RESUMEN INGENIERIA Y CALIDAD DE SOFTWARE

(Basado en clases de Covaro)

¿QUÉ ES EL SOFTWARE?

El software no es solo código, se tiende hacer una mala asociación entre que el software es solo el programa, pero es importante incorporar que el software es un set de programas y la documentación que acompaña.

Saber programar nada más, no alcanza para hacer software.

TIPOS BÁSICOS DE SOFTWARE

SYSTEM SOFTWARE	O software de sistemas, este tipo de software se encarga de controlar y gestionar los recursos del hardware de una computadora, así como de proporcionar una interfaz para que los usuarios interactúen
UTILITARIOS	Se refiere a un subconjunto de software de sistema que proporciona utilidades o herramientas adicionales para mejorar la funcionalidad del sistema operativo y el rendimiento de la computadora. Su función principal es mejorar y optimizar el rendimiento del sistema operativo y del hardware de la computadora.
SOFTWARE DE APLICACIÓN	Este software se utiliza para realizar tareas específicas en una computadora, como procesar texto, crear presentaciones o navegar por internet. Se enfoca en satisfacer las necesidades de los usuarios.

El **software de aplicación** es en general con el software que vamos a trabajar.

SOFTWARE ≠ MANUFRACTURA

Nos interesa destacar que no hay manera de comparar software con una línea de producción. Por distintos motivos, pero podemos destacar 5 razones principales:

- El software es **menos predecible**, no como los productos tangibles en una línea de producción
- Casi **ningún producto de software es igual a otro**, ya que de alguna manera los requerimientos o necesidades de los usuarios que van a hacer uso de ese software van a ser distintas.
- **No todas las fallas en software son errores**. El tratamiento de errores en software es totalmente distinto en software que en la producción.
- **El software no se gasta**. Cuando hablamos de productos, normalmente, hablamos de tiempo, que al pasar del tiempo el producto se desgasta, esto no sucede con el software. Si bien el mismo debe ir adaptándose a los cambios y las necesidades del usuario/organización, específicamente no tiene la cualidad de desgastarse.
- **El software (obviamente) no está gobernado por las leyes de la física.**

Evidentemente todas las diferencias están ligadas a la **intangibilidad** del software como producto y además de estas 5 razones podemos dar muchísimas más.

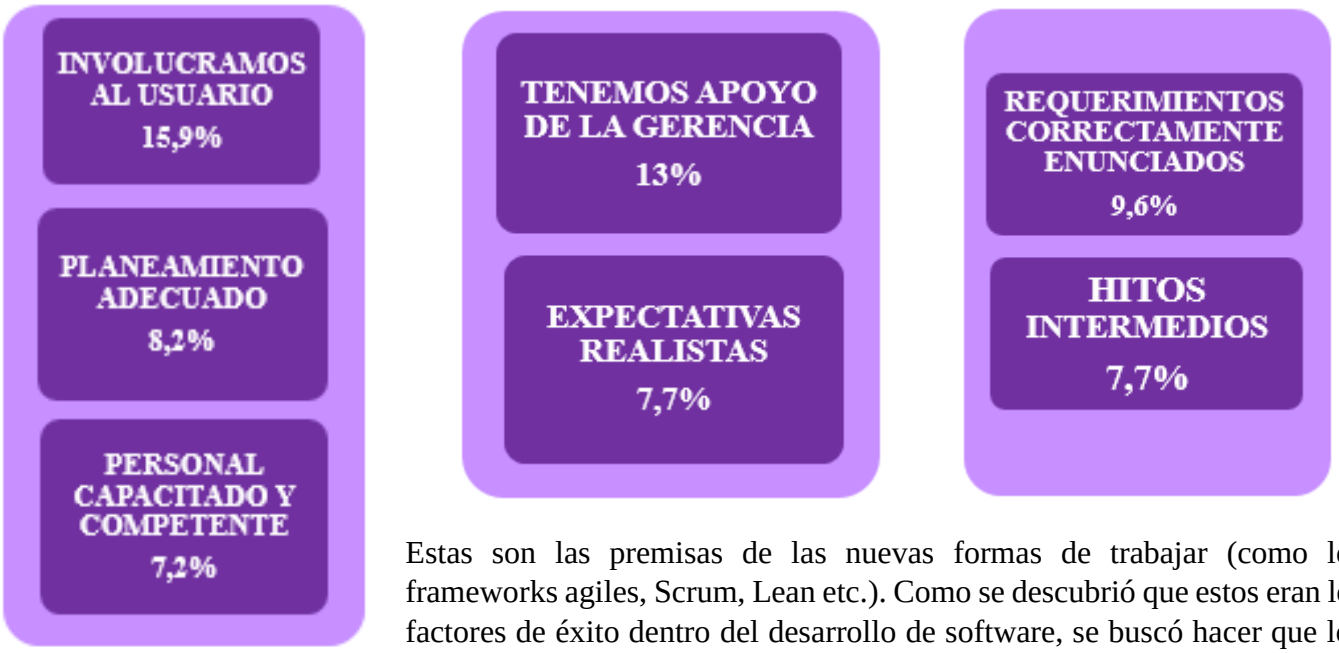
PROBLEMAS AL DESARROLLAR/CONSTRUIR SOFTWARE

- **MÁS TIEMPOS Y COSTOS QUE LOS PRESUPUESTADOS**. Este es el problema más repetido a la hora de desarrollar software.
- **QUE LA VERSIÓN FINAL DEL PRODUCTO NO SATISFAGA LAS NECESIDADES DEL CLIENTE.**

- **PROBLEMA EN LA ESCALABILIDAD Y/O ADAPTABILIDAD DEL SOFTWARE**. Agregar una funcionalidad en otra versión es casi una misión imposible.
- **MALA DOCUMENTACIÓN**, desactualizada y que no acompaña al software y por ende trae inconsistencias en la gestión o mantenimiento del mismo.
- **MALA CALIDAD DEL SOFTWARE**. Muchas veces el software de calidad se cree relacionado con la cantidad de testing que realizo sobre él, pero esto no es asi nemo.

SOFTWARE EXITOSO

El software termina siendo exitoso y redituable cuando:



Estas son las premisas de las nuevas formas de trabajar (como los frameworks agiles, Scrum, Lean etc.). Como se descubrió que estos eran los factores de éxito dentro del desarrollo de software, se buscó hacer que los mismos quedaran embebidos dentro del proceso de desarrollo.

Uno de los aspectos claves tiene que ver con los requerimientos, no solo con dejar los requerimientos claros, sino también incentivar a que el usuario tenga participación para esclarecer los mismos. Esto tiene que ver con la necesidad de que los requerimientos vayas “madurando” a medida que avanzamos en el desarrollo del sistema.

También debemos tener en cuenta la retroalimentación, es sumamente importante ir midiendo el avance del sistema y contar con información para poder corregir fallas o errores en el sistema.

El manifiesto ágil y metodologías como Scrum nos hablan sobre equipos capacitados y con habilidades cuya participación dentro del desarrollo del sistema sea competente.

SOFTWARE NO EXITOSO

Podríamos decir que el software no es exitoso o resulta en costos más elevados cuando hacemos lo contrario a lo anterior. Cuando tenemos requerimientos incompletos (13.1%) o requerimientos cambiantes (8,1%), cuando hay poco involucramiento del usuario (12.4%), cuando no tenemos los recursos suficientes (10,6%), cuando ponemos expectativas poco realistas y somos demasiado optimistas (9,3%) y cuando no tenemos suficiente apoyo por parte de la gerencia (8,7%).

En conclusión y habiendo analizado estos aspectos, confirmamos que no es suficiente saber programar para hacer software, sino que tenemos muchas aristas y características por cumplir para lograr un software exitoso.

EL PROCESO DE SOFTWARE

Un **proceso de software** se define como un conjunto estructurado de actividades que, a raíz de un conjunto de entradas, producen una salida. Estas actividades varían dependiendo de la organización y el tipo de sistema que debe desarrollarse. El proceso debe ser explícitamente modelado si va a ser administrado y llevado a cabo.

Hay diferentes percepciones de lo que es un proceso de desarrollo de software, pero el objetivo siempre va a ser obtener un producto o servicio de software a partir de distintos inputs. No solo vamos a tener como entrada o input los requerimientos (que nos van a definir o darle los límites al alcance de nuestro producto) sino que también vamos a tener a las personas (que son el principal capital/recurso que se consume en el desarrollo de software) y a su vez materiales, energía, equipamiento, librerías, hardware, etc.

Nuestro objeto es a raíz de todo este input que está colaborando generar un producto de software que aporte valor.

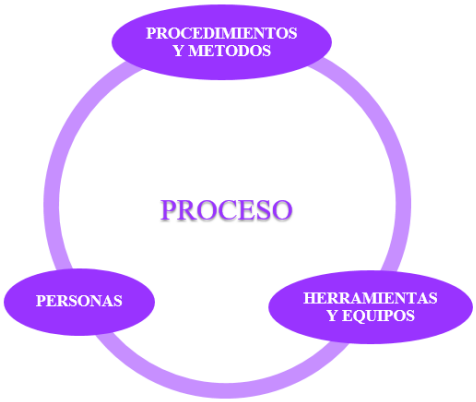
- **IEEE**: Un **proceso** es una secuencia de pasos ejecutados para un propósito dado.

Esta definición no es lo suficientemente completa y clara, es general y poco precisa por lo que nos basamos un poco más en la de CMM.

- **CMM**: Un proceso de software es un conjunto de actividades, métodos, prácticas y transformaciones que la gente usa para desarrollar o mantener software y sus productos asociados.

Dentro de esta definición tenemos más conceptos para vincular:

1. **Procedimientos y métodos**: Definición de cómo vamos a llevar a cabo las actividades de desarrollo
2. **Personas**: con habilidades, entrenamiento y motivación
3. **Herramientas y equipos**



Las personas hacen uso de las herramientas, equipos, procedimientos y métodos y a partir de estos producen un producto de software. Dentro de este proceso están definidas las responsabilidades, actividades y herramientas a partir de las cuales las personas transforman los requerimientos en software.

Al proceso de desarrollo de software lo define la organización en función de los objetivos y de las características del software.

Para producir software debo tener conocimiento en

PROCESO DEFINIDO

Se basan en una concepción basada en el modelo industrial (inspirados en líneas de producción), en la cual tengo un conjunto de inputs que van a ser utilizados en un conjunto dado de actividades, que van a producir el mismo resultado o salida siempre que los inputs sean los mismos.
Esto es difícilmente aplicable al software, ya que no es tan definible el software.
No podemos simplificar el software a una entrada de inputs que siendo utilizado para determinadas aplicaciones produce el mismo output, es decir que el output es predecible.
Asume que podemos repetir el proceso indefinidamente y obtener los mismos resultados.
La administración y el control provienen de la predictibilidad del proceso definido.

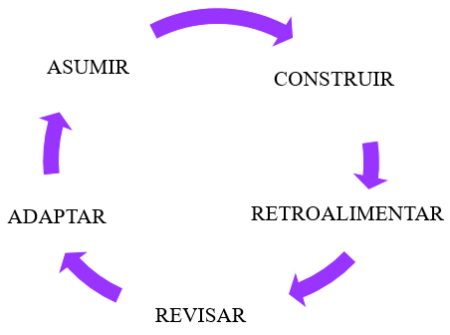
Muchas corrientes y procesos confían en estas premisas para llevar a cabo un proyecto de software.

Ej: PUD

PROCESO EMPIRICO

Viene del empirismo, es aquella corriente en la cual centramos el foco en la experiencia. (Más adelante sus pilares)
No solo la experiencia del usuario sino también la experiencia en el negocio, la experiencia técnica, en cómo voy a manipular las herramientas y aprender de los errores para poder aplicarlo.
Nos basamos en capitalizar la experiencia de los actores involucrados en la producción para maximizar la calidad de lo que estamos produciendo.
Vamos a tomarlos en contraste con los procesos definidos
Confiamos en que las variables no pueden ser definidas o controladas en su totalidad, estamos dentro de un ambiente complejo donde quizás no todo lo que está en juego está bajo nuestro control y esto nos implica la necesidad de **adaptación** para poder garantizar la producción de un software de calidad.
Se basan en la experiencia y el aprendizaje, y la principal diferencia con los procesos definidos, es la forma en la cual se puede mejorar el proceso.
Mientras que en el proceso definido el punto de mejora es el central (las actividades/etapas intermedias, ya que mis entradas y salidas son fijas), el proceso empírico no tiene un punto de mejora tan tangible, voy a tener que ir trabajando sobre distintos aspectos, siempre centrándome en las personas que son la clave del empirismo, son quienes capitalizan la experiencia.
Esta capitalización de la que hablamos se hace en un ciclo de retroalimentación que permite la adaptación y mejora, **aprendemos de la experiencia.**

PATRON DE CONOCIMIENTO DE PROCESOS EMPIRICOS



Empezamos con una **hipótesis** (**ASUMIR**), empezamos la etapa de construcción(**CONSTRUIR**) y en base a algunos puntos que podemos inspeccionar y analizar tomamos mediciones, sacamos información y haciendo retrospectiva (**RETROALIMENTAR**) revisamos (**REVISAR**) el resultado de la construcción y contrastando la hipótesis inicial con el resultado del proceso, podemos ir adaptando el proceso y seguir construyéndolo(**ADAPTAR**).
Esto se basa en una concepción que necesita repetición, es por esto que dentro de los procesos empíricos damos uso a un **ciclo de vida** específico (iterativo e incremental, más en detalle en la otra hoja)

CICLOS DE VIDA

El ciclo de vida es una abstracción. Me va a definir cuáles son las **etapas** (fases) y el **orden** de cada una de ellas. No me dice que tengo que hacer, ni quien lo va a hacer, ni las herramientas ni los procedimientos sino solo las fases y en qué orden se van a ejecutar.

Son la serie de pasos a través de los cuales un producto o un proceso progresa. Esto no dice que tanto los productos como los proyectos tienen ciclos de vida.

CICLOS DE VIDA DE PROYECTOS DE SOFTWARE

El ciclo de vida de un proyecto de software es una representación de un proceso. Grafica una descripción del proceso desde una perspectiva particular. Como dijimos anteriormente, el ciclo de vida de un proyecto nos define las fases del proceso y el orden en el cual se llevan a cabo las mismas.
Podemos decir que el ciclo de vida de un proyecto empieza y termina.

CICLOS DE VIDA DE PRODUCTO DE SOFTWARE

El ciclo de vida de un proyecto termina cuando genera un producto, cuyo ciclo de vida no va a terminar, sino que va a mantenerse mientras el objeto exista. ¿Cómo es esto? El producto va a necesitar mantenimiento una vez que "termino" de desarrollarse, y se va a mantener bajo este proceso de mantenimiento hasta que se deseche, ahí podemos decir que el ciclo de vida del producto llega a su fin.

Hay tres ciclos de vida básicos:

- SECUENCIAL: Se basan en ejecutar una etapa después de otra sin retorno generalmente. Hay algunas que pueden llegar a devolver información (como los ciclos de vida en cascada)

- ITERATIVO/INCREMENTAL: Los **procesos empíricos** en su mayoría implementan ciclos de vida iterativos e incrementales para poder dar la característica de adaptación y mejora del proceso. Esto permite que en cada una de las iteraciones capitalicemos lo aprendido, mejoremos la experiencia del equipo y realicemos otra iteración para incrementar el producto.
  Hago iteraciones para ganar información y convertir esa información en un incremento que será incorporado en el sistema.

- RECURSIVO: Se utiliza para casos particulares como proyectos de alto riesgo. Dentro de este tipo de ciclos tenemos el ciclo de vida de espiral, que se basa en la investigación de riesgos y el ciclo de vida en B que permite hacer una vuelta en particular de cada una de las etapas e ir avanzando en los incrementos.
  Se basa en tomar una característica específica y en una iteración me centro en esa funcionalidad, en la siguiente en otra, y asi.

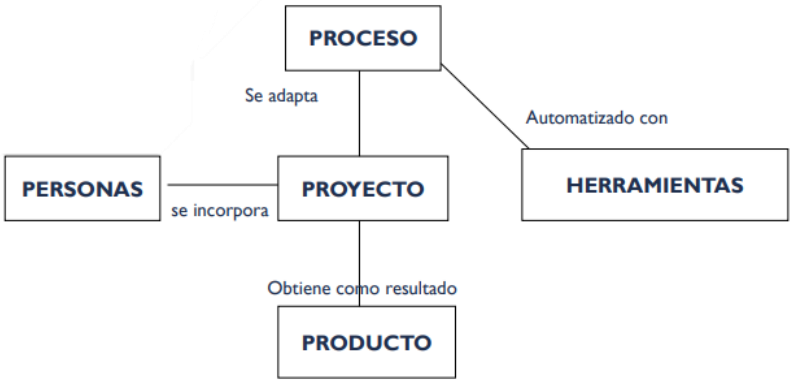
Están en desuso ya que se ha comprobado que no son útiles y no funcionan

Según las características del proyecto y del producto que se va a construir se va a elegir el ciclo de vida que se va a implementar para poder llevarlo a cabo de la mejor forma posible. Siempre va a depender de que beneficios tiene el mismo según costos asociados y el entorno en el que nos encontramos.

El **PROCESO** es una implementación del **CICLO DE VIDA** (que es una abstracción que nos guía en cuáles son las etapas y su orden) que tiene un objetivo que lo guía, que es la creación de un **PRODUCTO** (debe ser no ambiguo y alcanzable)

COMPONENTES DE PROYECTO DE DESARROLLO DE SOFTWARE

Se suele decir que el proyecto es una instanciación del proceso definida en un ciclo de vida.



UN **PROYECTO** ES LLEVADO A CABO POR **PERSONAS** QUE IMPLEMENTAN **HERRAMIENTAS** PARA AUTOMATIZAR LOS PROCESOS Y QUE TIENE OBTIENE COMO RESULTADO UN **PRODUCTO.**

El **proceso** es una plantilla, una definición abstracta que se materializa a través de los **proyectos**, en donde se adapta a las necesidades concretas del mismo. En un proceso se expresa en forma teórica lo que se debe hacer para hacer software y debo adaptarlo al proyecto (en caso de ser un proceso definido) o terminar de armarlo (en caso de ser un proceso empírico). Los procesos toman como entrada requerimientos y a través de un conjunto de actividades obtienen como salida un producto o servicio de software, esas actividades están estructuradas y guiadas por un objetivo.

El **proyecto** es la unidad organizativa/ de gestión, y necesita de personas, recursos y procesos para poder existir. A través del mismo, administro los recursos que necesito y las personas que van a formar parte del mismo, para obtener como resultado un producto de software.

Un proyecto de software tiene que cumplir con ciertas características:

- **DURACIÓN LIMITADA:** son temporarios, cuando se alcanza el/los objetivo/s, el proyecto termina. El proyecto tiene un inicio y un fin.
- **ORIENTADO A UN OBJETIVO:** (esto me permite que los proyectos sean únicos, para ello el objetivo tiene que ser claro, no ambiguo y alcanzable), tengo que poder definir hacia donde voy y poder establecer/medir cuando alcancé ese objetivo para poder dar por finalizado el proyecto. Los proyectos tienen un objetivo único, es decir que cada resultado de un proyecto, es distinto del resultado de otro proyecto.
- **SON ÚNICOS** todos los proyectos por similares que sean tienen características que los hacen únicos. Cada resultado de un proyecto es diferente al resultado de otro proyecto.
- **TIENE TAREAS RELACIONADAS ENTRE SI BASADAS EN ESFUERZOS Y RECURSOS PARA ALCANZAR EL OBJETIVO.** Las tareas dependen una de otra, tienen precedencia y a su vez tienen asociados/designados recursos y esfuerzos, lo cual hace que la gestión de proyectos sea una tarea compleja.
  La definición de que tareas hay que realizar se obtienen del proceso.

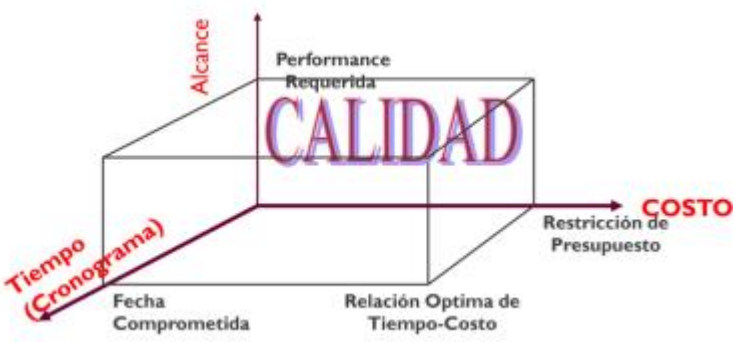
Cuando trabajamos en un proyecto de un **proceso definido** lo llamamos:

GESTIÓN TRADICIONAL DE UN PROYECTO

¿Qué es? Se basa en ejecutar las actividades definidas (con todo lo que necesito) para lograr el objetivo.

La **ADMINISTRACIÓN DE PROYECTOS (Project Managment)** en términos tradicionales, es la aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades del proyecto para satisfacer los requerimientos del proyecto y poder obtener como resultado el producto esperado.

LA TRIPLE RESTRICCIÓN



ALCANCE: Son los requerimientos que el cliente expresa, el objetivo, lo que quiero alcanzar.

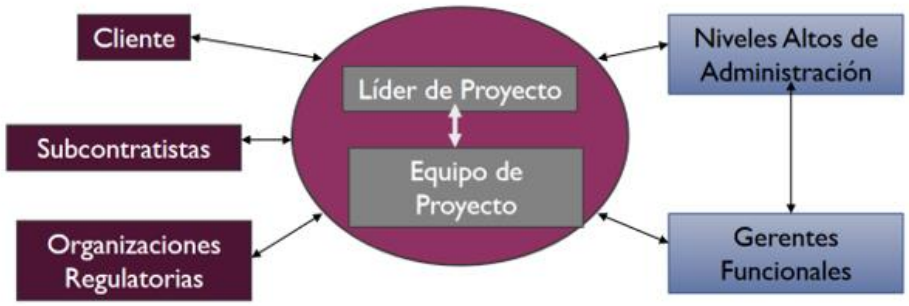
TIEMPO: Tiempo que debería llevar completar el proyecto.

COSTO: Costo en recursos y en dinero

La triple restricción es una de las bases de la gestión tradicional de proyectos. Plantea que en un proyecto vamos a tener suna **restricción de tiempo** (como un cronograma con una fecha específica), una **restricción de presupuesto** y un **alcance** (lo que voy a construir, relacionado con el objetivo que definí en términos del producto resultante). Es responsabilidad del **PM** balancear los tres objetivos que están involucrados en el proyecto y compiten entre sí dentro del mismo y definir el tiempo y el costo del proyecto dado el alcance del producto.

Estas tres variables están íntimamente relacionadas de manera tal que, si modifico una de ellas, voy a tener que hacer variar alguna de las otras o ambas. El balance de las mismas afecta directamente la calidad del proyecto, y debemos decir, que **la calidad del producto no es negociable**. Decimos que proyectos de alta calidad entregan el producto requerido o resultado, satisfaciendo los objetivos en el tiempo estipulado y con el presupuesto planificado

ROLES DENTRO DE UN PROYECTO



En la gestión tradicional nuestro equipo de proyecto está formado por un **EQUIPO** propiamente dicho y un **LÍDER DE PROYECTO (PM).**

LIDER DE PROYECTO - PROJECT MANAGER (PM)

Es quien se ocupa de todas las tareas de gestión que hacen que el equipo de proyecto sea guiado a lograr el objetivo. Es quien se relaciona con el ente que regula el avance del proyecto y el propio equipo, y con los demás involucrados en el proyecto. Entre otras tareas, es el encargado de administrar todos los recursos, organizar el trabajo de las personas involucradas y hacer el seguimiento de la planificación verificando que todo vaya según lo planeado.

Administrar un proyecto incluye poder definir los requerimientos, el alcance del producto, establecer objetivos claros y alcanzables y adaptar las especificaciones, planes y el enfoque a los diferentes intereses involucrados (stakeholders). Desde el principio el PM debe saber en términos de negocio o del producto de software qué tenemos que construir y qué es lo que más pesa o lo más importante dentro del proyecto, si cumplir con el tiempo, no superar el costo o no tocar el alcance.

En los procesos empíricos, el PM no es necesario ya que los proyectos se autogestionan. Es un rol propio de la gestión tradicional de proyectos.

EQUIPO DE PROYECTO

Se trata de un grupo de personas **comprometidas** en alcanzar un conjunto de objetivos de los cuales se sienten mutuamente responsables. Todo el equipo apunta a perseguir el mismo objetivo, que debe ser alcanzable y medible.

Para poder llegar a cumplir este objetivo, el equipo de proyecto debe poder **complementarse**, es decir, contar con distintos conocimientos y habilidades que sumen y hagan la diferencia entre el trabajo en equipo y un trabajo individual. A su vez deben poder trabajar en equipo sin problema y para esto usualmente se eligen grupos pequeños para mejorar y fomentar la comunicación.

Para llevar el proyecto adelante necesitamos lo que llamamos:

PLAN DE PROYECTO

Es como una hoja de ruta. Cuando nos sentamos frente a un proyecto debemos delinear un **plan de proyecto** que no va a estar completamente definido, pero va a esbozar que es lo que va a hacer el proyecto, el alcance, el objetivo, que recursos vamos a tener, en que tiempo lo vamos a desarrollar, cuales son las personas involucradas, sus roles, funciones, etc.

Lo primero que vamos a hacer entonces, es hacer un plan detallado que nos va a servir como guía para hacerle frente al proyecto.

A través del plan de proyecto **documentamos**:

- ¿QUÉ ES LO QUE HACEMOS? Alcance del proyecto
- ¿CUÁNDO LO HACEMOS? Calendarización
- ¿CÓMO LO HACEMOS? Recursos y decisiones disponibles
- ¿QUIÉN LO VA A HACER? Asignación de tareas

La idea del plan de proyecto es que sea algo **vivo** a lo largo del proyecto, que se vaya nutriendo y corrigiendo a medida que el proyecto se lleva a cabo. Nos va a servir siempre que esté en permanente actualización, ya que en el entorno de desarrollo del proyecto vamos a encontrarnos con variaciones que no estimamos al determinar el plan de proyecto debido a la incertidumbre inicial.

PLANIFICACIÓN DE PROYECTOS DE SOFTWARE

Cuando hablamos de planificar el proyecto de software incluimos:

1. **DEFINCIÓN DEL ALCANCE**: Es importarte hacer la diferenciación entre proyecto y producto de software.

ALCANCE DEL PROYECTO	ALCANCE DEL PRODUCTO
Es todo el **trabajo** y solo el trabajo que debe hacerse para entregar el producto o servicio con todas las características y funciones especificadas en el objetivo. El cumplimiento del alcance de proyecto se mide contra el Plan de Proyecto.	Son todas las características que pueden incluirse en un producto o servicio. El cumplimiento del alcance de producto se mide contra la Especificación de Requerimientos (Por ej.: CU)

La relación que hay entre ambos es que, si el producto que yo tengo que construir es grande, las tareas a definir en el proyecto van a ser más o van a requerir más tiempo y recursos, por lo tanto**, para definir el alcance del proyecto debo primero definir el alcance del producto.**

2. DEFINICIÓN DE PROCESO Y CICLO DE VIDA:

Cuando inicia el proyecto debo definir qué proceso quiero usar y que ciclo de vida voy a utilizar.

El **proceso de desarrollo** es el conjunto de actividades que necesito ejecutar para construir el producto de software (ej. PUD) y el **ciclo de vida** es de que manera ejecuto esas actividades.

Dentro de esta gestión tradicional y los procesos definidos, podemos elegir el proceso y cualquier ciclo de vida para llevar a cabo el proceso, en gestión ágil tenemos limitaciones (el único ciclo de vida que puede utilizarse es el iterativo).

3. ESTIMACIÓN:

Cuando vamos a hacer un plan de proyecto vamos a estimar todo, debemos definirlo antes de hacerlo, ya que no tenemos certeza de nada.

Tenemos que ver de qué manera podemos estimar todas las características necesarias para la planificación. Esto lo vamos a hacer con un cierto nivel de incertidumbre, pero vamos a intentar definir en orden las siguientes características:

- **TAMAÑO**: Definir el producto a construir
- **ESFUERZO**: Horas persona lineales (ideales)
- **CALENDARIO**: Determinar qué días y que horas se va a trabajar y cuantas personas van a trabajar.
- **COSTO**: Determinar un presupuesto necesario
- **RECURSOS CRÍTICOS:** Tanto humanos como físicos que nos van a hacer falta

Normalmente cuando hablamos de estimación se definen rangos también, no es de manera taxativa.

A medida que avanzo en el desarrollo del proyecto, la incertidumbre se va achicando y voy teniendo mayores certezas.

4. GESTIÓN DE RIESGOS:

Un riesgo es algo que podría suceder o no, pero si sucede puede comprometer el éxito del proyecto.

Lo que tenemos que hacer en primer lugar es listar/identificar los riesgos que son **más probables** de ocurrir o los que más **impacten** en el sistema, ya que es imposible gestionar TODOS los riesgos (son infinitos). Lo que hacemos es multiplicar estas dos variables de impacto y ocurrencia y obtenemos así lo que llamamos **exposición de riesgo**. Esta nos va a permitir determinar aquellos riesgos que vamos a tomar como principales y elegir cuales son los que voy a gestionar.

Una vez identificados lo que hacemos es gestionar los riesgos, esto es, generar acciones para disminuir el impacto o evitar/litigar la ocurrencia.

Hay algunos riesgos donde no podemos incidir sobre la probabilidad de ocurrencia o sobre el impacto, lo que debemos hacer es encontrar la manera de alivianarlos.

A medida que avancemos en el proyecto, los riesgos pueden aumentar, disminuir, ser más probables o aparecer nuevos (ya que todo lo que es plan de proyecto va modificándose y necesita constante actualización).

5. ASIGNACIÓN DE RECURSOS
6. PROGRAMACIÓN DE PROYECTOS

7. DEFINICIÓN DE MÉTRICAS:

Las métricas de software nos van a permitir darnos cuenta si nuestro proyecto está en línea con lo planificado o si esta desviado.

Como nos basamos en estimaciones para planificar, la métrica nos va a permitir saber (cuando el proyecto se está ejecutando) si estoy cumpliendo con lo planificado o no. Esto lo hago midiendo la realidad, lo que está pasando, y en base a eso saber si estamos acorde a lo planificado.

Si una métrica no tiene un fin particular, no tiene ningún sentido tomarla.

Las métricas pueden clasificarse de acuerdo a su dominio en:

MÉTRICAS DE PROCESO	Con proceso nos referimos al proceso de desarrollo de software definido por la organización. Nos permiten saber independientemente de un proyecto en particular, saber si como organización estamos trabajando bien o mal. Nos permite saber que pasa en términos organizacionales. Son básicamente las mismas métricas del proyecto, pero despersonalizadas de un proyecto en particular. **Ej.: Porcentaje de proyectos que terminan en termino y porcentaje de proyectos que terminan fuera de termino.**
MÉTRICAS DE PROYECTO	Son aquellas métricas que me van a permitir saber si un proyecto de software en ejecución se está cumpliendo de acuerdo a lo planificado o no. **Ej.: Comparación del tiempo calendario real con el planificado.**
MÉTRICAS DE PRODUCTO	Miden cuestiones/características que tienen una relación directa con el producto que estamos construyendo. **Ej.: Métricas de defectos, todas las métricas de tamaño**

MÉTRICAS BÁSICAS:

- TAMAÑO
- ESFUERZO
- TIEMPO(CALENDARIO)
- DEFECTOS

¿CADA CUÁNTO TOMO MÉTRICAS?

Uno de los aspectos clave de las métricas es **hacer foco**. Esto quiere decir que mis métricas van a ser tomadas en base a decisiones sobre el tiempo que va a tardar en realizarse el proyecto. Tengo que minimizar los esfuerzos que se ejecutan para tomar métricas.

8. MONITOREO Y CONTROL

El PM se va a encargar de trabajar sobre lo definido en el plan de proyecto y determinar si se está cumpliendo o no. Para hacer esto se va a basar en el plan de proyecto y en las métricas.

"Un proyecto se atrasa de a un día a la vez" – Fred Brooks, Mythical man months.
Esto quiere decir que, si puedo corregir los desvíos de mi proyecto en el momento adecuado, estoy a tiempo de poder cumplir mi objetivo**.**

Es súper normal en los proyectos comparar lo planificado y lo real. Como desarrolladores es muy común ser extremadamente optimistas al momento de hacer estimaciones, por lo que es totalmente necesario hacer monitoreo y control del desarrollo del proyecto.

FACTORES PARA EL ÉXITO	CAUSAS DE FRACASOS
1. **Monitoreo y Feedback**: tener un monitoreo permanente y generar acciones correctivas cuando sea necesario para evitar una desviación mayor. 2. **Misión/Objetivo claro**: saber hacia donde vamos 3. **Comunicación**: en todos sus aspectos, con el líder de proyecto, con el equipo, con los clientes y con los stakeholders.	1. Fallar al definir el problema 2. Planificar basado en datos insuficientes 3. La planificación la hizo el grupo de planificaciones 4. No hay seguimiento del plan del proyecto 5. Plan de proyecto pobre en detalles 6. Planificación de recursos inadecuada 7. Las estimaciones se basaron en “supuestos” sin consultar datos históricos 8. Nadie estaba a cargo 9. Mala comunicación

FILOSOFÍA ÁGIL

El a**gilísmo** o **filosofía ágil** es un movimiento que surgió de los desarrolladores aproximadamente 20 años cuando un grupo de referentes de la industria de software en un hotel de Utah acordaron un conjunto de enunciados a los que llamaron **Manifiesto Ágil.**

El manifiesto ágil se trata de un compromiso que hacen todas las personas involucradas en un proyecto para trabajar de una determinada manera independientemente de las prácticas que realice cada uno. Esto ha sido una evolución cultural que tiene que ver con las experiencias que cada uno de los referentes ha vivido.

El objetivo del **Movimiento Ágil** era lograr un equilibrio entre seguir procesos rigurosos y formales en el desarrollo de software y trabajar de manera eficiente y efectiva para entregar un producto de calidad. En otras palabras, evitar los extremos de la falta de profesionalismo y la excesiva burocratización, para lograr una entrega continua de software funcional y satisfacer las necesidades del cliente.

Todo lo que tiene que ver con Manifiesto Ágil se aplica en procesos empíricos.

Es un compromiso útil/ un punto medio entre nada de proceso y demasiado proceso.

También se suele llamar AGILE. No es una metodología ni un proceso, es una **ideología** que cuenta con un conjunto definido de principios que guían el desarrollo del producto. Busca encontrar un equilibrio entre procesos definidos y nada de procesos.

El manifiesto ágil se sustenta en los **procesos empíricos** que tienen como base la **experiencia**, y la misma sale del propio equipo, por ello es importante tener ciclos de realimentación cortos. Esto implica comenzar con una idea o requisito, construir un producto mínimo viable (MVP) y mostrarlo al cliente para obtener retroalimentación. A partir de esa retroalimentación, se realizan mejoras y correcciones continuas durante todo el proceso de desarrollo. Este enfoque es **compatible únicamente con ciclos de vida iterativos** y se basa en la idea de que la incertidumbre y los cambios son inevitables en el desarrollo de software, por lo que es necesario trabajar de manera flexible y adaptativa para satisfacer las necesidades del cliente.

Los procesos empíricos (como vimos antes) centran sus bases en el **empirismo.**

El empirismo tiene 3 pilares:

1. **TRANSPARENCIA**: Se refiere a la **comunicación abierta** y **honesta** de la información relevante a todas las partes interesadas.
   La transparencia promueve la confianza y la colaboración entre los miembros del equipo y las partes interesadas, lo que a su vez conduce a una toma de decisiones más informada y eficaz.
2. **ADAPTACIÓN**: Se refiere a la **capacidad de adaptarse** y **ajustar el trabajo** y el proceso para **hacer frente a las desviaciones** y **problemas** detectados durante la inspección.
   La adaptación permite una respuesta rápida y efectiva a los cambios y permite mantener la entrega de valor al cliente.
3. **INSPECCIÓN**: Se refiere a la **revisión regular** y **sistemática** del progreso del trabajo y los productos resultantes para **detectar problemas y desviaciones** del plan.
   La inspección permite una evaluación temprana y continua del progreso y ayuda a tomar decisiones informadas sobre los próximos pasos.

MANIFIESTO ÁGIL

El manifiesto ágil, tiene 4 valores a los que llamamos:

VALORES ÁGILES

01. Individuos e interacciones por sobre procesos y herramientas

Esto quiere decir que se enfatizan los vínculos, la comunicación efectiva y la colaboración entre los miembros del equipo y las partes interesadas, por sobre la adopción de la herramienta que tenemos que usar y el proceso a aplicar.

Las dos cosas son importantes sólo que los individuos e interacciones son más valorados.

02. Software funcionando por sobre documentación extensiva

Este valor enfatiza la importancia de proporcionar software funcional y de alta calidad en lugar de enfocarse en documentar cada detalle del proceso de desarrollo

Esto no quiere decir que NO se debe generar documentación, existe la necesidad de mantener la información del producto y del proyecto que estamos cursando para desarrollar el mismo.

El enfoque ágil plantea la idea de generar la información cuando haga falta. Debemos recordar, *"lo más importante es el acto de planificar no el resultado"*.

Las decisiones tomadas con respecto al producto de software deben quedar documentadas y van a trascender más allá de la iteración en la cual se generó el incremento al mismo. Lo que se debe decidir es: **qué aspectos del producto van a quedar documentados,** teniendo en cuenta que el conocimiento del producto debe ser transparente y de toda la organización no de un individuo**.**

Con respecto al proyecto, lo mismo, vamos a definir qué vamos a documentar del proyecto. Otro aspecto a tener en cuenta es la **permanencia**, no toda la información que se genera debe tener permanencia y disponibilidad infinita.

03. Colaboración con el cliente por sobre negociación contractual (o de contratos)

Este valor enfatiza la importancia de trabajar en colaboración con el cliente y comprender sus necesidades y requisitos en lugar de simplemente negociar contratos y acuerdo formales.

Está muy relacionado con la triple restricción.

Los problemas en las negociaciones contractuales se dan cuando el cliente firma un contrato con el equipo definiendo un alcance/costo/tiempo determinado y luego quiere realizar cambios en el mismo. Estos problemas se pueden evitar si se involucra al cliente de manera más activa en el proyecto y se fomenta una actitud de colaboración e integración. Es fundamental que el cliente **esté dispuesto a participar** y sumarse a este enfoque, ya que, de lo contrario, **la metodología Ágil no se puede aplicar**. Por lo tanto, no sólo es importante formar a los equipos que trabajarán con enfoque Ágil, sino también al cliente (quien en Scrum es el Product Owner).

04. Responder al cambio por sobre seguir un plan:

Este valor enfatiza la importancia de ser flexible y adaptativo en la respuesta a cambios y desviaciones en el proceso de desarrollo en lugar de seguir un plan rígido y predefinido que puede no ser el adecuado para el contexto actual.

Plantea que debemos construir en conjunto con el cliente, no definir tanto, empezar a trabajar con una idea del producto y después enfocarnos más a medida que avanzamos para darle la posibilidad al cliente de cambiar de opinión.

En lugar de tener una ERS firmada por el cliente (que seguro no leyó), mejor tener una actitud más ajustada con la realidad de que los requerimientos cambian, la gente se equivoca, se olvida, se da cuenta de lo que quiere cuando lo ve, y no por un contrato. En lugar de depender únicamente de un contrato, es mejor tener una comunicación frecuente y colaborativa con el cliente para adaptarse a los cambios y lograr una entrega exitosa del producto.

Cuando hablamos de **Requerimientos Ágiles**, debemos ponerlo en contexto, se trata de una manera de trabajar con los requerimientos que está alineada con los

12 PRINCIPIOS DEL MANIFIESTO ÁGIL

1. SATSIFACCIÓN DEL CLIENTE
Nuestra mayor prioridad es satisfacer al cliente a traves de la entrega temprana y continua de software con valor.

2. ADAPTACIÓN AL CAMBIO
Aceptamos que los requerimientos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.

3. ENTREGAS FRECUENTES
Entregamos software funcional con frecuencia, con preferencia a la documentación exhaustiva.

4. TRABAJO EN EQUIPO
Colaboramos con el cliente y los interesados durante todo el proyecto para asegurarnos de que el software entregado satisfaga las necesidades del negocio.

5. PERSONAS MOTIVADAS
Construimos proyectos en torno a individuos motivados. Les damos el entorno y el apoyo que necesitan y confiamos en que harán el trabajo.

6. COMUNICACION DIRECTA
El método más eficiente y efectivo de comunicar informacion dentro de un equipo de desarrollo de software es la conversación cara a cara. La efectividad y la riqueza de la comunicación crece exponencialmente conforme estamos en un mismo espacio físico (co-locados) y compartiendo un pizarrón para construir juntos el producto que necesitamos.

7. SOFTWARE FUNCIONANDO
El software funcionando es la medida principal de progreso

8. CONTINUIDAD
Los procesos ágiles promueven el desarrollo sostenible, es decir, el equipo de trabajo debe ser capaz de mantener un ritmo constante y sostenible de trabajo a lo largo del tiempo. Los framworks agiles apuntan a ciclos de vida iterativos de duraciíin fija, y si no llegue con todo lo previsto en la iteración entrego lo que tenga listo. **La forma de llegar a que un equipo tenga un desarrollo sostenible, es lograr que el equipo tenga un ritmo de trabajo que asegure entregar en un ciclo de tiempo fijo**

9. EXCELENCIA TECNICA
La excelencia tecnica y el buen diseño mejoran la agilidad. La calidad del producto no es negocible, debemos ajustar cualquier aspecto del producto menos su calidad al entregar.

10. SIMPLICIDAD
Al buscar la simplicidad, se puede maximizar la eficiencia y la efectividad, al mismo tiempo que se reduce la complejidad y el riesgo de errores y problemas. Esto se logra a través de la eliminación de tareas innecesarias y la concentración en los objetivos más importantes y críticos del proyecto.
No agregar funcionalidades porque si, si el cliente no lo pidió.

11. EQUIPOS AUTO-ORGANIZADOS
Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados con libertad de tomar decisiones y diseñar soluciones de manera colaborativa. Debemos tener en cuenta la importancia de permitir que los equipos de desarrollo tengan cierto grado de autonomia y capacidad de decision en el proceso de diseño y desarollo en lugar de tener un enfoque jerárquico y rigido. El agilismo rompe totalmente con el enfoque traidcional y expresa que los más aptos para definiri caracteristicas del productos, son los que van a desarrollarlo.

12. MEJORA CONTINUA
El equipo reflexiona sobre como ser más efectivo para luego ajustar y perfeccionar su comportamiento en secuencia durante ciertos intervalos de tiempo.

TRIÁNGULO DE HIERRO vs TRIÁNGULO ÁGIL

(LA TRIPLE RESTRICCIÓN DEL ENFOQUE ÁGIL)

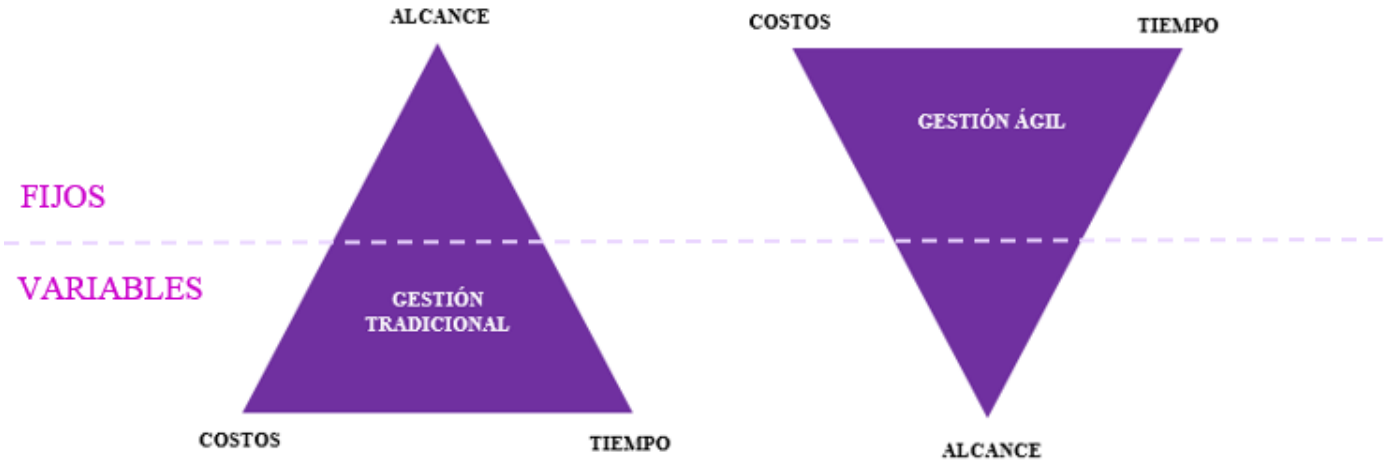
El triángulo de hierro es un modelo utilizado en la gestión tradicional de proyectos que establece que los tres elementos fundamentales en cualquier proyecto son el alcance, el tiempo y el costo. Según este modelo, cualquier cambio en uno de estos elementos puede afectar los otros dos. En la gestión tradicional de proyectos, se busca establecer un plan detallado y fijo en cuanto a estos tres elementos antes de comenzar el proyecto, y se trabaja para asegurar que se cumplan en el transcurso del proyecto.

Por otro lado, en las **metodologías ágiles**, se considera que el alcance, el tiempo y el costo son variables flexibles que pueden adaptarse y cambiarse a lo largo del proyecto. En lugar de establecer un plan fijo al inicio del proyecto, las metodologías ágiles utilizan iteraciones cortas y regulares para adaptar el proyecto según las necesidades y los requisitos cambiantes.

La triple restricción tradicional es considerada en menor medida ya que la gestión ágil establece que los tres elementos fundamentales son el **valor**, la **calidad** y la **restricción del tiempo**. En este modelo, se busca **maximizar el valor que se entrega al cliente**, **asegurar la calidad del producto** y **cumplir con los plazos establecidos**, en lugar de centrarse en el alcance y el costo. A partir de esto crea un nuevo triángulo o restricción llamada **Triangulo Ágil**

Si planteamos un ciclo de vida iterativo para un proyecto definido, sabemos que vamos a dejar fijos los requerimientos y vamos a definir las iteraciones en base a porciones de funcionalidad (CU) a desarrollar en cada iteración. Ahora bien, las iteraciones que plantea un framework ágil, se centran en el logro de entregas tempranas y por lo tanto la priorización del tiempo. Esto se plasma en la fijación del tiempo y los costos (ya que tengo un equipo fijo de personas) y la variación de las funcionalidades o alcances.

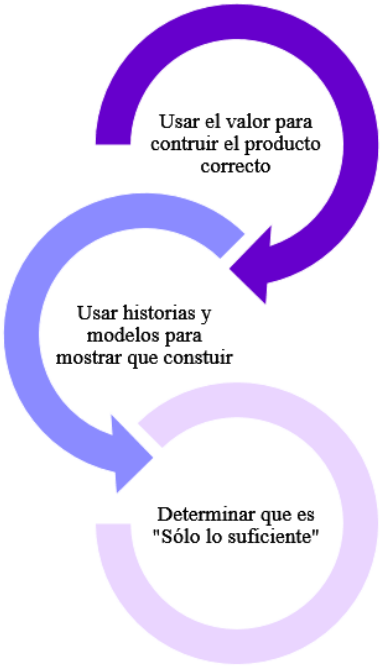
En la gestión ágil, si consideramos las mismas variables, primero me voy a guiar por el valor que le aportan las funcionalidades al desarrollo del producto. "***Tengo estos recursos y este tiempo, con eso ¿que le puedo entregar al cliente?"***



REQUERIMIENTOS ÁGILES

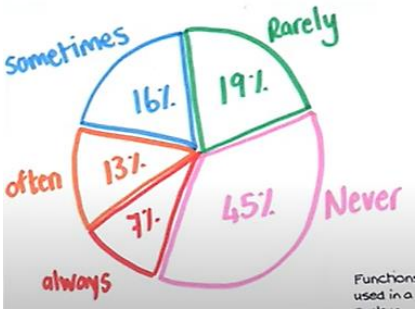
El enfoque ágil no solo plantea una gestión diferente del proyecto sino también una definición de requerimientos distinta.

En la gestión ágil, el **valor de negocio** y la construcción del producto correcto son fundamentales. Se plantea que los requerimientos se irán descubriendo a medida que avanza el proyecto y se busca definir **"sólo lo suficiente"** para comenzar a trabajar con lo mínimo necesario y obtener retroalimentación lo antes posible para mejorar continuamente.



En lugar de especificar todos los requerimientos desde el inicio, el desarrollo de requerimientos está enfocado en trabajar en conjunto con el cliente.

Los requerimientos se expresan en forma de **historias de usuario**, que son breves descripciones de los requisitos del cliente que se escriben en lenguaje natural y se enfocan en las necesidades del usuario. Estas historias se utilizan para **definir el alcance** de cada iteración del proyecto y **se priorizan en función del valor que aportan al producto** (esto significa que debe estar planteado el valor que tiene para el cliente la satisfacción de cada requerimiento). Esto difiere del enfoque tradicional que especifica todos los requerimientos antes de avanzar en el proyecto (A esto se le llama *BRUF Big Requirements Up Front*).



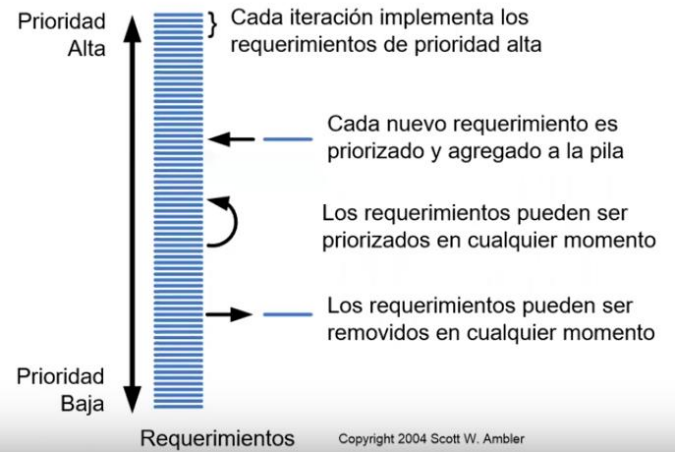
Una de las razones por las cuales necesitamos cambiar el enfoque, es decir, determinar **solo lo que suficiente**, es porque todos los productos de software tienen un **desperdicio significativo**, esto quiere decir que, de todas las funcionalidades desarrolladas, es muy alto el porcentaje de aquellas que el cliente/usuario no usa.

La gestión ágil de requerimientos intenta contrarrestar esta situación, priorizando solo aquellas funcionalidades que aportan valor al cliente. Para lograr esto, la gestión ágil utiliza el concepto de dueño del producto (Product Owner) como una figura clave en el proceso de desarrollo. El PO es el responsable de identificar las necesidades y prioridades del negocio y, en consecuencia, de priorizar los requerimientos del producto (decidir qué necesita primero y que da un valor más significativo al negocio). Estos requerimientos se organizan en una lista priorizada y ordenada, llamada **Product Backlog**. Los requerimientos más importantes se encuentran en la parte superior de la lista, mientras que los menos importantes se encuentran en la parte inferior. La organización en forma de pila del Product Backlog obliga a priorizar los requerimientos por orden de importancia.

LA GESTIÓN ÁGIL DE LOS REQUERIMIENTOS APUNTA A TRABAJAR Y A PONER EL ESFUERZO EN LO QUE NOS DA VALOR, Y A TRABAJARLO EN EL MOMENTO ADECUADO

¿CÓMO VAMOS A TRABAJAR CON LA GESTIÓN ÁGIL DE REQUERIMIENTOS?

Primero vamos a definir los requerimientos con mayor prioridad, es decir, **los que agregan más valor al cliente.**



Los requerimientos que están en el tope de la lista son aquellos en los que vamos a trabajar con **mayor nivel de detalle** y a medida que voy terminando con esos requerimientos, continuo con los que siguen en orden decreciente

A MEDIDA QUE AVANZAMOS, HAY REQUERIMIENTOS QUE PUEDEN SER REMOVIDOS, OTROS QUE PUEDEN SUBIR Y TOMAR UN MAYOR VALOR EN LA PILA Y OTROS QUE PUEDEN BAJAR, O BIEN PUEDEN AGREGARSE REQUERIMIENTOS A LA PILA.

Esta dinámica forma parte de la **gestión ágil de requerimientos**, trabajamos solo con los requerimientos de **alta prioridad** que consideramos que podemos manejar, mientras que los que se encuentran más abajo en la lista se mantienen en un estado de posibilidad de cambio en el tiempo sin que nos afecte, hasta que alcancen la suficiente prioridad como para que decidamos trabajar en ellos. En ese momento, avanzamos en su detalle y, por supuesto, posteriormente avanzamos en su implementación.

JUST IN TIME

Analizo cuando lo necesito, no antes ni después (que no se haga tarde). Analizo con detalle un requerimiento cuando tenga el suficiente valor.

Es un concepto que se usa fundamentalmente para **evitar desperdicios** (que en este contexto sería invertir tiempo en especificar requerimientos que después cambian). El producto no va a estar especificado 100% desde el principio, vamos a ir encontrando requerimientos y describiéndolos conforme haga falta.

Lo que nosotros tenemos que entregarle al cliente es valor de negocio, no características de software. El software es el medio por el cuál nosotros le entregamos valor de negocio.

METODOLOGIA TRADICIONAL vs METODLOGÍA ÁGIL

	TRADICIONAL	ÁGIL
Prioridad	Cumplir el plan	Entregar Valor
Enfoque	Ejecución ¿Cómo?	Estrategia (¿Por qué? ¿Para qué?)
Definición	Detallados y cerrados. Descubrimientos al inicio	Esbozados y evolutivos. Descubrimiento progresivo
Participación	Sponsors, stakeholder de mayor poder e interés.	Colaborativo con stakeholders de mayor interés (clientes, usuarios finales)
Equipo	Analista de negocios, Project Manager y Áreas de Proceso.	Equipo multidisciplinario.
Herramientas	Entrevistas, observación y formularios	Principalmente prototipado. Técnicas de facilitación para descubrir.
Documentación	Alto nivel de detalle. Matriz de Rastreabilidad para los requerimientos.	Historias de Usuario. Mapeo de historias (Story Mapping)
Productos	Definidos en alcance	Identificados progresivamente
Procesos	Estables, adversos al cambio.	Incertidumbre, abierto al cambio

La diferencia más marcada entre la gestión tradicional y la gestión Ágil se encuentra en la triple restricción

Anteriormente en: TRIÁNGULO DE HIERRO vs TRIÁNGULO ÁGIL

TIPOS DE REQUERIMEINTOS

DE NEGOCIO	Son los de más alto nivel, nos referimos a cuales son los requerimientos en términos de la visión del negocio
DE USUARIO	Tienen que ver concretamente con los requerimientos de usuario final
FUNCIONALES	Podríamos relacionarlo 1 a 1 con los casos de uso en la gestión tradicional
NO FUNCIONALES	Suelen ser los más ocultos, que el cliente no tiene en claro y debo especificarlos. Un requerimiento no funcional puede hacer que el software que estoy desarrollando no sirva (IMPORTANCIA FUNDAMENTAL)
DE IMPLEMENTACIÓN	Se suele decir que están dentro de los no funcionales, pero tienen que ver con cuestiones de restricción o implementación específicos.

A raíz de esta clasificación, queremos identificar aquellos requerimientos que tienen que ver con el dominio del problema y luego de la solución y por ende con los requerimientos específicos de software. Esta distinción es muy importante ya que es fundamental entender las necesidades del usuario y las restricciones del contexto en el que se utiliza el software. Esta comprensión permite a los equipos de desarrollo adaptarse rápidamente a los cambios y priorizar las características más importantes del software.
Al enfocarse en los requerimientos del dominio del problema y diferenciarlos de los requerimientos específicos de software, los equipos de desarrollo pueden tener una visión más clara de lo que el software debe hacer y enfocar su esfuerzo en las características esenciales para cumplir con los objetivos del proyecto. Esto es clave para el éxito de la metodología ágil y su capacidad para entregar software de alta calidad en un plazo corto.

En resumen, en los requerimientos ágiles se busca entender qué le da valor al negocio, cuales son los requerimientos de negocio y a raíz de estos construir una solución que pueda entregarse al cliente, priorizando lo que le da valor al mismo y a partir de entregas continuas.

En otras palabras, la gestión ágil de requerimientos implica trabajar junto a técnicos y no técnicos entendiendo las necesidades y el negocio para construir un software que de **valor**. Y luego, junto con los usuarios descubrir **cuál es la mejor forma de satisfacer esas necesidades** (Aquí aparece el Product Backlog). Entregada la característica, obtenemos feedback que nos va a ayudar a mejorar el Product Backlog si es necesario.

En este contexto, algunos conceptos relacionados con los principios agiles que aplicamos son:

- **LOS CAMBIOS VAN A EXISTIR TODO EL TIEMPO DE FORMA CONSTANTE:** Sabemos que hay cambio, por lo que se busca gestionar los requerimientos orientados a poder aceptar el cambio.
  Si el Product Owner, el cliente o los stakeholders relacionados con el producto no plantean cambios es porque el producto no está siendo utilizado.
- **DEBEMOS TENER UNA MIRADA DE TODOS LOS QUE ESTÁN INVOLUCRADOS**: Todos los que tienen algo para decir del producto (StakeHolders).
- **"EL USUARIO DICE LO QUE QUIERE CUANDO RECIBE LO QUE PIDIÓ".** Es por esto que las iteraciones cortas y las entregas continuas son esenciales, ya que permiten una retroalimentación y un enriquecimiento para futuras entregas y para una retro significativa
- **NO HAY TÉCNICAS NI HERRAMIENTAS QUE SIRVAN PARA TODOS LOS CASOS.** A veces se necesita trabajar con más prototipos o modelos, otras veces con técnicas sencillas es suficiente, pero hay que ver en cada caso que es lo que conviene.
- **LO IMPORTANTE NO ES ENTREGAR UNA SALIDA, UN REQUERIMIENTO, SINO ES ENTREGAR UN RESULTADO, UNA SOLUCIÓN DE VALOR.** Vamos a apostar a entregar una solución que le de valor al cliente, con esto nos referimos a algo que sea útil para el negocio o el cliente minimizando el desperdicio.

PRINCIPIOS ÁGILES RELACIONADOS A LOS REQUERIMIENTOS ÁGILES

Solo enunciamos los MÁS relacionados, pero de alguna manera podríamos encontrar la forma de relacionar TODOS los principios con lo que tiene que ver con Requerimientos Ágiles.

1. La prioridad es satisfacer al cliente a través de releases tempranos y frecuentes (2 semanas a un mes)

2. Recibir cambios de requerimientos, aun en etapas finales.

4. Técnicos y no técnicos trabajando juntos todo el proyecto: El PO es parte del equipo, porque si los no técnicos no son parte del equipo del proyecto, difícilmente pueda identificar qué da valor al negocio

6. El medio de comunicación por excelencia es cara a cara.

11. Las mejores arquitecturas, diseños y requerimientos emergen de equipos auto organizados

USER STORIES

Es una técnica para capturar requerimientos. Se llama "User Stories" porque es una herramienta o técnica que la manera de utilizarlo es como si contáramos una historia.

Una de las partes más difíciles de construir software es identificar los requerimientos. En primer lugar, sabemos que los requerimientos funcionales son difíciles de especificar cuando el usuario no logra definirlos, pero también sabemos la dificultad de definir requerimientos NO funcionales. Los errores cometidos en la etapa de definición de requerimientos son más difíciles de corregir ya que solo se identifican cuando el usuario tiene el sistema en frente y por ende el problema se arrastra a lo largo de toda la vida del proceso.

No es tan importante lo que yo escribo como parte de la historia, sino todo lo que se habla acerca de esa historia.

Las US no son especificaciones detalladas de requerimientos (como los CU), sino que expresan la intención de hacer algo. En general no tienen demasiado detalle al principio del proyecto y son elaboradas evitando especificaciones anticipadas sobre demoras en el desarrollo, inventario de requerimientos y una definición limitada de la solución.

La US necesita poco o nulo mantenimiento y puede descartarse después de la implementación. Junto con el códgio, sirven de entrada a la documentación que se desarrolla incrementalmente después.

Las 3 "C" de una US (o HU)

CONVERSATION

Es la parte más importante que no queda escrita en ningún lugar. Es la parte **no visible**. Todo lo que nosotros definamos en la historia nos va a ayudar a dar límites o a establecer algunos lineamientos a partir de esa conversación. Es el momento en el cual uno dialoga con el PO, o quien conoce esta interfaz en el equipo de desarrollo, para captar las necesidades del problema.

En esta conversación, es donde se obtienen todos los detalles para que el equipo pueda trabajar esa historia.

CONFIRMATION

Condiciones que se tienen que dar para dar por satisfecha la US. Esta dentro de la Card.

CARD

Es la parte visible de la US. Es donde escribimos o expresamos algo que nos indique cual es la historia de usuario, de qué se trata y su valor del negocio.

FORMA DE EXPRESAR LAS HISTORIAS DE USUARIO

As **WHO** I want **WHAT** so that **WHY**

COMO <NOMBRE DE ROL> YO PUEDO <ACTIVIDAD> DE FORMA TAL QUE <VALOR DE NEGOCIO QUE RECIBO>

El **nombre de rol** representa quién está realizando la acción o quién recibe el valor/beneficio de la actividad/acción; la **actividad** representa la acción que realizará el sistema y el **valor de negocio que recibo** comunica porque es necesaria la actividad.

Lo que más nos importa es el valor de negocio que recibo y la misma descripción explícitamente nos obliga a definir o identificar el valor de negocio que la actividad que voy a realizar devuelve.

Debemos intentar expresar la US con una **frase verbal** que nos permita identificar de que se trata/ que va a tener esa tarjeta

El rol no representa a una persona, sino lo que puede estar haciendo una persona. Ejemplo: en una estación de servicio, el playero puede estar en el despacho de combustible y después puede ir y facturar, en esas instancias tiene **roles distintos**.

Las US son **multipropósito** porque modelan o representan distintas cosas, me plantea:

- **Una necesidad del usuario.** Especifica que necesita el usuario.
- **Una descripción del producto.** Describe las características específicas del usuario.
- **Un ítem de planificación**: Al priorizar las historias, cada una de ellas es un ítem de planificación. Vamos a definir que lo que está arriba de la pila es lo que vamos a implementar, y la US nos sirve para especificarlo.
- **Token para una conversación:** Cuando una US llegue arriba de la pila de requerimientos, vamos a tener que sentarnos a especificar los detalles que va a tener esta US.

  "Tengo que acordarme que tenemos que hablar acerca de esto porque es importante"
- **Mecanismo para diferir una conversación:** También sirve para definir otras conversaciones que debemos tener a futuro.

El PO prioriza las historias en el Producto Backlog. Esta es la lista de todos los requerimientos que tiene que cumplir mi producto y que vamos a ir trabajando. Las US que están en el tope de pila son aquellos que tiene que ser modelados con mayor nivel detalle y a los que debemos prestar más atención.

La user story está planteada como una **implementación en porciones verticales**.
No se trata de solo escribir o detallar la funcionalidad que necesita el usuario, esa es la primera parte, también debemos implementarla desde la lógica de interfaz de usuario hasta como resuelvo la lógica en la BD (persistencia).
¿Si bien yo voy construyendo las porciones verticales de la US, quien ve todo para plantearlo en términos de la arquitectura del software?

MODELADO DE ROLES

Intentamos modelar los roles dentro del sistema para lograr identificar aquellos que probablemente tengamos en las US, y a partir de ellos encontrar en si las US.

- **TARJETA DE ROL DE USUARIO:** Definimos un nombre especifico y su explicación en la cual detallamos su función.

Es ambiguo poner como rol de usuario el nombre "usuario", debemos evitar esto.

También debemos evitar el uso de **usuarios representantes** como:

- Gerentes de Usuarios
- Gerentes de Desarrollo
- Alguien del grupo de marketing
- Vendedores
- Expertos del Dominio
- Clientes
- Capacitadores y personal de soporte

CRITERIOS DE ACEPTACIÓN

Los criterios de aceptación son aquellas cuestiones que le definen límites a la User Story. Ayudan a que el PO responda los criterios que necesita para que la US provea valor al negocio (definiendo los requerimientos funcionales mínimos) y también a que el equipo tenga una visión compartida de la US.

Estos criterios no contienen cuestiones de implementación, tienen que ver con la perspectiva del usuario; siempre definen una intención, no pensamos en la solución, sino en lo que de alguna manera el Product Owner nos manifiesta en su intención.

Los criterios de aceptación ayudan a su vez a los desarrolladores a saber cuándo parar de agregar funcionalidad y a los testers a derivar las pruebas pertinentes

Los criterios de aceptación buenos son aquellos que definen una intención y no una solución, que son independientes de la implementación y que son relativamente de alto nivel (no es necesario que se escriba cada detalle)

La redacción debe ser clara, tienen que haber validaciones concretas. Usamos las palabras:

PUEDE	Opcional
DEBE	Obligatorio

Son las consideraciones que define nuestro usuario a la hora de hablarnos de cómo imagina él esa funcionalidad.
No hay limitación en la cantidad de criterios
No van detalles dentro de los criterios de aceptación

PRUEBAS DE ACEPTACIÓN

Las **pruebas de aceptación** se encuentran en el dorso de la tarjeta, complementan la historia, y también expresan detalles de la conversación. Nos dan la confirmación de que finalmente lo que nosotros estamos haciendo, si todas las pruebas que nosotros definimos las pasan, va a hacer lo que el PO espera.

Profundiza lo que estamos definiendo en la US y nos ayuda a terminar de definir cómo implementar la US.

Agregamos también lo que esperamos que suceda cuando ejecutamos la historia de usuario. Entre paréntesis agregamos **PASA** o **FALLA** refiriéndonos al resultado del test.

DEFINICIÓN DE LISTO o CRITERIO DE READY

El criterio o definición de ready nos permite definir qué tiene que cumplir la US para asegurarnos de que esta lista para ser implementada. Como equipo se define este criterio.

Normalmente es una definición que acuerda el equipo y en base a eso se arma un checklist a partir del cual se toma una User Story y se determina si cumple con los aspectos acordados, si cumple con todo, puede ser incluida en un sprint, pero si no cumple con alguno de los criterios, tengo que trabajar más en esa user para poder incluirla en un sprint.

Que una US sea incluida en una Sprint quiere decir que **puedo sentarme a implementarla**, eso significa que ya está lista.

¿Qué nos ayuda a definir cuando un user está lista?

Si la US cumple con el **Modelo Invest**, se considera lista. Es una manera de trabajar sobre la Definition of Ready.

I: INDEPENDIENTE	Que no dependa de otra US, poder incluirla en una Sprint porque total no depende de otra US, puede ser implementable en cualquier orden. No me afecta a otra US mover mi US dentro del Product Backlog
N: NEGOCIABLE	Se negocia el QUÉ, no el CÓMO; por esto es que decimos que en los criterios de aceptación no ponemos detalles de implementación, lo que negociamos con nuestro cliente es el QUÉ. Si la user define el COMO está mal definida
V: VALUABLE	Debe tener un valor concreto para el cliente
E: ESTIMABLE	Tengo que poder definir cuánto esfuerzo me va a llevar hacer la US, para ayudar al cliente a armar un ranking basado en costos.
S: SMALL	La US tiene que entrar en una Sprint. Para saber si es small, tengo que poder estimarla.
T: TESTEABLE	Tengo que poder demostrar que fueron implementadas.

DEFINICIÓN DE HECHO

Esta definición de Hecho también es propia del equipo. También se valida con un checklist donde especificamos cuales son todas las características que tiene que tener la US y lo que indica es si la historia está decentemente terminada para **presentársela al PO/Cliente**. Debe tener las pruebas unitarias y de aceptación en verde, tiene que haber pasado el ambiente de prueba, etc.

NIVELES DE ABSTRACCIÓN

Cuando hablamos de requerimientos ágiles y de User Stories, es cierto que los requerimientos evolucionan con el tiempo. Puede suceder que en diferentes momentos, los requerimientos tengan diferentes niveles de abstracción, y en ese sentido, son las User Stories que cumplen con el criterio de "ready" las que estarán incluidas en el Backlog de un proyecto en un momento en específico. Sin embargo, también podemos tener otros tipos de US, como las Épicas y los Temas, que todavía no pueden ser incluidas en el Product Backlog, pero que podrían hacerlo en el futuro.

Las **ÉPICAS** son historias de usuario muy grandes, que en principio son así de grandes porque están en un lugar de la pila en donde todavía no fueron detalladas. Como todavía no debo implementarlas las dejo plasmadas como una gran US sabiendo que en algún momento voy a tener que llegar a disolverla, detallarla y hacerla más pequeña.

Y después lo que se conocen como **TEMAS**, que incluso podrían ser hasta más grandes que una épica. Es un conjunto de US relacionadas que defino en función de un título para recordarme que todo lo que está incluido en ese tema lo voy a tener que tratar en algún momento. Tienen un nivel de abstracción hasta más grande que las épicas, ya que todavía no ha llegado el momento de detallarlas y no solo son ideas concretas, sino que pueden ser iniciativas o propuestas de valor.

SPIKES

Un tipo especial de US son las **spikes** que sirven para quitar riesgo o incertidumbre de otro requerimiento, de otra US o de alguna faceta del proyecto que nosotros queremos investigar. Son específicamente creadas para esto.

Ejemplo: investigar una tecnología que no conocemos, dudas de cómo plantear alguna interacción del usuario con el sistema.

Pueden ser técnicas o funcionales.

Las SPIKES TÉCNICAS son utilizadas para investigar enfoques técnicos en el dominio de la solución. Cualquier situación en la que el equipo necesite una comprensión más fiable sobre alguna tecnología a aplicar antes de comprometerse a desarrollar una nueva funcionalidad en un tiempo fijo.

Las SPIKES FUNCIONALES son utilizadas cuando hay cierta incertidumbre respecto de cómo el usuario interactuará con el sistema. Usualmente son mejor evaluadas con prototipos para obtener realimentación de los usuarios o involucrados.

Normalmente se trabajan siempre en un sprint anterior al sprint en donde yo decido que voy a implementar la User Story en la que tengo incertidumbre.

Pueden utilizarse para:

- Familiarizar al equipo con una nueva tecnología o dominio
- Analizar un comportamiento de una historia compleja y poder dividirla en piezas manejables
- Ganar confianza frente a riesgos tecnológicos, investigando o prototipando para disminuir la incertidumbre
- Enfrentar riesgos funcionales donde no está claro como el sistema debe resolver la interacción con el usuario para alcanzar el valor/beneficio esperado.

Los spikes deben ser estimables, demostrables y aceptables. Tiene que cumplir con el Criterio de Ready.

El spike es una excepción, no siempre tengo que aplicarla, va a ser la última opción. Antes de implementar un spike, se gestiona todo dentro de la misma user.

LINEAMIENTOS A TENER EN CUENTA

Diferir el análisis detallado tan tarde como sea posible, lo que es justo antes que el trabajo comience. Hasta ese entonces, los requerimientos se capturan en forma de User Stories (como épicas o temas) y se agregan al Product Backlog según el valor que aportan al negocio

Las US no son requerimientos de software, no necesitan descripciones exhaustivas de la funcionalidad del sistema

Tener en cuenta que en la US se hace un paso a la vez, **evitar usar la palabra "y"**, usar palabras claras en los criterios de aceptación y sobre todo y más importante**, escribir la user story desde la perspectiva del usuario.**

Recordar que la parte invisible de la user (CONVERSACION) es la más importante.

ESTIMACIONES DE SOFTWARE

Como tratamos anteriormente, al momento de planificar un proyecto de software, debemos hacer estimaciones sobre el uso de recursos, costos y demás características que vamos a implementar durante el proceso de desarrollo del producto.

Estimar es predecir en un momento donde no tenemos gran conocimiento, donde hay incertidumbre.

Por definición una estimación no es precisa, tiene un alto riesgo de incertidumbre y si estamos en el inicio del proyecto aún más todavía. Cuanto más al inicio del proyecto estamos, mayor es la incertidumbre y mientras vamos avanzando en el desarrollo del mismo, la incertidumbre va disminuyendo. Si avanzamos en el proyecto y tenemos más información, tenemos que ajustar las estimaciones para que sean más precisas. No hay que esperar a tener toda la información que necesito para hacer la primera estimación. Las estimaciones deben ir ajustándose a medida que avanzamos en el desarrollo del proyecto, para volverlas más precisas

Estimar NO es planear y no necesariamente nuestro plan tiene que ser lo mismo que lo estimado. Si bien la estimación sirve como base para planificar, los planes no tienen que seguirla, las estimaciones no son compromisos. Al planear también incluimos otras consideraciones como cuestiones que tienen que ver con el objetivo del negocio y estas hacen la diferencia con la estimación de entrada. Eso sí, debemos tener en cuenta que mientras mayor diferencia haya entre lo estimado y lo planeado mayor riesgo va a haber.

¿PARA QUÉ ESTIMAMOS?

Hablamos de estimar como hablamos de predecir en un momento en donde no tenemos la certeza necesaria o hay un espectro de riesgo que puede hacer que lo que estoy prediciendo no ocurra de la manera esperada.

Generalmente al momento que estimamos tenemos menos información de la requerida, y a medida que avanzamos vamos a tener más información que va a avalar o negar la estimación realizada, pero aún así, ante el riesgo del desarrollo del producto, necesitamos estimar.

No podemos esperar a tener mucha precisión para hacer estimaciones, sino que tenemos que empezar a trabajar en esas estimaciones desde un principio, sabiendo que la incertidumbre es mayor y debemos encontrar la forma de ir disminuyéndola.

Tenemos distintas formas o **técnicas de estimación** que nos ayudan a disminuir/acotar esa incertidumbre, que vamos a ver a continuación. Estas técnicas no nos van a eliminar la incertidumbre, pero nos van a poner en un contexto medianamente esperable.

Cada técnica tiene sus propias ventajas y desventajas, y es importante seleccionar la técnica adecuada para cada proyecto en función de sus requisitos y características específicas. Además, es importante realizar una revisión y actualización constante de la estimación a medida que se avanza en el proyecto para garantizar que se ajuste a la realidad del desarrollo del software.

Si sabemos que inicialmente tenemos más probabilidades de cometer errores, es una buena práctica saber cuáles son las razones por las que nos equivocamos al estimar:

- Es súper normal y de gran impacto que, si el proyecto no está organizado, el nivel conocimiento sobre el producto es menor y eso hace que las estimaciones sean imprecisas.
- Si no tenemos en claro cuáles son los recursos con los que la empresa cuenta para desarrollar el producto ni las capacidades de los mismos, las estimaciones no suelen ser buenas.
- Las técnicas para hacer estimaciones también pueden ser fuente de error, por lo que una de las recomendaciones para evitar que esto suceda es utilizar más de una técnica y comparar sus resultados. Si tengo resultados muy distintos utilizando diferentes técnicas entonces tengo errores.

Lo primero que estimamos es el **TAMAÑO DEL SOFTWARE.**

Con tamaño del software nos referimos a que tan grande va a ser el trabajo que vamos a hacer. Para esto utilizamos la técnica de **CONTAR**

- Contar funcionalidades
- Contar requerimientos
- Contar la cantidad User Stories (estableciendo una métrica de conteo)
- Contar la cantidad de Casos de Uso

Acá hay un juego entre la información que tenemos para hacer la predicción o estimación y el nivel de incertidumbre o riesgo que vamos a tener.

A su vez nos sirve utilizar **datos históricos** que nos van a servir para comparar el tamaño posible de nuestro software con un software ya desarrollado con características similares.

Otra de las estimaciones que realizamos es el **ESFUERZO.**

Con esfuerzo nos referimos a la cantidad de horas lineales que voy a necesitar para construir el software, no incluyo qué personas lo hacen, ni calendario ni tampoco me paro a especificar si las actividades son paralelas o secuenciales, solo **CUENTO,** en estimación, cuantas horas lineales de desarrollo voy a necesitar.

Luego de estimar el esfuerzo, estimo el **CALENDARIO**, que es justamente calendarizar el esfuerzo y proponer una fecha de entrega.

Una vez que logramos estimar tamaño, esfuerzo y tiempo calendario, vamos a empezar a trabajar con **COSTOS** y **PRESUPUESTOS.**

TÉCNICAS FUNDAMENTALES DE ESTIMACIÓN

Dentro de las técnicas de estimación, hay distintos métodos utilizados, es decir, están basados en distintos métodos.

Estos son:

- **METODOS BASADOS EN LA EXPERIENCIA:** Esta técnica se basa en la experiencia pasada del equipo en proyectos similares para determinar la estimación de esfuerzo y tiempo requerido para un nuevo proyecto.
- **METODOS BASADOS EXCLUSIVAMENTE EN RECURSOS**: Esta técnica se basa en la cantidad y el tipo de recursos (personas, tiempo, herramientas, etc.) necesarios para llevar a cabo un proyecto. La estimación se realiza en función de la disponibilidad y costo de los recursos necesarios.
- **METODOS BASADOS EXCLUSIVAMENTE EN EL MERCADO**: Esta técnica se basa en el costo de proyectos similares en el mercado para determinar la estimación de esfuerzo y tiempo requerido para un nuevo proyecto. Esta técnica se utiliza comúnmente en proyectos de desarrollo de software para clientes externos.
- **METODOS BASADOS EN LOS COMPONENTES DEL PRODUCTO O EN EL PROCESO DE DESARROLLO:** Esta técnica se basa en la identificación y descomposición de los componentes del software y del proceso de desarrollo para estimar el esfuerzo y tiempo requerido para cada componente. Luego, se suman las estimaciones para obtener una estimación total del proyecto.
- **METODOS ALGORITMICOS:** Esta técnica se basa en modelos matemáticos y estadísticos para determinar la estimación de esfuerzo y tiempo requerido para un proyecto. Estos modelos pueden tener en cuenta diferentes variables como el tamaño del software, la complejidad, la productividad del equipo, etc.

Dentro de los **métodos basados en la experiencia** tenemos las técnicas de estimación de:

DATOS HISTORICOS: Implica comparar un proyecto nuevo con uno anterior que me nutra y me permita hacer la estimación de mi proyecto.
Los datos básicos históricos concretos que voy a necesitar para tomar como entrada a mis estimaciones son el tamaño, el esfuerzo, el tiempo y los defectos. Necesito contar con información completo de estos datos de manera que se tomen siempre de la misma manera para los proyectos que analizo.
Problema: Los dominios analizados pueden ser muy distintos. Debemos tener estructurados los datos históricos

JUICIO EXPERTO
(más utilizado)
Esta técnica se basa en la experiencia y conocimiento de expertos en la materia para determinar la estimación de esfuerzo y tiempo requerido para un proyecto de software. Una vez identificados los expertos, que es importante identificarlos bien, se les presenta la descripción detallada del proyecto y se les solicita que proporcionen estimaciones basadas en su experiencia y conocimiento. Es importante tener en cuenta que la técnica de juicio de experto se basa en la subjetividad y experiencia de los expertos, y las estimaciones pueden variar significativamente dependiendo de la experiencia y conocimiento de los expertos involucrados

PURO
Un experto estudia las especificaciones y hace su estimación.
Se basa fundamentalmente en los conocimientos del experto.

Desventaja → Si desaparece el experto, la empresa deja de estimar

WIDEBAND DELPHI
Un grupo de personas son informadas y tratan de adivinar lo que costara el desarrollo tanto en esfuerzo como en duración.
Las estimaciones en grupo suelen ser mejores que las individuales.
Las estimaciones individuales **se recopilan de forma anónima** y se envían a los expertos para que puedan revisarlas y ajustarlas en rondas sucesivas.
Desventaja → requiere más tiempo y recursos que el juicio de experto puro, ya que implica múltiples rondas de estimaciones y análisis de datos hasta que la estimación converja de forma razonable.

Algunos criterios que pueden seguir los expertos para lograr una buena estimación y lograr una estructuración del juicio de experto son:

- ✓ Estimar solo tareas con granularidad aceptable (no muy alta)
- ✓ Usar el método **Optimista, Pesimista y Habitual** que tiene que ver con estimar según 3 características y luego aplicar esto a una formula. **(O+4H+P)/6.**
  4 veces el tiempo habitual + el tiempo optimista + el pesimista y todo dividido 6
  No es para seguirlo a rajatabla. Yo defino cuanto me va a llevar una determinada actividad, después planteó la fórmula y es el tiempo que me llevará.
- ✓ Usar un checklist y un criterio definido para asegurar cobertura.

ANALOGÍA: También se basa en datos históricos pero se seleccionan específicamente aquellos proyectos que son similares al proyecto actual para realizar la comparación y estimación.

Al hacer estimaciones muchas veces omitimos ciertos aspectos que recaen en errores, por lo tanto, debemos tener en cuenta ciertas actividades omitidas como:

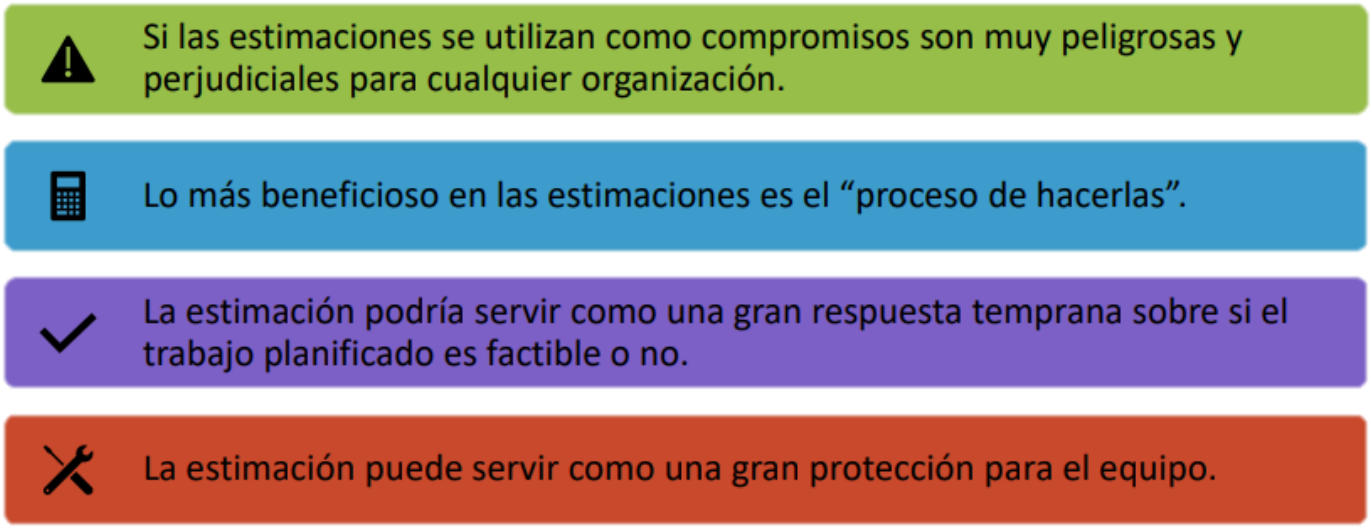
- Requerimientos faltantes
- Actividades de desarrollo faltantes (documentación técnica, participación en revisiones, creación de datos para el testing, mantenimiento de producto en previas versiones…)
- Actividades generales (días de enfermedad, licencias, cursos, reuniones de compañía…) –

Al estimar somos más bien optimistas. Los desarrolladores siempre generan cronogramas demasiado optimistas ☺ y esto tiene que ver con que las estimaciones de software son algo nuevo y muchas veces subestimamos las dificultades con las que nos podemos encontrar.

No debemos olvidar que la estimación se va refinando a medida que el proyecto se va desarrollando.

ESTIMACIÓN EN AMBIENTES AGILES

Cuando queremos precisar la estimación en ambientes agiles seguimos con los mismos conceptos anteriores, pero debemos tener en cuenta algunas cuestiones:



Debemos recordar a su vez que no debemos ser dogmáticos sobre nada y que la clave del éxito en casi todo, en contraposición es ser pragmáticos.

Repetimos una vez más que la estimación no es un plan, ya que un plan implica compromiso y si la estimación implicar un compromiso va a resultar perjudicial para cualquier organización.

El tener estimaciones que se vayan ajustando nos permite observar si la estimación realizada es alcanzable o no.

En **estimaciones ágiles** puntualmente, se va a trabajar con las **features** o **stories**.

Las features/stories van a ser estimadas usando una medida de tamaño relativo conocida como **STORY POINTS (SP).** Los story points son una ponderación que se le da a la historia de usuario cuyo peso es la combinación de su **incertidumbre**, **esfuerzo** y **complejidad**.
La estimación en Story Points separa completamente la estimación de esfuerzo de la estimación de la duración del proyecto.

Se trata de **medidas o estimaciones relativas** ya que las define un equipo y no son comparables entre distintos equipos.
Al ser específicas de cada equipo **no son absolutas** y a su vez tampoco es una medida basada en el tiempo.
Una de las premisas de la estimación relativa es la realización de la misma a partir de la comparación. Esto lo hacemos a través de la definición de una User Story Canónica (de base) no necesariamente la más chiquita pero una que defina en base a que vamos a definir los Story Points.
De esta forma se define una mejor dinámica y acuerdo grupal y un pensamiento de equipo más centrado; además de permitir emplear mejor el tiempo de análisis de las stories.

Para poder estimar una US debo estimar el **tamaño**, el **tiempo** y el **esfuerzo necesario** de la misma.
El **tamaño** es una medida de la cantidad de trabajo necesario para producirla o completarla, y a su vez indica cuan compleja y cuán grande es.
Las estimaciones basadas en base al **tiempo** son más propensas a errores debido a factores externos como las habilidades de las personas, el conocimiento del dominio, la experiencia, etc.
No debemos confundirnos, **tamaño NO es esfuerzo**, ya que **esfuerzo** tiene que ver con horas lineales.
El esfuerzo lo que tiene de bueno con respecto al tiempo calendario es que nos ayuda a despegarnos de cuestiones que implican tiempos determinados (que suelen tener más errores).

Para medir el progreso de un equipo durante el desarrollo de un proyecto y específicamente en una iteración, tenemos la llamada:

VELOCITY
Velocidad/Velocity es una medida (métrica) del progreso de un equipo. Se calcula a partir de la suma de los Story Points asignados a cada User Story que el equipo completó al 100% durante la iteración que se está midiendo. No vamos a contar los SP de las US parcialmente completas. Esta métrica o medida nos sirve para corregir errores de estimación gracias a la gran cantidad de información que nos brinda.

A partir de esta métrica también podemos derivar la duración de un proyecto. Lo que hacemos es tomar el número total de story points de las US del proyecto entero y lo dividimos por le velocidad del equipo.
La velocidad nos ayuda a determinar un horizonte de planificación apropiado.

POSIBLES MÉTODOS DE ESTIMACIÓN

POKER ESTIMATION

Combina el juicio de experto con analogía y utiliza algo de Wideband Delphi.
Se basa en la *Serie de Fibonacci* ya que al asignarle a la complejidad valores de la serie, planteo un crecimiento exponencial del peso de la user. Esto me permite una estimación un poco más exacta.
La secuencia empieza en 1 y cada número subsecuente es la suma de los dos precedentes. (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…)

Generalmente si tenesmo un SP muy grande, es probable que la US no se pueda ejecutar en un solo sprint y por ende no se cumplirá con el **criterio de Ready**.

La canónica o base, debe tener complejidad 1 y si se puntúa una con 0 quiere decir que hay desconocimiento sobre la misma y es necesario detallarla.
Si el SP es 100, evidentemente algo está muy mal.
El límite de SP es 8.

Los participantes en **Poker Planning** (como se suele llamar) son desarrolladores, ya que quienes estiman deben ser competentes en resolver la tarea. Se llama así ya que se usan cartas.
Cada miembro del equipo de desarrollo selecciona una carta que representa un valor numérico, que puede ir desde 0 hasta 100, y la coloca boca abajo sobre la mesa. Una vez que todos los miembros del equipo han elegido su carta, se vuelven a dar vuelta simultáneamente para que todos puedan verlas.
Los miembros del equipo de desarrollo que eligieron las cartas con los valores más altos y los valores más bajos tienen la oportunidad de explicar su razonamiento para elegir esas cartas. Luego, se lleva a cabo una nueva ronda de estimación hasta que se llegue a un consenso en cuanto al valor de Story Points para cada User Story.

Cuando hablamos de Agile hablamos de empirismo, por lo tanto, en las estimaciones también tenemos que cumplir con las características y pilares del empirismo, que son ejecutar, inspeccionar y adaptar, retroalimentación.

GESTIÓN DE PRODUCTOS

Cuando hablamos de gestión de productos, empezamos a pensar en términos de visión para que construimos productos de software, en este contexto aparecen distintas razones:

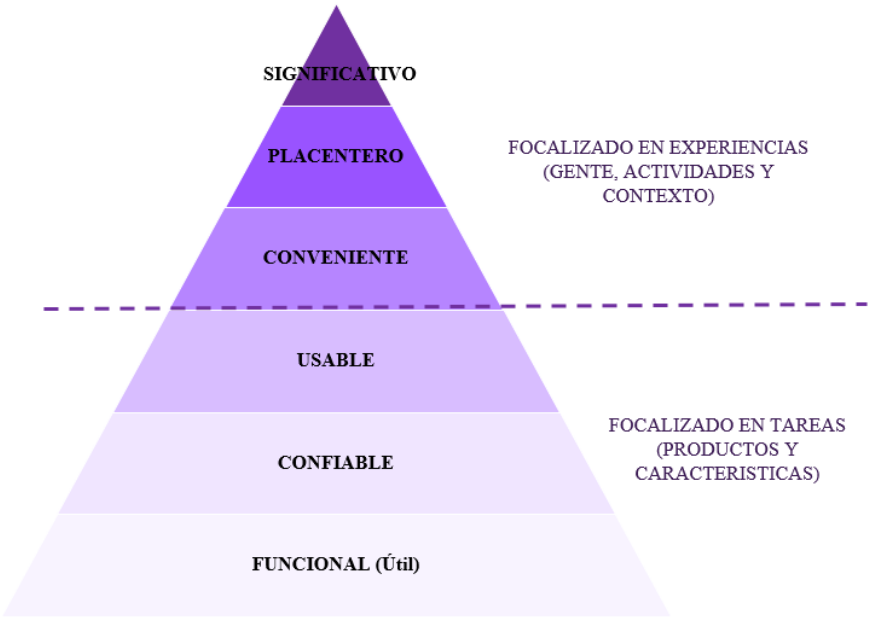
¿POR QUÉ CREAMOS PRODUCTOS?

No solo está asociado con lograr la satisfacción de los clientes, la obtención de dinero o la masividad y popularidad de la aplicación, sino también con una visión de cambiar el mundo o nuestra diaria (es una visión más ambiciosa que crear un producto que nos pide un cliente)

Independientemente de esta visión, sabemos que muchas veces se invierte mucho tiempo en construir características del software que no se usan o se usan muy rara vez. Esto implica un esfuerzo innecesario en algo que no se va a utilizar o vender. Es muy importante entonces, eliminar el desperdicio para poner nuestro esfuerzo en aquello que sea útil.

EVOLUCIÓN DE LOS PRODUCTOS DE SOFTWARE

Teniendo en cuenta las dos concepciones enunciadas, podríamos armar una especie de pirámide con respecto a la evolución de los productos de software, pensando en las características del software.



En la **base** de la pirámide pensamos aquellas que, si o si, tienen que existir para que el software funcione para el propósito esperado, que sean confiables y que de alguna manera tengan cierto lineamiento en experiencia de usuario que nos permitan lograr lo que esperamos de la funcionalidad.
Estos 3 aspectos que se encuentran en la base de la pirámide son lo que todos esperamos cuando construimos o cuando vamos a usar un producto de software.

En el **tope** de la pirámide aparecen otros aspectos relacionados con otras cuestiones de la visión del producto que son difíciles de alcanzar. Hablamos de un producto conveniente, placentero y significativo.

Normalmente los productos que se construyen en las organizaciones se quedan en la base de la pirámide y hasta ahí llegan.

El desafío tiene que ver con **cómo hacemos para construir un producto de software donde desechamos el desperdicio y logramos abarcar los aspectos que estamos persiguiendo cuando creamos el producto de software** (ya sea satisfacer a un cliente o la venta del producto, etc.)

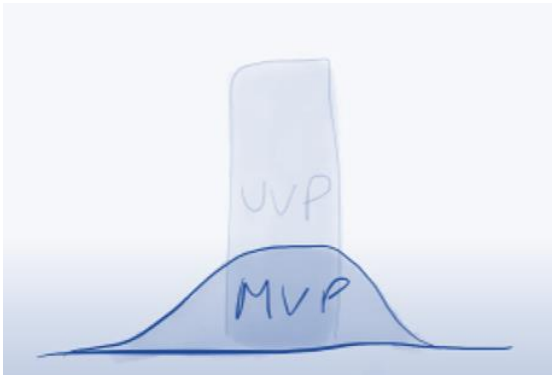
Nos vamos a centrar en desarrollar el mínimo producto o mínima característica para saber si el producto que vamos a construir va a cubrir las expectativas del usuario o las propias al desarrollar el producto.
La idea es plantear una hipótesis y con un **mínimo** desarrollo, un **mínimo** esfuerzo, y validarla para hacer justamente que el desperdicio tienda a 0. No gastar esfuerzo/energía en algo que no va a agregar valor/beneficio al producto.
Para esto, podemos utilizar la **técnica UVP** (User Value Proposition o Propuesta de Valor para el Usuario) es una técnica de desarrollo de productos que se centra en comprender las necesidades, deseos y problemas de los usuarios y en crear soluciones que satisfagan esas necesidades de manera efectiva.

En principio nos vamos a imaginar una **hipótesis** o un valor que tiene el producto o servicio que queremos desarrollar. Mi visión del producto va a ser poder resolver esa hipótesis única.
Si nosotros a raíz de esta hipótesis desarrollamos un producto completo con muchas funcionalidades y recursos utilizados y de ahí lo comercializamos, corremos el riesgo de que el producto no sea algo que el mercado o los usuarios estén demandando o que no cubra las expectativas del usuario exigente.
A raíz de esto estaríamos desperdiciando recursos y esfuerzo para un producto que no va a dar beneficio.

Es por esto que buscamos una forma de **minimizar el esfuerzo** para poder construir un producto que nuestro usuario quiere, si es que la hipótesis es algo que realmente el mercado este demandando.
Aparece entonces el concepto de:

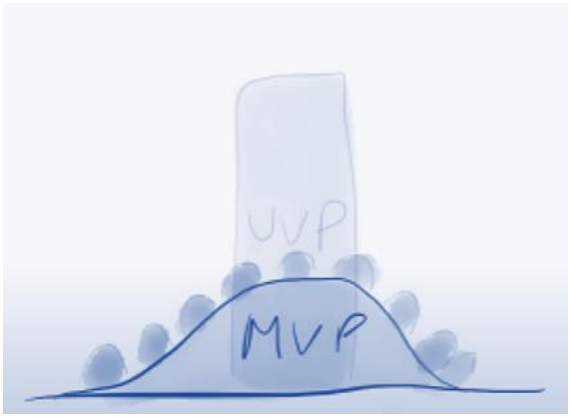
MVP (Mínimum Viable Product – Producto Mínimo Viable)

Es un concepto de Lean Startup que enfatiza el impacto del aprendizaje en el desarrollo de nuevos productos.



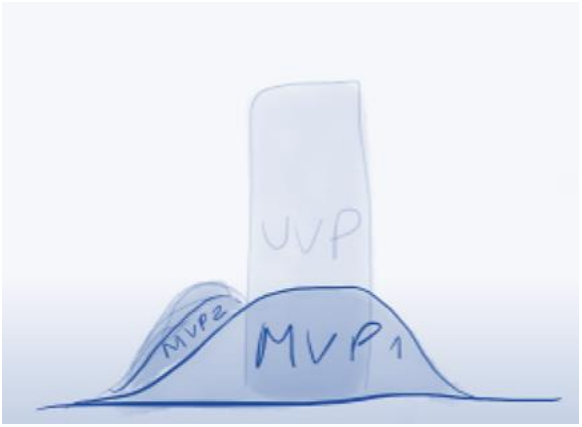
La idea del MVP es trabajar con lo mínimo del producto que necesito obtener. Lo mínimo que tendría que tener el producto para que los (posibles) usuarios del mismo lo puedan evaluar y me puedan decir si el producto les atrae o no.
En este producto de funcionalidades mínimas, evidentemente, va haber funcionalidades extra que lo definan como producto en sí.

El **objetivo** del MVP es **comprobar que la hipótesis del producto a construir funciona** o si tengo que hacer cambios. La idea es iniciar un circuito de aprendizaje a partir de la participación de los usuarios y la retroalimentación sobre si el producto cubre expectativas, sirve o si tengo que hacer cambios en las características incluidas.
No es vender el MVP, es validar la hipótesis.



Puede suceder, como bien dijimos, que asi como está definida la hipótesis sirva (todos los clientes/usuarios que vean ese MVP se sientan a gusto y expresen que lo necesitan) o puede ocurrir que cada potencial usuario de ese MVP cuando lo vea pida funcionalidades diferentes. Si con cada cliente potencial o con cada usuario que prueba la MVP recibo un feedback distinto, quiere decir que no está tan claro el mercado en el cual este producto va a funcionar (por eso todas las pequeñas desviaciones) y significa que tengo que seguir trabajando sobre la visión del producto para encontrarlo.

Otra alternativa que podría suceder es que, en algún punto, la hipótesis planteada tenga algunas variaciones. Y en este contexto, la hipótesis puede moverse o modificarse según la exigencia del mercado.
Mi MVP se va moviendo hacia el foco en el cual los clientes se van a interesar. Si bien el foco elegido en la hipótesis no es erróneo, algunas características del mismo no están tan bien, por lo tanto, las desviamos.



En este escenario debemos redefinir el MVP en base al conocimiento adquirido, por lo que creamos un nuevo MVP2. Lo que hacemos es pivotar hasta encontrar específicamente cual es el producto que nosotros queremos llegar a construir.

Este circuito de retroalimentación se vuelve hacer, pero con este nuevo MVP. (Este proceso puede ocurrir varias veces hasta encontrar un feedback exitoso)

Una vez que encontramos el MVP exitoso, es decir que el feedback y la investigación son exitosas, surge el concepto de:

MMF (Mínimum Marketable Feature – Característica Mínima Comercializable)

Se enfoca en definir los elementos mínimos que deben estar presentes en un producto o servicio para que sea comercializable. Es la validación del producto en términos de la comercialización

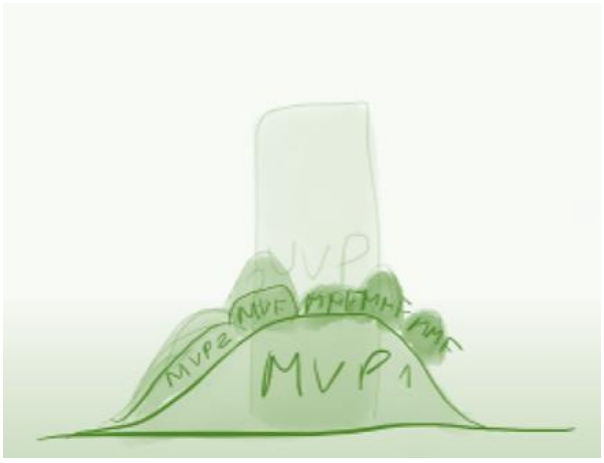
Ya no estoy parado en una hipótesis para validar si el producto tiene mercado o no, sino que estoy definiendo cual es la pieza mínima que voy a tener que construir y comercializar para que el producto sea rentable. Esto es ya que, de nuevo, no tiene sentido construir el producto completo hasta no ver si va a funcionar su comercialización.

También es importante construirlo en cuestiones mínimas para sacarlo rápidamente a la venta cuando el producto es atractivo y no está en el mercado, ya que cuanto más tiempo pase más riesgo corro de que mi idea sea implementada antes. Seguimos con la idea de aplicar el menor esfuerzo posible y no esfuerzos innecesarios antes de saber si el producto es comercializable o no.

El **objetivo** es asegurarnos de que este primer MMF sirve o no. Con la diferencia de que ahora hablamos de una Feature, algo mucho más chiquito que el MVP. Es lo mínimo que puedo sacar a la venta.

Para conjugar/combinar el MVP con la MMF, es a partir de la llamada:

MVF(Minimum Viable Feature - Característica Minima Viable)



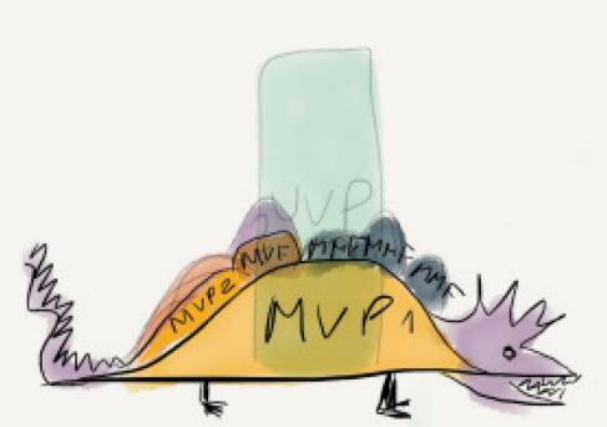
Se trata de la característica mínima que se puede construir e implementar rápidamente, utilizando recursos mínimos, para probar la utilidad de la misma.

Quiero validar cual es la característica que es la que hace la diferencia y que va a hacer en primera medida que compren mi producto.

Es una feature que buscamos que por si misma ofrezca un valor agregado. Este es nuestro objetivo, ahora la hipótesis se centra en una característica en vez de un producto.

Si la MVF resulta exitosa, se pueden desarrollar más MMF en esta área para tomar ventaja (ya que está comprobada).

De alguna forma la MVF es una versión mini del MVP.



Si seguimos trabajando en el contexto de validar las hipótesis y ver cómo vamos a trabajar con cada característica, cuando se logra ajustar el producto en el mercado, se combinan MMF y MVP según el nivel de incertidumbre del negocio o las áreas en las que se está enfocando, y se forma el dinosaurio ☺

También tenemos la **MRF (Minimum Release Feature – Caracteristicas Mínimas del Release)** que se trata del release del producto que tiene el conjunto de características más pequeño posible. El incremento más pequeño que ofrece un valor nuevo a los usuarios y satisface sus necesidades actuales.

MMP = MMR1 (pueden salir más releases después, que mejoren el producto/release inicial)

Al MRF lo obtengo en varias iteraciones hasta q tengo una cierta cantidad de características a lanzar.

RELACIONES CONCRETAS

MVP

- Versión de un **nuevo producto** creado con el **menor esfuerzo posible**
- Dirigido a un **subconjunto de clientes potenciales**
- Utilizado para obtener **aprendizaje validado**.
- Más cercano a los **prototipos que a una version real funcionando de un producto**.

MMF

- es la **pieza más pequeña de funcionalidad** que puede ser liberada
- tiene valor tanto para la organización como para los usuarios.
- Es parte de un MMR or MMP.

MMP

- Primer release de un MMR dirigido a **primeros usuarios** (early adopters),
- Focalizado en características clave que satisfarán a este grupo clave.

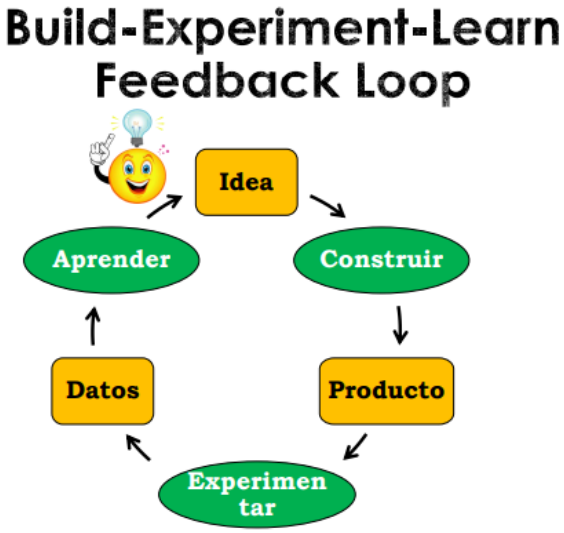
MMR

- Release de un producto que tiene el conjunto de características más pequeño posible.
- El incremento más pequeño que ofrece un valor nuevo a los usuarios y satisfice sus necesidades actuales.
- **MMP = MMR1**

ERRORES MÁS COMUNES MVP vs MMF o MMP

- Confundir un MVP (que se enfoca en el aprendizaje y en validar la hipótesis) con Característica Comerciable Mínima (MMF) o con Producto Comercializable Mínimo (MMP) que ambos se enfocan en tener un valor de retorno, en ganar.
- El riesgo de esto es entregar algo sin considerar si es lo correcto que satisface las necesidades del cliente.
- Enfatizar la parte mínima de MVP con exclusión de la parte más viable. El MVP no tiene que ser solo mínimo sino también viable. El producto entregado no es de calidad suficiente para proporcionar una evaluación precisa de si los clientes utilizarán el producto.
- Entregar lo que consideran un MVP, y luego no hacer más cambios a ese producto, independientemente de los comentarios que reciban al respecto no está bien.

Uno de los aspectos importantes de esta metodología o técnica es el **ciclo de Feedback** o **de Aprendizaje**.



El éxito no es entregar un producto, el éxito se trata de entregar un producto (o característica de producto) que el cliente usará.

La forma de hacerlo es alinear los esfuerzos continuamente hacia las necesidades reales de los clientes.

The Build-Experiment-Learn feedback Loop permite descubrir las necesidades del cliente y alinearlas metodológicamente.

LA FASE CONSTRUIR DEL MVP

Es importarte saber que cuando construimos el MVP, su complejidad puede variar en complejidad, desde anuncios, explicaciones simples, hasta prototipos tempranos. La idea es tener rápidamente y con el menor esfuerzo tener el MVP para poder validarlo y retroalimentar la hipótesis para entrar en el ciclo de aprendizaje.

Cuando hablamos de MVP empiezan a surgir un montón de problemáticas sobre cómo hacemos para minimizar el desperdicio y el esfuerzo al construir el MVP. Para esto debemos:

- En caso de duda, simplificar
- Evitar la construcción excesiva y la promesa excesiva
- Saber que cualquier trabajo adicional más allá de lo que necesita para comenzar el ciclo podría ser un desperdicio

El punto central del Loop de Feedback entonces, tiene que ver con experimentar: **LA FASE EXPERIMENTAR** pasa a ser clave al trabajar en la construcción del MVP.

Aca empiezan a surgir distintos planteos al momento no solo de construir un MVP sino de construir en si un nuevo producto.

AUDACIA DE CERO (incentivo)

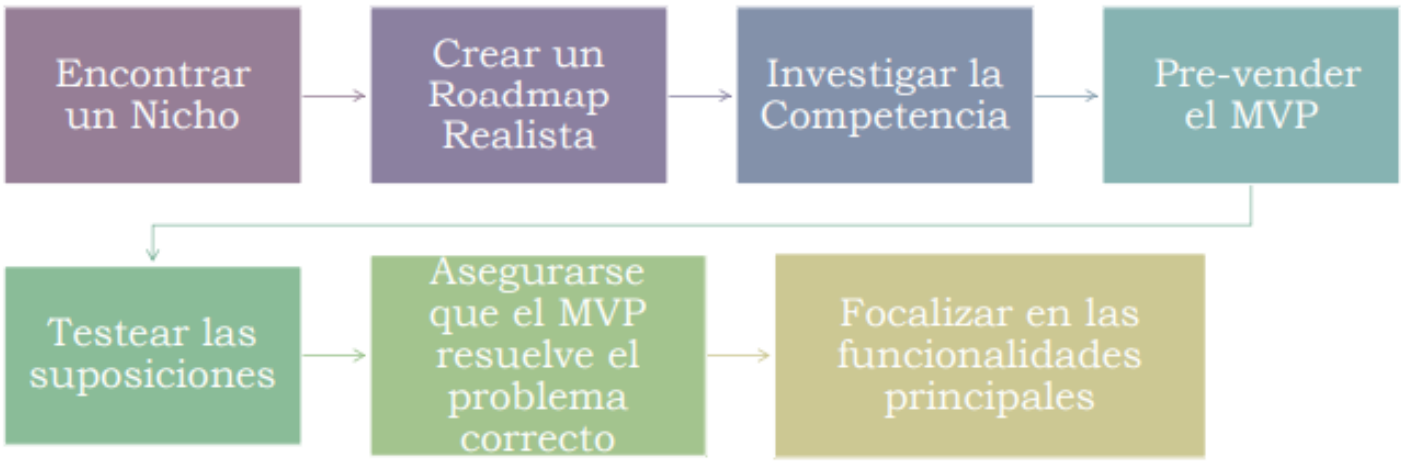
Debemos pensar en término de la construcción de un producto en grande, aunque resulte más riesgoso. Si se pospone la experimentación con el MVP, van a surgir resultados como una gran cantidad de trabajo desperdiciado, perdida de información esencial y el riesgo de construir algo muy poco significativo.

Es por esto que debemos utilizar el MVP para **experimentar** (inicialmente, en silencio) con los primeros usuarios en el mercado verificando mi hipótesis probando todos los elementos disponibles comenzando por los más riesgosos

SUPUESTOS DE "SALTOS DE FE"

Los elementos más riesgosos del plan / concepto de un nuevo producto o stratup se denominan supuestos de salto de fe. La mayoría de las personas no conocen una determinada solución (o incluso un problema); pero una vez que experimentan la solución, ¡no pueden imaginar cómo vivirían sin ella!

PREPARAR UN MVP



SOFTWARE CONFIGURATION MANAGEMENT (SCM)

(Gestión de configuración de software)

Cuando pensamos en software no solo pensamos en el producto sino en todo el entorno, las herramientas, los procesos, etc. El software maneja información estructurada con propiedades lógicas y funcionales, creada y mantenida en varias formas y representaciones y confeccionada para ser procesada por computadora en su estado más desarrollado.

Los sistemas de software siempre cambian durante su desarrollo y uso. Conforme se hacen cambios al software, se crean nuevas versiones del sistema. Los sistemas pueden considerarse como un conjunto de versiones donde cada una de ellas debe mantenerse y gestionarse. Esto es necesario para no perder la trazabilidad de los cambios que se incorporan a cada versión.

Estos cambios en el software tienen su origen en cambios del negocio y nuevos requerimientos, cambio en productos asociados al software, reorganización de las prioridades de la empresa, cambios en el presupuesto, defectos a corregir y oportunidades de mejora. En este contexto, el software evoluciona y esta evolución puede verse en muchos aspectos del producto.

Ante estos cambios, es ideal que haya integridad en el producto de software y a través de SCM logramos esto.

SCM es una actividad de soporte transversal a todo el proyecto que aplica dirección y monitoreo administrativo y técnico, y cuyo propósito es mantener la integridad del producto a lo largo de todo el ciclo de vida. Sirve de contención para poder construir un producto íntegro y de calidad.

El propósito de la SCM es también establecer la integridad de los productos y esto involucra:

1. Identificar características técnicas y funcionales de **ítems de configuración**
2. Documentar características técnicas y funcionales de **ítems de configuración**
3. **Controlar los cambios** de esas características identificadas y documentadas
4. **Registrar y reportar estos cambios**
5. Verificar **correspondencia con los requerimientos** (trazabilidad).

Tiene aplicación en diferentes disciplinas como:

- Control de calidad de proceso
- Control de calidad de producto
- Prueba de software

Su propósito es resolver **problemas** de diferente índole, a través del establecimiento y el mantenimiento de la integridad del producto de software a lo largo de todo su ciclo de vida.

Algunos de esos problemas son:

PÉRDIDA DE COMPONENTES
PÉRDIDA DE CAMBIOS **(LA VERSIÓN DEL COMPONENTE NO ES LA ÚLTIMA)**
REGRESIÓN DE FALLAS
DOBLE MANTENIMIENTO
SUPERPOSICIÓN DE CAMBIOS
CAMBIOS NO VALIDADOS.

Su propósito es establecer y mantener la integridad de los productos de software a lo largo de su ciclo de vida. Esto implica identificar la configuración en un momento dado, controlar sistemáticamente sus cambios y mantener su integridad y origen.

La integridad es el medio por el cual podemos garantizar que el producto a entregar tiene la calidad correspondiente. Y nos garantiza un nivel mínimo de confiabilidad.

El problema de la calidad es que es subjetiva y se suma a que el software es intangible, por lo tanto, es muy difícil medir si la calidad que responde al cumplimiento de las expectativas del cliente realmente se verifica. La idea de la integridad del producto es hacer explícita las características o expectativas que tiene el cliente sobre el producto de software.

Decimos que se mantiene la integridad de un producto de software cuando:

1. **SATISFACE LAS NECESIDADES DE USUARIO**: hace lo que el usuario espera que haga
2. **PERMITE LA RASTREABILIDAD DURANTE TODO SU CICLO DE VIDA**: existen vínculos o conexiones entre los ítems de configuración (los cuales deben ser definidos desde un primer momento del ciclo de vida del producto), que permiten analizar en donde impactará un cambio en un ítem de configuración. De esta forma, se puede determinar cómo impactará cada cambio de requerimientos a través de la trazabilidad. A mayor cantidad de vínculos → mayor información → mayor trazabilidad → mayor costo (analizar la relación costo-beneficio);
3. **SATISFACE CRITERIOS DE PERFORMANCE Y RNF**
4. **CUMPLE CON EXPECTATIVAS DE COSTO**: este apartado incluye la satisfacción del equipo de proyecto. No solo el cliente debe estar satisfecho, sino también el desarrollo del producto debe ser redituable. Mientras el producto existe hay actividad de gestión de configuración, que es responsabilidad de todo el equipo. Sin embargo, existen roles específicos como el rol del Gestor de configuración que tiene tareas adicionales como por ejemplo mantener la herramienta, marcar línea base, etc.

CONCEPTOS GENERALES

ÍTEM DE CONFIGURACIÓN

Son aquellos artefactos que forman parte del producto o proyecto, y pueden ser almacenados en un repositorio, sin importar su extensión/tipo. Pueden sufrir cambios, y se desea conocer su estado y evolución a lo largo del ciclo de vida (ya sea del producto o proyecto, dependiendo del artefacto). Es decir, es cualquier aspecto asociado al producto de software (requerimientos, diseño, código, datos de prueba, documentos, etc.) que es necesario mantener.

Son todos aquellos elementos que componen toda la información producida como parte del proceso de ingeniería de software, como ser programas de computadora (código fuente y ejecutables), documentos que describen los programas (documentos técnicos o de usuario), datos (de programa o externos), etc.

Los ítems, dependiendo de su naturaleza, pueden durar lo que dure el ciclo de vida del proyecto, producto o sprint.

Algunos ejemplos: prototipo de interfaz, manual de usuario, ERS, arquitectura del software, casos de prueba, código fuente, íconos, etc, etc.

REPOSITORIO

Es un contenedor de ítems de configuración, se encarga de mantener la historia de cada ítem con sus atributos y relaciones, además es usado para hacer evaluaciones de impacto de los cambios propuestos. Puede ser una o varias bases de datos. Tiene una estructura para mantener el orden y la integridad. El que tenga una estructura ayuda a la seguridad, los controles de acceso, las políticas de Backup y todo aquello que se aplica sobre un repositorio.

Tenemos 2 tipos de repositorios:

- **CENTRALIZADO**: está caracterizado porque un servidor contiene todos los archivos con sus versiones. Los administradores tienen un mayor control sobre el repositorio. Tiene como desventaja qué si falla el servidor, todo lo positivo se cae
- **DESCENTRALIZADO**: está caracterizado porque cada cliente tiene una copia exactamente igual del repositorio completo. Tiene como ventaja que se resuelve el problema de los servidores centralizados, debido a que si el servidor falla sólo es cuestión de "copiar y pegar", además posibilita otros workflows no disponibles en el modelo centralizado. La desventaja que podemos mencionar es que es más complicado llevar un control sobre el mismo

VERSIÓN

Instancia de un ítem de configuración que difiere de otras instancias de este ítem.

Las versiones se identifican unívocamente. Controlar una versión refiere a la **evolución de un único ítem de configuración**.

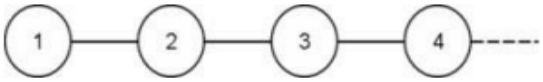
"La gestión de configuración permite a un usuario especificar configuraciones alternativas del sistema de software mediante la selección de las versiones adecuadas"; esto se puede gestionar asociando atributos a cada versión (que pueden ser datos sencillos como un nro. de versión asociado a cada objeto).

Cada versión de software es una colección de **elementos de configuración** (ECS) (como ser código fuente, documentos, datos).

La versión es un punto particular en el tiempo de ese ítem de configuración (es un estado).

Una versión se define, desde el punto de vista de la evolución, como la forma particular de un artefacto en un instante o contexto dado.

El **control de versiones** se refiere a la **evolución de un único ítem de configuración** (IC), o de cada IC por separado. La evolución puede representarse gráficamente en forma de grafo

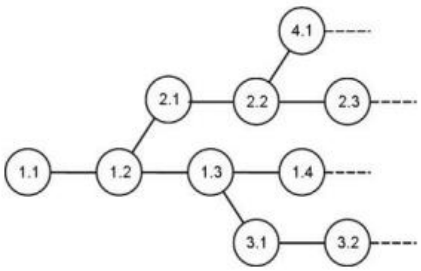


VARIANTE

Una variante es una versión de un ítem de configuración (o de la configuración) que evoluciona por separado.

Las variantes representan configuraciones alternativas.

Un producto de software puede adoptar distintas formas (configuraciones) dependiendo del lugar donde se instale. Por ejemplo, dependiendo de la plataforma (máquina + S.O.) que la soporta, o de las funciones opcionales que haya de realizar o no.



CONFIGURACIÓN DE SOFTWARE

Conjunto de todos los ítems de configuración con su versión específica. Define una foto de los ítems de configuración en un momento de tiempo determinado.

Una configuración es el conjunto de todos los componentes fuentes que son compilados, sus documentos y la información de la estructura que definen una versión del producto a entregar. La configuración de un software es la sumatoria **de todos los ítems de configuración que tiene en un momento determinado**, equivale a una instantánea o una foto de todos los ítems de configuración con su versión en un momento del tiempo.

LINEA BASE

Es un conjunto de ítems de configuración que han sido construidos y revisados formalmente, de manera que pueden ser tomados como referencia para demostrar que se ha alcanzado cierto nivel de madurez en ellos, y que sirve como base para desarrollos posteriores. Esto es lo mismo que decir que es una configuración de software que ha sido formalmente revisada y aprobada, que sirve como base para desarrollos futuros.

Este conjunto de ítems debe tener una referencia única, y esto se hace a través de "tags".

Se acuerdan parámetros para determinar qué se considera o qué debe cumplir los ítems de configuración para considerarse como línea base.

Si se desea cambiar una línea base, existe un **protocolo formal de control de cambios**, dirigido por el Comité de Control de Cambios, que permite definir el procedimiento a seguir para manejar peticiones de cambios, y en caso de aceptarse, acordar nuevamente los parámetros y comunicar los cambios a todo el equipo, para que tomen la nueva línea base como modelo a seguir.

¿PARA QUÉ SIRVE?

Fundamentalmente es para tener un punto de referencia, pero también sirve para hacer Rollback o saber qué se pone en producción. Nos permite saber cuál era la última situación estable en un momento de tiempo y cómo se fue evolucionando. Permiten ir atrás en el tiempo y reproducir el entorno de desarrollo en un momento dado del proyecto.

Existen dos tipos:

1. De especificación: (Requerimientos, Diseño) son las primeras línea base, dado que no cuentan con código. Podría contener el documento de especificación de requerimiento.
2. Operacionales: contiene una versión de producto cuyo código es ejecutable, han pasado por un control de calidad definido previamente. La primera línea base operacional corresponde con la primera Release. Es la línea base de productos que han pasado por un control de calidad definido previamente.

Release: Entrega de un sistema que se libera para su uso a los clientes u otros usuarios de la organización.

RAMA (BRANCH)

Es un conjunto de ítems de configuración con sus correspondientes versiones, que permiten bifurcar el desarrollo de un software, por varios motivos, ya sea experimentación, resolución de errores en el desarrollo o para desarrollar un mismo software para distintas plataformas.

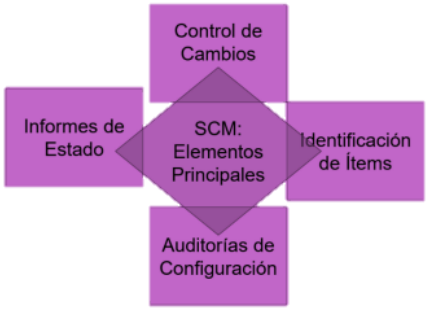
Integración de ramas o merge: Se da cuando se integran dos ramas, para fusionar la configuración de los ítems que la conforman con sus correspondientes versiones en cada una de ellas.

Una buena práctica es mantener la rama principal como la versión estable, y fusionar los cambios hacia ella. En caso de no integrarse a la Main, las ramas deberían descartarse

ACTIVIDADES RELACIONADAS A LA GESTION DE CONFIGURACIÓN

Estas actividades/elementos son las cosas que contienen las secciones de un plan de gestión de configuración. Recordemos que la gestión de configuración también se debe planificar.

Todas las actividades que vamos a mencionar se realizan en ambas metodologías (tradicional y agil), menos las auditorías, ya que el concepto de ser auditado y controlado por alguien externo, no es compatible con una metodología ágil pura.



IDENTIFICACIÓN DE ÍTEMS DE CONFIGURACIÓN

Esto implica una identificación unívoca para cada ítem, donde en el equipo se definirán **políticas y reglas de nombrado y versionado** para todos ellos. También, se debe definir la **estructura del repositorio**, y la **ubicación de los ítems de configuración dentro de esa estructura**.

Además, implica una definición cuidadosa de los componentes de la línea base.

Esta identificación y documentación proveen un camino que une todas las etapas del ciclo de vida del software, lo que permite a los desarrolladores **controlar y velar por la integridad del producto**, como así también a los clientes **evaluar esa integridad**.

También se debe tener en cuenta al momento de identificar los ítems, la duración de su integridad, ya que difieren en función del tiempo que es necesario mantenerlos.

Se clasifican en:

- **Ítems de producto**: tienen el ciclo de vida más largo, y se mantienen mientras el producto exista. Ejemplo: documento de arquitectura, casos de uso, código, manual de usuario y de parametrización, casos de prueba, una ERS, los casos de prueba, la base de datos.
- **Ítems de proyecto**: el plan de proyecto, el listado de defectos encontrados, tienen un ciclo de vida de proyecto. Un plan de iteración, un Burn - Down Chart, se mantiene durante una iteración. La duración impacta también en el esquema de nombrado, plan de proyecto. Ejemplo: Los ítems de configuración a nivel de proyecto, el plan de proyecto y el cronograma.
- **Ítems de iteración**: Conocer el ciclo de vida de un ítem permite establecer la nomenclatura de este. Ejemplo: planes de iteración, cronogramas de iteración, reporte de defectos.

CONTROL DE CAMBIOS

Tiene su origen en un requerimiento de cambio a uno o varios ítems de configuración que se encuentran en una **línea base.**

Es un Procedimiento formal que involucra diferentes actores y una evaluación del impacto del cambio.

Una vez definida la línea base, no es posible cambiarla sin antes pasar por un proceso formal de control de cambios. Es decir que el control se hace sobre los ítems de configuración que pertenecen a la línea base, ya que todos los trabajadores del software tienen a dichos ítems como referencia.

COMITÉ DE CONTROL DE CAMBIOS

Este proceso es llevado a cabo por el comité de control de cambios, el cual se reúne para autorizar la creación y cambios sobre la línea base. Forman parte de dicho comité los interesados en evaluar y en enterarse del cambio, decidiendo si lo aceptan o no. El líder del proyecto, el analista, el arquitecto e incluso el cliente pueden constituirlo. Realmente la integración del comité depende de la propuesta del cambio.

ETAPAS:

1. Se recibe una propuesta de cambio sobre una línea base determinada, no sobre un ítem.
2. El comité de control de cambios realiza un análisis de impacto del cambio para evaluar el esfuerzo técnico, el impacto en la gestión de los recursos, los efectos secundarios y el impacto global sobre la funcionalidad y la arquitectura del producto.

3. En caso de que se autorice la propuesta de cambio, se genera una orden de cambio que define lo que se va a realizar, las restricciones a tener en cuenta y los criterios para revisar y auditar.
4. Luego de realizado el cambio, el comité vuelve a intervenir para aprobar la modificación de la línea base y marcarla como línea base nuevamente, es decir que realiza una revisión de partes.
5. Finalmente se notifica a los interesados los cambios realizados sobre la línea base.

El control de cambios permite tener una trazabilidad entre los ítems de configuración, y ante un cambio saber cuáles ítems están afectados por este.

AUDITORÍAS DE CONFIGURACIÓN

Son controles autorizados a realizar por el equipo de desarrollo en un momento determinado, donde un auditor externo al equipo (independiente y objetivo) analiza las líneas base, las cuales permiten "congelar" y analizar en un momento determinado cuál es el estado del software, y si se están cumpliendo todas las pautas que plantea esta disciplina de SCM.

En metodologías ágiles, esta es la única actividad de la disciplina SCM que no se soporta por la filosofía.

Es objetiva cuando hay **independencia de criterio**, por lo que el auditor no debe tener ningún condicionamiento respecto de lo auditado.

Es independiente cuando no depende jerárquica, funcional ni salarialmente del equipo o elemento a auditar.

La auditoría de configuración se hace mientras el producto se está construyendo y su objetivo es que el producto **tenga calidad e integridad**. Se entiende que lo más barato es prevenir.

La auditoría de configuración complementa a la revisión técnica y consiste en revisar si se están realizando las tareas tales como se planificaron y especificaron en el plan de gestión de configuración.

Las auditorías se realizan sobre una línea base, es decir, requiere la existencia de un plan de gestión de configuración y de la línea base a auditar.

Auditoría física de configuración (PCA): Asegura que lo que está indicado para cada ICS en la línea base o en la actualización se ha alcanzado realmente.

Auditoría funcional de configuración (FCA) Evaluación independiente de los productos de software, controlando que la funcionalidad y performance reales de cada ítem de configuración sean consistentes con la especificación de requerimientos.

La auditoria de configuración sirve a dos procesos básicos:

VALIDACIÓN	VERIFICACIÓN
Se encarga de asegurar que cada ítem de configuración resuelva el problema apropiado, es decir, lo que el cliente necesita.	Implica asegurar que un producto cumple con los objetivos definidos en la documentación de líneas base. Todas **las funciones son llevadas a cabo con éxito** y los test cases tengan status "ok" o bien consten como "problemas reportados" en la nota de reléase. Es decir, que se cumpla lo definido para cada ítem de configuración en la documentación de una línea base.

Las auditorías permiten visualizar si se están satisfaciendo los requerimientos y si se ha cumplido la intención de la línea base anterior. Ante un análisis, el auditor puede detectar defectos y ajustar correcciones

REPORTES E INFORMES DE ESTADO

Provee un mecanismo para mantener un registro de cómo evoluciona el sistema, y dónde está ubicado el software, comparado con lo que está publicado en la línea base. Esto permite mantener a todo el equipo informado sobre la última línea base, y que se trabaje en base a su última versión, y no sobre una versión obsoleta.

Estos reportes incluyen todos los cambios que han sido realizados a las líneas base durante el ciclo de vida del software. Normalmente esto consta de grandes unidades de datos, por lo que se utilizan herramientas automatizadas por una computadora.

El objetivo principal es asegurarse que la información de los cambios llegue a todos los involucrados.

El reporte más conocido es el de inventario, el cual provee una copia del contenido del repositorio con la estructura de directorios.

Permite responder a preguntas como: ¿Cuál es el estado del ítem? ¿La propuesta de cambio fue aceptada o rechazada por el comité? ¿Qué versión de ítem implementa un cambio de una propuesta de cambio aceptada? ¿Cuál es la diferencia entre dos versiones de un mismo ítem?

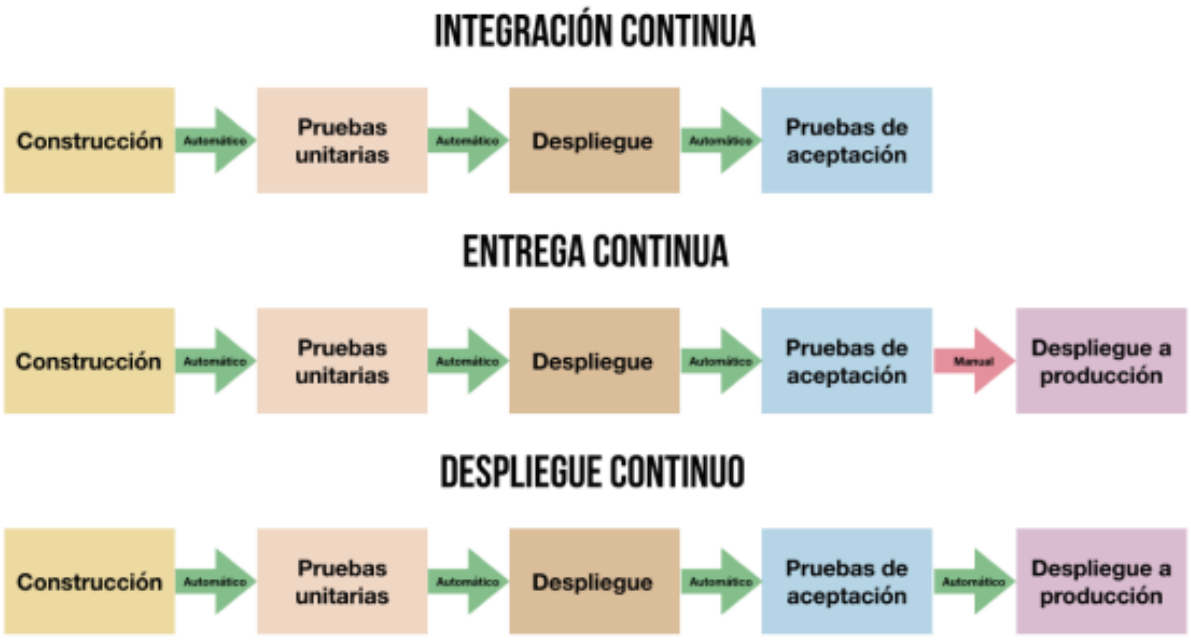
PLANIFICACIÓN DE LA GESTIÓN DE CONFIGURACIÓN DE SOFTWARE

Este plan se debe confeccionar al inicio de un proyecto, y existen diferentes estándares donde se expresa cómo planificar:

- Reglas de nombrado de los ítems de configuración
- Herramientas para utilizar en SCM
- Roles e integrantes del Comité de Control de Cambios
- Procedimientos formales de cambios
- Procesos de auditoría
- Estructura del repositorio
- SCM para software externo (opcional
- Cómo se hará el control de cambios
- Registros que deben mantenerse
- Tipos de documentos.

Debe hacerse tempranamente, se deben definir los documentos que se van a administrar y no debe quedar ningún producto del proceso sin administrarse.

EVOLUCIÓN DE LA GESTIÓN DE CONFIGURACIÓN DE SOFTWARE



CONTINUOUS INTEGRATION

Es una práctica de desarrollo que promueve que los desarrolladores adopten la costumbre de integrar su código a un repositorio compartido varias veces al día. Cada integración de código es luego verificada por una serie de pruebas automatizadas, permitiéndole al equipo **detectar problemas de manera temprana**, ya que al integrar el código al repositorio de manera frecuente resulta más fácil detectar y corregir los errores.

La integración continua es asegurar que el software pueda ser desplegado en cualquier momento, es decir, que el **código compile y que sea de calidad**. Cada desarrollador en su entorno de trabajo realiza pruebas unitarias (en la medida de lo posible, automatizadas) desarrollando con algo que se llama **TDD** (desarrollo conducido por pruebas) y cuando terminó ese componente de código y lo probó y sabe que funciona, lo sube a un repositorio de integración. Así, la versión del producto está en condiciones de ir a las pruebas de aceptación de usuario sin problemas.

CONTINUOUS DELIVERY

Es una disciplina de desarrollo de software en la que el software se construye de tal manera que puede ser **liberado en producción en cualquier momento**.

Suma la **automatización de las pruebas de aceptación** y entonces el producto **ya está listo para desplegarlo a producción**.

No implica necesariamente que se libere cada vez que hay un cambio, sólo ocurre si el responsable de negocio, el Product Owner de turno, decide pasar a producción, esto quiere decir que hay un componente «humano» a la hora de tomar la decisión, pero, en cualquier caso, la versión está lista de inmediato.

Esto implica, entre otros, que se prioriza que la versión esté en un estado en el que pueda ser puesta en producción.

La **integración continua** es prerrequisito de la **entrega continua**. Con esto se refiere a que los artefactos producidos en el servidor de integración continua son puestos en producción con solo un click.

Hay que asegurarse de tener un despliegue automatizado, para que cada puesta en producción no lleve demasiado tiempo, lo que hará que las puestas puedan llegar a hacerse más seguidas, generando que lo que se despliegue no tenga demasiados cambios y el riesgo de que algo se rompa disminuya.

El software debe estar siempre en un estado de **"entregable"**, es decir, el software buildea, el código se compila y los test pasan.

CONTINUOUS DEPLOYMENT

Consiste en poner en el ambiente de producción del usuario final el producto. A diferencia de la entrega continua, en el despliegue continuo **no existe la intervención humana para desplegar el producto en producción**. Para esto, se utilizan **pipelines** que contienen una serie de pasos que deben ejecutarse en un orden determinado para que la instalación sea satisfactoria.

El propósito de las estrategias es que sea transparente para el usuario que pusiste en producción una nueva versión del producto. Se recomienda hacer el despliegue a poco tiempo de haber trabajado con los cambios en el código, pues **un error en producción un día después de haberlo hecho significa que todavía nos acordamos de lo que hicimos y será más fácil de resolver**.

Esto permite que los cambios de código sean entregados a los usuarios finales con mayor rapidez y frecuencia, lo que reduce el tiempo de lanzamiento de nuevas funcionalidades y mejora la experiencia de los usuarios.

ESTRATEGIAS DE DEPLOYMENTS

- **CANARY DEPLOYMENT**
- **BLUE/GREEN DEPLOYMENT**

SCM EN AMBIENTES ÁGILES

En las metodologías ágiles la gestión de configuración cambia el enfoque, ya que la misma es de utilidad para los miembros del equipo de desarrollo y no viceversa.

En orden con la caracterización de los equipos ágiles, decimos que la gestión de configuración posibilita el seguimiento y la coordinación del desarrollo en lugar de controlar a los desarrolladores.

Los equipos ágiles son auto organizados, por lo que las Auditorías de configuración no son una actividad propia de la gestión de configuración en los ambientes ágiles. Todos los procesos definidos establecidos en la gestión de configuración del enfoque tradicional son relativizados en agile. Por ejemplo, no existe un comité para el proceso forma de control de cambios

Entre otras tareas, SCM en Agile:

- Hace seguimiento y coordina el desarrollo en lugar de controlar a los desarrolladores.
- Responde a los cambios en lugar de tratar de evitarlos.
- La automatización debe ser aplicada donde sea posible. Esto colabora con la entrega frecuente y rápida de software. (Continuos Integration)
- Coordinación y automatización frecuente y rápida.
- Eliminar el desperdicio - no agregar nada más que valor.
- Provee documentación Lean y trazabilidad.
- Provee retroalimentación continua sobre calidad, estabilidad e integridad

Es responsabilidad de todo el equipo y vamos a tener tareas de SCM embebidas en las demás tareas requeridas para alcanzar el objetivo del sprint.

EN EL RESUMEN FALTA SCRUM (Se ve con la guía) MÉTRICAS AGILES Y PLANIFICACION DE RELEASES Y SPRINTS