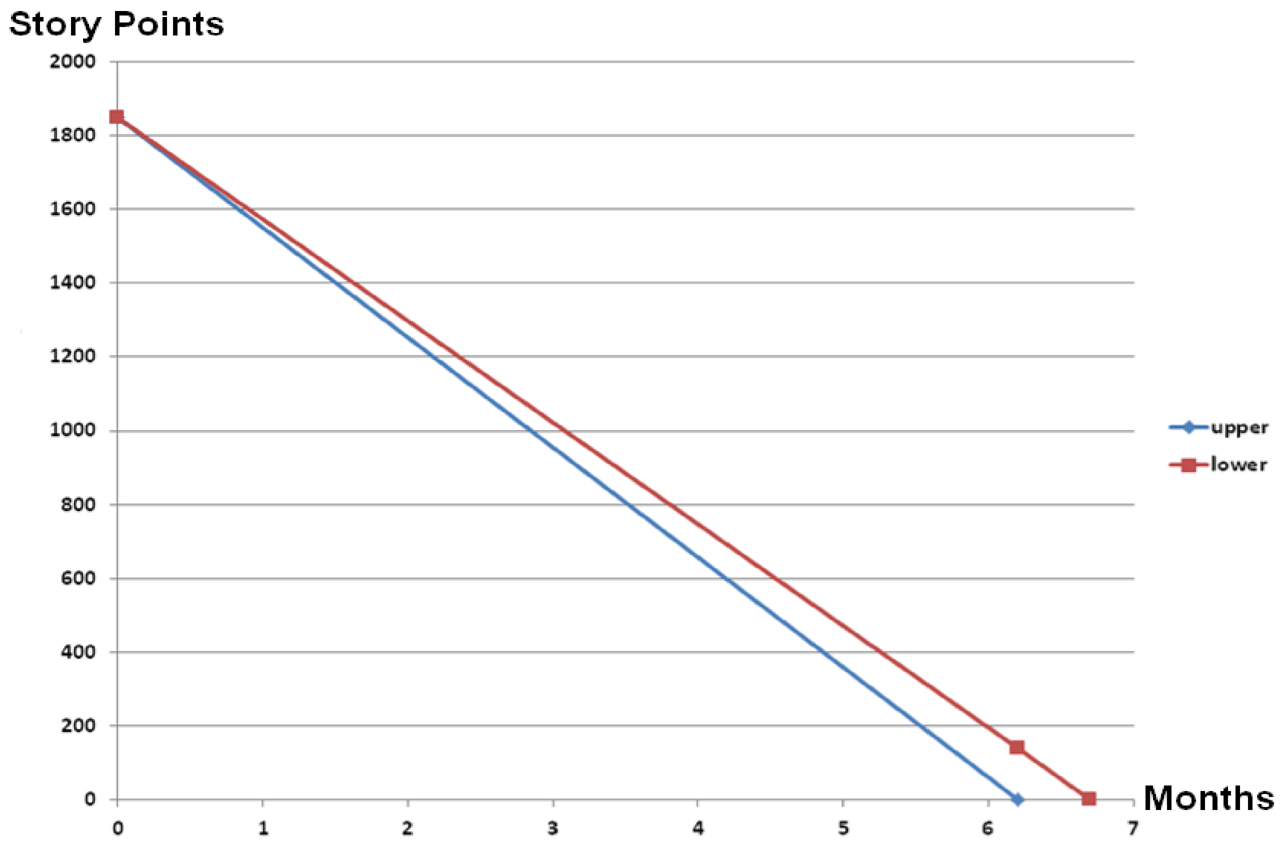
We use this range, and not a single value of average or median, to address a fundamental aspect of software engineering: the uncertainty.
The range gives us a lower value and an upper value which we will use as follows.

- **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.

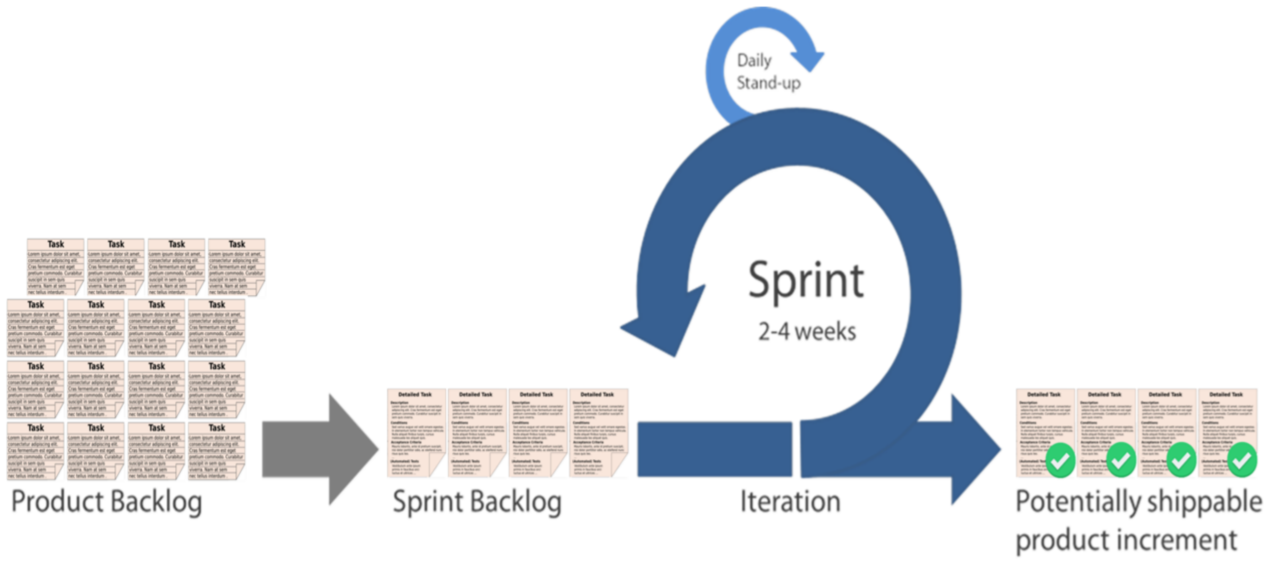


- Using the lower limit of 128 SP per sprint, it would take us 15 sprints to complete the scope, hence 30 weeks or 6.7 months
- Using the upper limit of 138 SP per sprint, it would take us 14 sprints to complete the scope, hence 28 weeks or 6.2 months

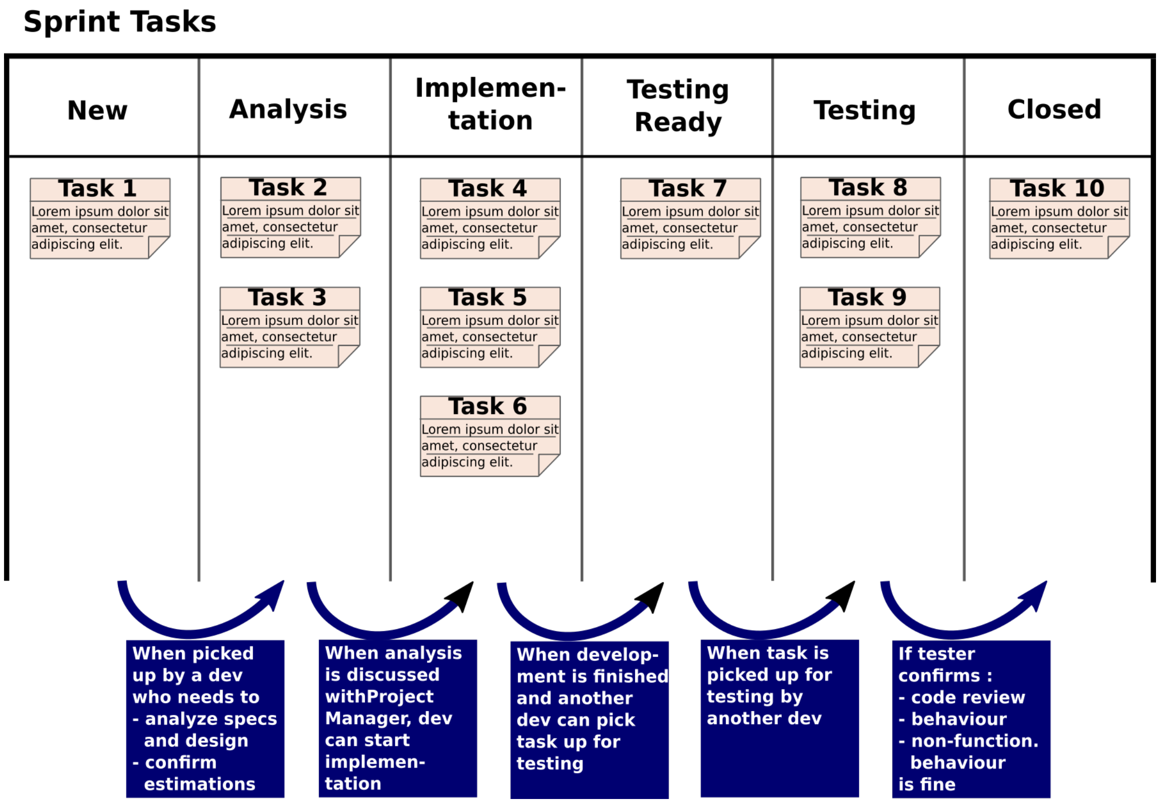
Based on this, the PMC or the Product Owner can communicate to the stakeholders that the feature would be release not before 6 months but before 7 months.

3.3.6 Development process: Scrum

I said enough about Scrum in both this paper and my previous article.
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**:


Sprint Tasks
New
Analysis
Implemen-tation
Testing Ready
Testing
Closed
Task 1 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Task 2 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Task 3 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Task 4 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Task 5 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Task 6 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Task 7 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Task 8 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Task 9 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Task 10 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
When picked up by a dev who needs to - analyze specs and design - confirm estimations
When analysis is discussed withProject Manager, dev can start implemen-tation
When develop-ment is finished and another dev can pick task up for testing
When task is picked up for testing by another dev
If tester confirms : - code review - behaviour - non-function. behaviour is fine

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.

3.4 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

3.4.1 Product Management Committee

Product Management Committee
Team Leader
Architect
Product Owner
Business Represent.

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:

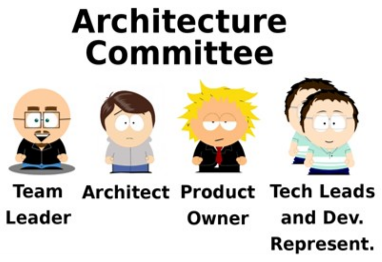
Story Mapping

- 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

3.4.2 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

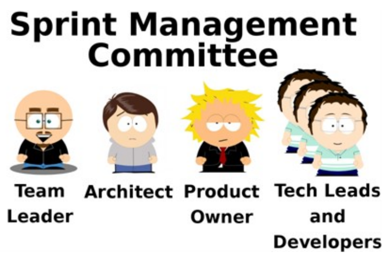
Estimation of User Stories

- 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

3.4.3 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

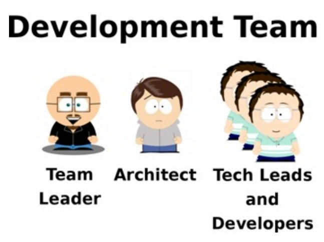
Sprint Retro

- 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.

3.4.4 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

3.5 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.

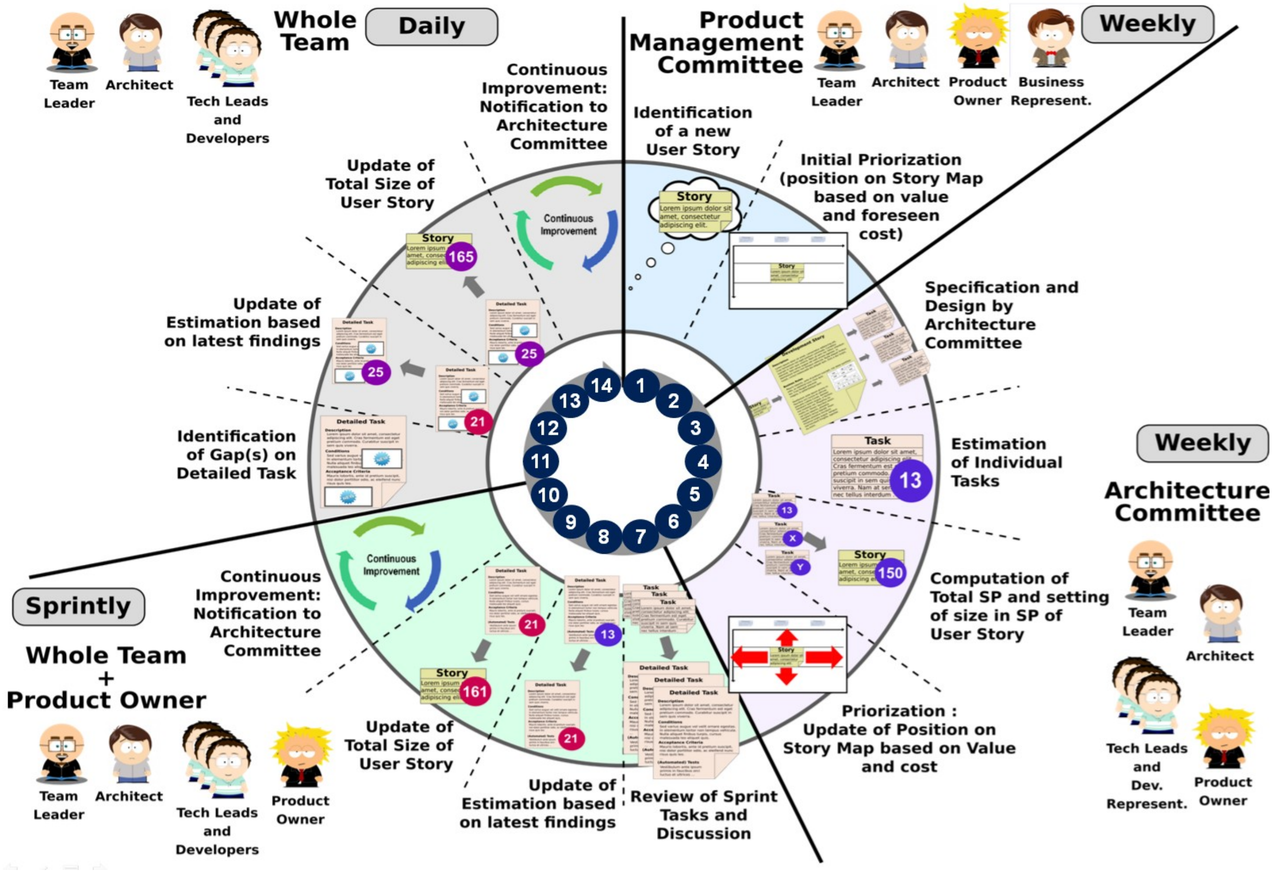
Finally, discipline and courage are enforced by a strict definition of the processes and rituals and a proper maintenance of this definition as the culture and practices evolve.
At the end of the day, defining these committees and rituals is all about that. Why are all these committees / teams / rituals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

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:

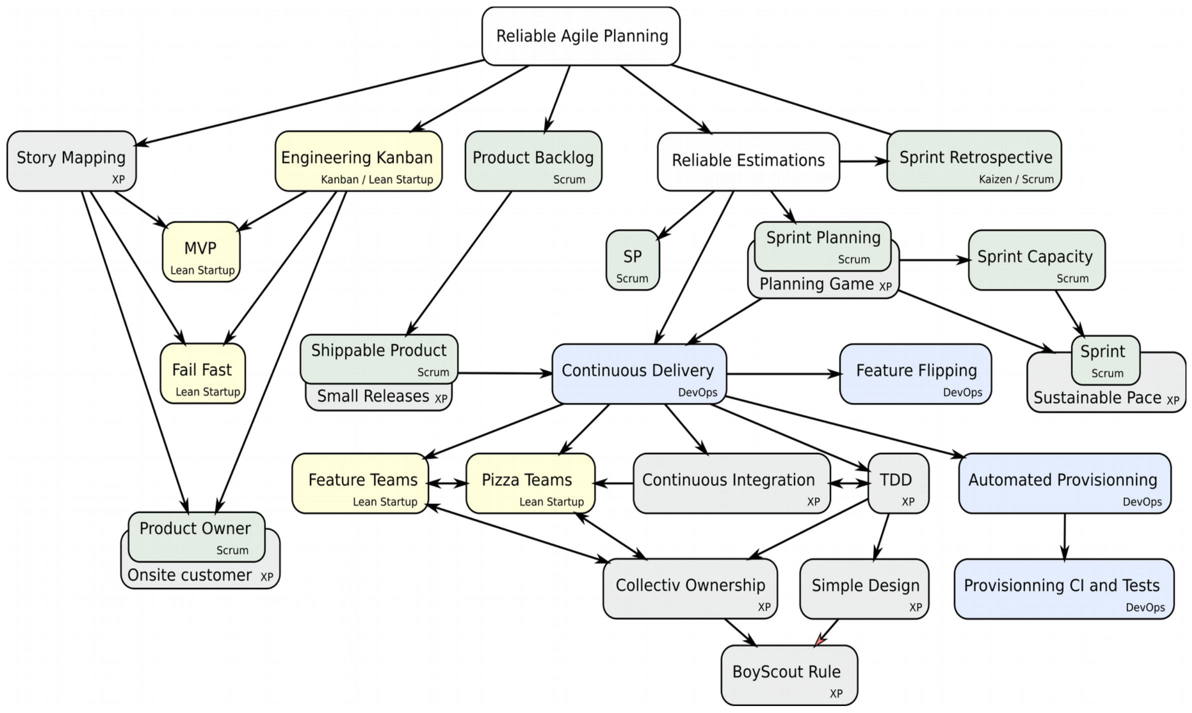

Whole Team
Daily
Team Leader
Architect
Tech Leads and Developers
Product Management Committee
Weekly
Team Leader
Architect
Product Owner
Business Represent.
Continuous Improvement: Notification to Architecture Committee
Identification of a new User Story
Initial Priorization (position on Story Map based on value and foreseen cost)
Update of Total Size of User Story
Specification and Design by Architecture Committee
Update of Estimation based on latest findings
Estimation of Individual Tasks
Weekly
Architecture Committee
Identification of Gap(s) on Detailed Task
Computation of Total SP and setting of size in SP of User Story
Team Leader
Architect
Tech Leads and Dev. Represent.
Product Owner
Sprintly
Whole Team + Product Owner
Continuous Improvement: Notification to Architecture Committee
Priorization : Update of Position on Story Map based on Value and cost
Update of Total Size of User Story
Update of Estimation based on latest findings
Review of Sprint Tasks and Discussion
Team Leader
Architect
Tech Leads and Developers
Product Owner
Continuous Improvement
Story
Task
Detailed Task
1
2
3
4
5
6
7
8
9
10
11
12
13
14
165
150
161

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.

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.