

## **RESUMEN INGENIERIA Y CALIDAD DE SOFTWARE**

### **Ingeniería de software**

Disciplina de la ingeniería cuya meta es el desarrollo costeable de sistemas de software, este es abstracto, intangible y fácil de modificar. La ingeniería de software comprende todos los aspectos de la producción de software, desde las etapas iniciales de la especificación del sistema hasta el mantenimiento de este después que se utiliza.

#### **Retos que afronta la Ingeniería del software**

- De Heterogeneidad (consiste en desarrollar técnicas para construir software que pueda operar como un sistema distribuido en redes con distintos tipos de computadoras, con sistema en otros lenguajes, etc)
- De la Entrega (reducir los tiempos de entrega para sistemas grandes y complejos sin comprometer la calidad del sistema)
- De la Confianza (desarrollar técnicas que demuestren que los usuarios pueden confiar en el software)

¿Cuál es la diferencia entre ingeniería de software e ingeniería de sistemas?

La ingeniería en sistemas se ocupa de los aspectos basados en computadoras del desarrollo de sistemas, incluye hardware, software y procesos de ingeniería. La ingeniería de software es parte de este proceso general.

Función del ingeniero: Los ingenieros aplican teorías, métodos y herramientas de la manera más conveniente siempre tratando de descubrir soluciones a los problemas teniendo en cuenta que deben trabajar con restricciones financieras y organizacionales por lo que buscan soluciones contemplando estas restricciones

### **Crisis del Software**

El término crisis del software hace referencia a un conjunto de hechos relativos al software, planteados en la conferencia de OTAN en 1968 por Friedrich Bauer, quien recalco la dificultad para generar software libre de defectos, fácilmente comprensibles y que sean verificables.

La noción de la ingeniería de software surgió en 1968 debido a la crisis de software causada por la introducción de nuevas computadoras basadas en circuitos integrados que provocó la posibilidad de desarrollar software más grande y complejos, esto llevó a que los proyectos se atrasaran, otros se cancelaban, se sobrepasaban los presupuestos, como consecuencia se generaban software de mala calidad y desempeño pobre que requerían intensas actividades de mantenimiento.

Entonces fue evidente que para crear un software de esta magnitud tomar un enfoque informal no era adecuado, se necesitaban nuevas técnicas y métodos para controlar la complejidad de estos sistemas.

## **¿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. De manera más correcta, podemos definir al software como conocimiento empaquetado, a distintos niveles de abstracción.

Si bien el código instalado en una computadora funcionando es software, el concepto es más amplio e incorpora también otros aspectos. Podemos decir que la definición de ese producto también es software, una base de datos también, y el diseño de la base de datos también, y una user story también, y así entre otros. Si no tenemos en cuenta aspectos como estos, estamos perdiendo una gran cantidad de información que es la que nos permitirá llegar a tener ese programa.

### TIPOS BÁSICOS DE SOFTWARE

|                               |   |
|-------------------------------|---|
| <b>SYSTEM SOFTWARE</b>        | 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   |
| <b>UTILITARIOS</b>            | 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. |
| <b>SOFTWARE DE APLICACIÓN</b> | 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.  |

Tipos de Productos:

- Productos Genéricos:** sistemas producidos por una organización y que se venden al mercado abierto a cualquier cliente, la especificación es controlada por quien lo desarrolla
- Personalizados o A Medida:** son sistemas requeridos por un cliente en particular, la especificación es desarrollada y controlada por la organización que compra el software

### Buen Software

Los atributos reflejan la calidad del software, no están directamente asociados con lo que el software hace, más bien reflejan su comportamiento durante su ejecución y en la estructura y organización del programa fuente y en la documentación asociada.

- Mantenibilidad (debe escribirse de tal forma que pueda evolucionar para cumplir necesidades de cambio)
- Confiabilidad (no debe causar daños físicos o económicos en el caso de una falla del sistema)
- Eficiencia (no debe malgastar los recursos del sistema, como memoria o procesamiento)
- Usabilidad (debe ser fácil de usar por el usuario, con interfaz apropiada y su documentación)

## SOFTWARE ≠ MANUFACTURA

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.

Hay que hacer estimaciones. El problema es que estimar tiene asociado un factor de probabilidad, entonces puede salir mal. Cuando se gestiona un proyecto para construir software dejando fijo el alcance, se deben estimar en base a estos los costos y el tiempo, pero surge el problema de que el alcance varía, entonces hace variar a estos también y como resultado termina habiendo mas tiempos y costos que los presupuestados

- △ **QUE LA VERSIÓN FINAL DEL PRODUCTO NO SATISFAGA LAS NECESIDADES DEL CLIENTE**.

Tenemos que tener el foco en que que tenemos que entregar al cliente es valor, no características de software. Las características de software son el medio por el cual le entregamos valor de negocio. Y para ello es importante tener en cuenta a todos los involucrados, todos aquellos que tienen algo para decir acerca del producto.

- △ **PROBLEMA EN LA ESCALABILIDAD Y/O ADAPTABILIDAD DEL SOFTWARE**. Agregar una funcionalidad en otra versión es casi una misión imposible.

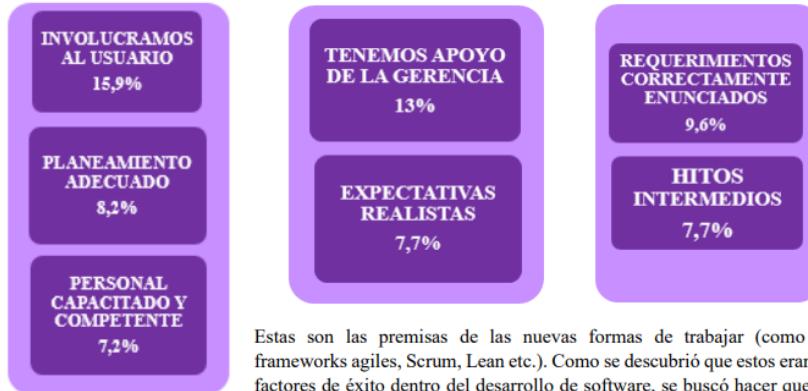
La calidad del sistema no debería ser una variable decisión. No debemos dejar de construir un producto siguiendo los principios, buenas prácticas y documentación.

- △ **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 así 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 vayan “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 de 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.



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.

△ **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.

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.

Las actividades del proyecto son ejecutadas por personas y automatizadas con tecnología. Es importante tener en cuenta la participación de las personas al realizar las actividades ya que el desarrollo de software es una actividad humano intensiva, el software es desarrollado por personas.

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

Encontramos dos tipos de procesos:

### PROCESO DEFINIDO

Son aquellos procesos que plantean el paso a paso todo lo que es necesario saber. Buscan tener definidos ampliamente el paso a paso que vamos a hacer en cada momento. Se ocupan de desagregar un conjunto de actividades y decirnos tarea por tarea quien la va a hacer, cuando la va a hacer, que entradas necesita, que artefactos va a generar, y en qué orden. Nos cuentan paso a paso que es lo máximo, la mayor cantidad de opciones posibles que deberíamos tener en cuenta para hacer un producto de software. Tienen la intención de ser repetibles o reproducibles, y buscan brindar visibilidad, de donde estamos parados, qué es lo que hicimos y cuánto falta para terminar. Es una necesidad natural, para saber qué avance tenemos de la situación y si vamos a poder terminar. El líder de proyecto se siente más cómodo si tiene visibilidad del proyecto por el que tiene que dar la cara. Los procesos definidos tienen controlado qué se hace paso a paso, y si saben que algo en otro proyecto tardó tanto tiempo, este más o menos en las mismas circunstancias nos va a llevar tanto.

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

Los empíricos lo primero importante que hay que tener presente es que no hay ninguno que sea un proceso completo, como el PUD intenta cubrir todo lo que hace falta para hacer soft.

Son aquellos que tienen un fuerte arraigo en la experiencia, pero no siempre es fácil de conseguir, porque hay que basarse en algo anterior y hay que generarla desde un principio. Surgen en contraposición de los procesos definidos

Cada equipo al iniciar un trabajo decide basado en la experiencia que es lo que quiere hacer, cuándo y cómo. El empirismo se basa en ciclos de entrega cortos para poder generar retroalimentación que sirva como experiencia para evolucionar y seguir avanzando. Los procesos empíricos dicen que la experiencia es aplicable al mismo equipo, este equipo puede generar su propia experiencia en este proyecto, en este contexto particular para los distintos ciclos, pero la experiencia de ese equipo no se puede extrapolar en otros equipos, proyectos, que es lo que si esperan los procesos definidos. Los procesos definidos esperan repetibilidad.

Los frameworks (Scrum, Kanban) no están completos. Son lineamientos, buenas prácticas de alguna parte o partes de un proceso. Pero como son empíricos la idea es que el equipo cierre, defina y crea su propio proceso para el proyecto.

## 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

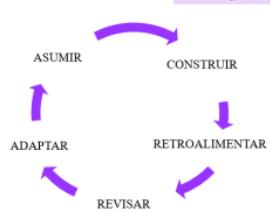
Confiamos en que las variables no pueden ser definidas o controladas en su totalidad, estamos dentro de un ambiente complejo donde quizás no todo lo que está en juego está bajo nuestro control y esto nos implica la necesidad de **adaptación** para poder garantizar la producción de un software de calidad.

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)

Los procesos empíricos trabajan basados en una hipótesis, asumen que el proceso puede funcionar de alguna manera, hacen algo, obtienen realimentación, revisan y si algo no salió bien, adaptan y comienzan el ciclo nuevamente, ganando experiencia.

## 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 progresan. Esto no dice que tanto los productos como los proyectos tienen ciclos de vida.

El ciclo de vida no es lo mismo que el proceso. Para un proyecto determinado, se define un ciclo de vida a utilizar en el proceso. El proceso define el paso a paso para alcanzar un objetivo, el ciclo de vida define cómo se van a realizar dichas actividades, su orden, duración y (si así lo establece el ciclo de vida), repeticiones. Ciclo de vida y proceso se complementan pero son asuntos diferentes.

Proceso define lo que hay que hacer. Ciclo de vida es más abstracto, dice en que orden se van a ejecutar, cuanto de cada una, en que momento llevar a cabo cada tarea definida del proceso.

El ciclo dice el cuando y cuanto. El proceso el que y como

Ciclo de vida tiene el proyecto, y también el producto, son distintos.

#### **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.

El ciclo del proyecto son las fases/etapas, que son estados de evolución del proyecto que determinan que cosas se pueden hacer y en que orden, para entregar el resultado o servicio

El ciclo del proyecto es todas las etapas del proyecto desde que empieza hasta que 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.

El ciclo de vida del producto son todos los estados por los que pasa desde que nace por una idea hasta que se descontinua

El ciclo del producto dura mas que el ciclo del proyecto

Tenemos varios proyectos a lo largo del ciclo de vida del producto

Dentro del ciclo de vida del producto se ejecutan varios ciclos de vida de proyecto que dan como resultado una nueva versión del producto.

Hay tres tipos básicos de ciclos de vida para los proyectos de desarrollo de software:

- Secuencial: se realiza una etapa después de otra y no tienen generalmente retorno. Una actividad no inicia hasta que ha terminado la anterior. Ej, en cascada.
- Iterativo e incremental: El sistema se desarrolla como una serie de versiones (incrementos) y cada una añade funcionalidades a la versión anterior. Se saca ventaja de lo aprendido a lo largo del desarrollo anterior incrementando entregables. Permite mejorar y ajustar el proceso
- Recursivos: Se inicia con algo en forma completa, como una subrutina que se llama a si misma e inicia nuevamente. Se presenta un prototipo que va mejorando con cada vuelta. No hacen entregas parciales como hacen los iterativos, pero tampoco son que hasta el final no vas a tener nada como cascada. Están al medio. Hacen énfasis en la gestión de riesgos. En cada vuelta evaluar si conviene seguir con el

proyecto o cambiar. Define, evalúa riesgos, construye y entrega. Ej, el espiral, toma todo el producto y lo refina en distintas vueltas pero el producto finalmente para operar y poner en producción esta al final. Se justifica con productos muy complejos, muy riesgos, que no tiene sentido liberar el producto por partes (ej poner un satélite en orbita)

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.



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 proceso se define en término de roles porque para cada proyecto, las personas asumen uno o más roles

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.

Medio de gestión de organización para obtener como resultado un producto o servicio (soft en este caso)

El proceso se instancia en el proyecto, este “le da vida” al proceso. El proceso dice teóricamente lo que hay que hacer, pero el proyecto define las tareas basadas en el proceso, estableciendo cuales tareas definidas en el proceso se van a realizar y cuales no, y arma así el conjunto de tareas que se van a llevar a cabo en el proyecto. Las tareas se ejecutan concretamente en el ámbito proyecto.

Características de un proyecto:

1. Orientación a un objetivo: (esto permite que los proyectos sean únicos, para ello el objetivo tiene que ser claro y alcanzable), tengo que poder definir hacia donde voy y poder establecer/medir cuando alcancé ese objetivo para poder dar por finalizado el proyecto; estos objetivos son los que guían el proyecto, no deben ser ambiguos y deben ser también alcanzables. Los proyectos tienen un objetivo único, es decir que es distinto de los productos resultantes de otros proyectos. Cada resultado de un proyecto, es distinto del resultado de otro proyecto.

2. Duración limitada: son temporarios, cuando se alcanza el/los objetivo/s, el proyecto termina; una línea de producción no es un proyecto. El proyecto tiene un inicio y un fin.

3. Tareas interrelacionadas basadas en esfuerzos y recursos para alcanzar el objetivo: tienen precedencia, tienen vínculos entre ellas, que una tarea depende de otra y que a su vez se le asocian/designan esfuerzos y recursos lo cual hace que la gestión de proyectos sea una tarea compleja. Dividimos todo el trabajo a realizar en tareas que tienen una relación en término de que algunas pueden hacerse en conjunto o unas dependen de otras. La definición de que tareas hay que realizar se obtienen del proceso.

4. 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

Producto es el software, conocimiento empaquetado. Incluye todo. Los manuales de usuario son soft, los manuales de configuración también. Todo lo que sale como resultado de la ejecución de las tareas de un proyecto, todo es software

## 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 Management)** 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.

para ello voy a administrar todos los recursos que me dieron para el proyecto, organizar el trabajo de la gente afectada y hacer un seguimiento de si las cosas se están dando como estaban planificadas. Tenemos una figura de líder de proyecto que es el responsable de hacer todo lo mencionado.

Administrar un proyecto incluye:

- Poder definir los requerimientos, el alcance del producto.
- Establecer objetivos claros y alcanzables.
- Adaptar las especificaciones, planes y el enfoque a los diferentes intereses de los involucrados (stakeholders)

La gestión tradicional habla sobre la necesidad de que el líder del proyecto encuentre un equilibrio entre 3 factores que actúan como restricciones del proyecto. Esta triple restricción plantea que en un proyecto se van a tener restricciones de:

### 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 una **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 defini 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 intimamente relacionadas de manera tal que, si modifco 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)**.

### LÍDER 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.

El líder es quien se relaciona con los demás involucrados; es conductor, le dice a la gente que es lo que tiene que hacer, para cuando lo tiene que hacer, y el equipo recibe las asignaciones del trabajo y trabaja. Informa cuando ya está terminado, informa cuánto se

demoró, informa el porcentaje de avance y si hay una desviación respecto a los planes informarlo al líder para que tome ciertas correcciones. Dentro de este enfoque, como líderes tenemos la responsabilidad, estimar, planificar, asignar trabajo a la gente y hacer seguimiento de lo que está pasando. No solemos tener trabajo técnico, ya que, si tuviéramos trabajo técnico y de gestión, se suele priorizar el técnico dejando de lado el de gestión.

Como líderes de proyecto debemos tener un plan de proyecto, un mapa que nos va a guiar durante todo el proyecto. Este es el artefacto resultante de la planificación.

**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.

#### 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 qué 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 incluimos:

- 1. DEFINICIÓN DEL ALCANCE:** Es importante hacer la diferenciación entre proyecto y producto de software.

| ALCANCE DEL PROYECTO   | ALCANCE DEL PRODUCTO   |
|--|--|
| Es todo el <b>trabajo</b> y solo el trabajo que debe hacerse para entregar el producto o servicio con todas las características y funciones especificadas en el objetivo.<br>El cumplimiento del alcance de proyecto se mide contra el <b>Plan de Proyecto</b> . | Son todas las características que pueden incluirse en un producto o servicio.<br>El cumplimiento del alcance de producto se mide contra la <b>Especificación de Requerimientos</b> (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 qué 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 qué 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:

La estimación no se aborda directamente en la planificación del proyecto y el cronograma sino que sirve de input para la planificación. La definición de compromisos, fechas y objetivos tienen que ver con otras variables y no solo con lo estimado.

Estimar es predecir el trabajo, esfuerzo que será necesario realizar en un momento donde no se tiene conocimiento sobre lo que se estima. Por definición no es precisa, acarrea riesgo es incertidumbre propio del inicio del proyecto

Los errores en las estimaciones provienen de:

- caos del proyecto
- la incertidumbre
- bajo nivel de conocimiento de la capacidad de la empresa
- falta de claridad en el alcance
- no utilización de técnicas adecuadas

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

**Recursos Críticos:** Recursos particulares de un proyecto. A esas cosas que nos hacen falta para desarrollar, ejemplo, si desarrollamos un software que maneja sensores de alarma para incendio, el desarrollador necesita el sensor y el de testing también. No es propio de todos los proyectos pero es necesario tener en cuenta la posibilidad de su existencia.

Primero se determina que es lo que se debe construir (tamaño) y su alcance. Es difícil medir el tamaño del software. Luego, se estima el esfuerzo, es decir, cuantas horas lineales necesito para construir lo que definí en el tamaño; todavía no pensamos quienes lo van a hacer ni qué disponibilidad de recursos tengo, simplemente es una medida en cantidad de horas de esfuerzo. El siguiente paso es llevarlo a calendario, a esas horas lineales las distribuyo en las tareas que hay que realizar que tienen que ver con estas horas y en las secuencias que tienen estas tareas entre sí. Luego se estima el costo, que esta directamente vinculado con la mano de obra; en función de la cantidad de horas y del calendario voy a poder determinar el costo de lo que tiene que ver con la mano de obra, pero también el costo de otras variables, como el uso de herramientas, disponibilidad de almacenamiento, disponibilidad de las distintas instalaciones que puedo llegar a necesitar. Por último, estimo los recursos críticos, cuales son aquellos momentos del tiempo y recursos que son críticos y que me pueden generar algún inconveniente en el caso de no contar con ese recurso.

#### Técnicas de estimación

- Contar funcionalidades / features / requerimientos / historias de usuario
- Basadas en la experiencia
  - Datos históricos sobre procesos anteriores
  - Juicio experto: una persona experta estima → quien realiza la tarea.
  - WIDEBAND Delphi (Variante del juicio experto) → Conjunto de personas informadas hacen una estimación y discuten hasta llegar a un acuerdo.
    - Poker planning
- Basadas en los recursos
- Basados en el mercado
- Basados en los componentes del producto o proceso de desarrollo
- Métodos algorítmicos

#### 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

Todos los riesgos tienen asociados una probabilidad de ocurrencia y un impacto (en tiempo o dinero). Asociando una escala numérica para ambas variables y multiplicando obtenemos la exposición al riesgo

Ante los riesgos se pueden tomar actitudes reactivas, de reaccionar a estos si ocurren, o actitudes proactivas, de determinar de antemano que hacer en caso de que estos ocurran.

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 está 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.

Hacen falta las métricas porque son números, definiciones cuantitativas, y son buenos porque son objetivos.

Las métricas pueden clasificarse de acuerdo a su dominio en:

|                             |  |
|-----------------------------|--|
| <b>MÉTRICAS DE PROCESO</b>  | 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.<br><br><b>Ej.: Porcentaje de proyectos que terminan en término y porcentaje de proyectos que terminan fuera de término.</b> |
| <b>MÉTRICAS DE PROYECTO</b> | 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.<br><br><b>Ej.: Comparación del tiempo calendario real con el planificado.</b>   |
| <b>MÉTRICAS DE PRODUCTO</b> | Miden cuestiones/características que tienen una relación directa con el producto que estamos construyendo.<br><br><b>Ej.: Métricas de defectos, todas las métricas de tamaño</b>   |

¿Cada cuánto se toman las métricas? Depende de las características y tiempos del proyecto. Deben ser lo suficientemente frecuentes para detectar desvíos a tiempo.

¿Qué se debe hacer con las métricas? Se desea saber si el proyecto va a poder cumplir con los objetivos propuestos. Si las métricas obtenidas indican que no, debo tomar acciones para corregirlo e intentar cumplirlo

Los proyectos se atrasan de un día a la vez, si puedo corregir los desvíos de mi proyecto en el momento adecuado, estoy a tiempo de poder cumplir mi objetivo.

## 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.

Valentina Lascano R. 10

Tiene que ver con ir comparando lo planificado y lo real; la línea perfecta entre lo planificado y lo real no existe, suele estar algo dibujado

### FACTORES PARA EL ÉXITO

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.

### CAUSAS DE FRACASOS

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 agilismo es un movimiento que se gestó desde los desarrolladores.

Se juntaron hace aproximadamente 20 años un grupo de referentes de la industria de software en un hotel de Utah a discutir y generaron un acuerdo al que llamaron Manifiesto ágil.

Este manifiesto ágil es un compromiso de todas las personas involucradas 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 El movimiento ágil tenía como objetivo lograr un equilibrio entre hacer las cosas como si no fuéramos profesionales, y tampoco hacer las cosas en el otro extremo llegando al punto de cumplir con el proceso en lugar de entregar un producto de calidad.

#### **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.

Es el que nos permite a nosotros crecer como equipo y transformar el conocimiento implícito, el conocimiento de cada uno de los miembros del equipo, en conocimiento explícito, conocimiento que sea del equipo

- 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.

Valentina Lascano R. 13

---

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. SATISFACCIÓN DEL CLIENTE**

Nuestra mayor prioridad es satisfacer al cliente a través 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 satisface 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 información dentro de un equipo de desarrollo de

#### **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 frameworks agiles apuntan a ciclos de vida iterativos de duración fija, y si no llegue con todo lo previsto en la iteración entregó 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 TÉCNICA**

La excelencia técnica y el buen diseño mejoran la agilidad. La calidad del producto no es negociable, 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.**

## 6. COMUNICACION DIRECTA

El método más eficiente y efectivo de comunicar información 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

en comunicación con los sujetos 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 autonomía y capacidad de decisión en el proceso de diseño y desarrollo en lugar de tener un enfoque jerárquico y rígido. El agilismo rompe totalmente con el enfoque tradicional y expresa que los más aptos para definir las características del producto, 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 **Triángulo Ágil**

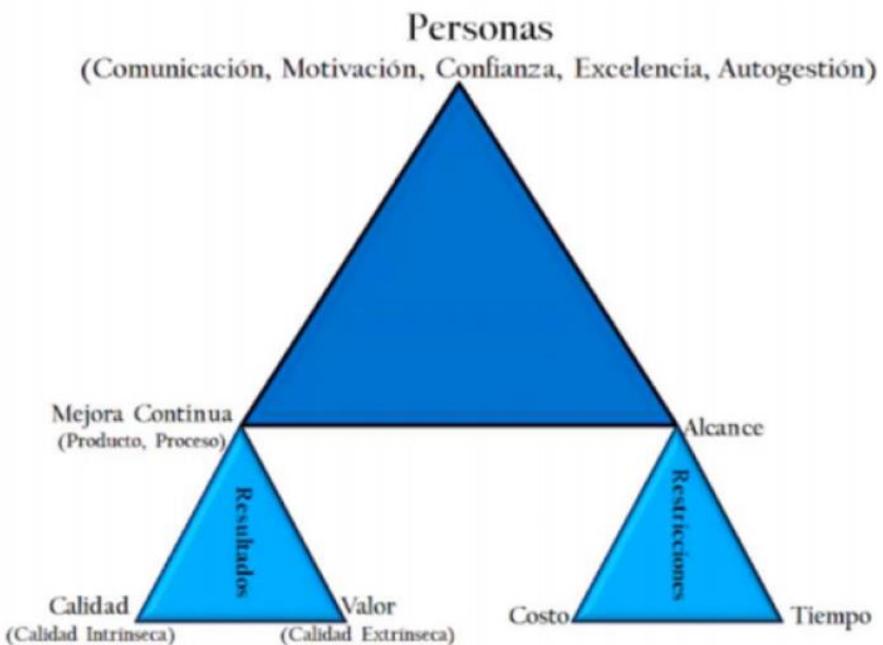
Si planteamos un ciclo de vida iterativo para un proyecto definido, sabemos que vamos a dejar fijos los requerimientos y vamos a definir las iteraciones en base a porciones de funcionalidad (CU) a desarrollar en cada iteración. Ahora bien, las iteraciones que plantea un framework ágil, se centran en 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?**”

Desde el punto de vista de los requerimientos, con el enfoque ágil, debemos en lugar de dejar fijo los requerimientos o el alcance, y a partir de ahí derivar recursos y tiempo (como lo plantea el enfoque tradicional), vamos a dejar fijo el tiempo (con iteraciones de duración fija, las que SCRUM llama Sprint) y los recursos (tener un equipo de trabajo con una determinada capacidad) asignado a trabajar, en base a esto, defino cuánto del producto funcionando puedo construir en este tiempo y así se acuerda el alcance, luego se repite

para la siguiente iteración. Teniendo en cuenta esta forma de pensar, se puede hablar de gestión ágil de los requerimientos

### **Triangulo Ágil Modificado:**



El triángulo de hierro hace mucho foco en proyecto, en los recursos y en cumplir. Highsmith apunta con el triángulo ágil que lo de valor es el soft funcionando para el cliente. El triángulo de hierro va a ser considerado pero se le va a dar menor importancia, vamos a darle mucha más importancia al valor, a la calidad del producto, al valor que el producto tiene para el cliente (esa es la calidad extrínseca, la que es visible) y por otro lado a las características de calidad que nosotros le ponemos al producto para que este sea mejor y satisfaga. La versión modificada del triángulo apunta a un producto de software que pone a las personas en el lugar mas importante que tiene que estar, porque las personas son las que tienen que resolver el software. El éxito o el fracaso depende de la gente. Personas con estas características (comunicación, motivación, confianza, excelencia, autogestión). La intención de mejora continua de todo el producto y el proceso, y finalmente tengamos en cuenta las variables que se usan para gestionar el trabajo (las del triángulo de hierro). En vez de hacer foco en que las tres dimensiones tienen que cerrar tenemos que hacer foco

en la gente involucrada y en entregarle a esa gente un producto de calidad que se vaya mejorando y evolucionando en forma continua.

## 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.

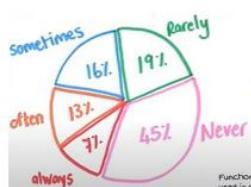


Se asume que:

Los cambios van a existir todo el tiempo de forma constante. Tenemos que tener una mirada de los que están involucrados, es decir, todos los que tienen algo para decir del producto (Stakeholders). El product owner es el representante de todos los stakeholders. Hay una realidad de que el cliente dice si está satisfecho o no, cuando tenga algo para ver. Es por ello que estamos apurados para mostrarle algo al cliente.

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 **sólo 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.

Para poder entregar un producto de calidad en tiempo y forma, es importante tomar conciencia que la realidad en el mercado nos indica que es un porcentaje muy pequeño de características que realmente se usan, siendo el 20% del total del producto. Un error muy común que cometen los desarrolladores es "hacer las cosas ya que estamos" pero que nadie solicitó. Al hacer esto, terminamos con una gran cantidad de características que nos hicieron perder tiempo y dinero, y además nos hicieron quedar mal con el cliente porque nos hicieron perder tiempo y entregamos el producto tarde, y que como nadie lo pidió nadie lo reconoce, ni tampoco lo usa ya que como dijimos anteriormente, es un porcentaje muy pequeño de características el que se utiliza.

**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 qué 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**

El producto se define conforme haga falta, just in time, diferiendo las decisiones lo mas que se pueda, tomando decisiones en el ultimo momento responsable. Se usa para evitar desperdicios. Desperdicio es especificar reqs que después cambian y yo invertí mucho tiempo en especificarlos. El enfoque just in time dice que el producto no va a estar especificado al 100% desde el principio sino que vamos a ir encontrando reqs y describiendo lo que nos hace falta, conforme nos haga falta. Just in time, ni antes ni después. Porque si llega antes es un desperdicio, pero si llega tarde es un problema porque alguien ya tomó una decisión, y puede pasar que el producto no haga lo que tiene que hacer y haga otra cosa

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

### TIPOS DE REQUERIMIENTOS

|                          |  |
|--------------------------|--|
| <b>DE NEGOCIO</b>        | Son los de más alto nivel, nos referimos a cuales son los requerimientos en términos de la visión del negocio  |
| <b>DE USUARIO</b>        | Tienen que ver concretamente con los requerimientos de usuario final   |
| <b>FUNCIONALES</b>       | Podríamos relacionarlo 1 a 1 con los casos de uso en la gestión tradicional  |
| <b>NO FUNCIONALES</b>    | 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) |
| <b>DE IMPLEMENTACIÓN</b> | 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.

Tenemos que trabajar juntos técnicos y no técnicos, entendiendo las necesidades y el negocio, si no entendemos el negocio no podemos construir un soft que ayude. Tenemos que convertirnos en expertos del dominio. No importa el enfoque, hay que entender el negocio. Y luego junto con los usuarios descubrir cual es la mejor forma de satisfacer las necesidades. Y acá aparece el producto backlog. El dueño de esto es el po. Esto se usa para determinar en una específica iteración que es lo que vamos a entregar. Le entregamos

la característica, la liberamos y obtenemos feedback que nos ayuda a refinar el pb, ver que hay que agregarle, que sacarle, si esta bien priorizado.

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

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

El rector es entregar al cliente soft funcionando

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

Toda la gestión ágil se arma para estar preparados para que vengan reqs de cambio en cualquier momento

**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

Es que el equipo esta conformado por alguien del negocio, y no se suplanta, no se contrata una persona externa, no puedo inventar una persona que sepa del negocio. Agiles muchas veces fracasa porque el cliente no esta dispuesto a participar en los términos que se necesita que participe.

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

La técnica anterior no es mala, solo se implementa mal. No enviar las cosas por mail al cliente sino citarlo a una reunión y explicarle.

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

Acá el enfoque ágil dice demoles responsabilidad al equipo porque esta formado, capacitado, dejemos que las decisiones técnicas respecto a como construir el producto, y que arq va a tener, salgan del equipo

## USER STORIES

**Una historia de usuario es una descripción corta de una necesidad que tiene el usuario respecto del producto de software.**

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.

**Son porciones verticales:** Las US se cortan verticalmente incluyendo todas las capas a nivel arquitectónico (Interfaz de usuario, lógica de negocio, base de datos), para entregarle al usuario algo de software funcionando que le sirva y pueda trabajar con eso. Si las corto horizontalmente no entrego valor al cliente. 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)

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

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.

### **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 definimos 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.

La conversación se considera la parte más importante de las User Stories. Esta no queda guardada en ningún lado, porque segúnn los principios ágiles el mejor medio de comunicación es la comunicación cara a cara.

### **CONFIRMATION**

Las pruebas de usuario que se identifican necesarias para hacerle después a la funcionalidad una vez realizada, para que el product owner me acepta la característica de software construida a partir de la US

#### **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.

Son las pruebas que se entiende va a hacer el Product Owner y los usuarios, cuando se entregue la funcionalidad lista. Están relacionadas con los criterios de aceptación. Se describe lo que se tiene que probar, estos no son casos de prueba, son los títulos. Que cosas deberían poder hacerse y que cosas no se aceptan porque contradicen los criterios de aceptación.

### **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.

Siguen el siguiente formato.

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

As WHO I want  
WHAT so that WHY

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 explicitamente 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**.

#### **CRITERIOS DE ACEPTACIÓN**

- Definen límites para la user y son la base para definir las pruebas de aceptación
- Muestra lo necesario para que la user provea valor según lo que establece el Product Owner
- Genera una visión compartida con el equipo
  - Guía a los equipos en las pruebas (desarrolladores y testers)
  - Ayuda a determinar cuando parar de agregar funcionalidades

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 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

#### **DEFINICIÓN DE LISTO o CRITERIO DE READY**

Una medida de calidad que construye el equipo para poder determinar que la user está en condiciones de entrar a una iteración de desarrollo. La User está lista cuando cumple con la definición de listo (o Ready) impuesta por el equipo.

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.

|                         |   |
|-------------------------|---|
| <b>I: INDEPENDIENTE</b> | 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                        |
| <b>N: NEGOCIALBE</b>    | 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 |

la user tiene que estar escrita en términos de que necesita el usuario no como lo vamos a implementar.

|                     |   |
|---------------------|---|
| <b>V: VALUABLE</b>  | Debe tener un valor concreto para el cliente  |
| <b>E: ESTIMABLE</b> | Tengo que poder definir cuánto esfuerzo me va a llevar hacer la US, para ayudar al cliente a armarse un ranking basado en costos. |
| <b>S: SMALL</b>     | La US tiene que entrar en una Sprint. Para saber si es small, tengo que poder estimarla.  |

Tiene que ver con no hacer el trabajo a medias, que algo sea pequeño depende mucho del equipo

|                     |   |
|---------------------|---|
| <b>T: TESTEABLE</b> | Tengo que poder demostrar que fueron implementadas. |
|---------------------|---|

se debe poder demostrar que la user se implementó cumpliendo los criterios de aceptación definidos

#### **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 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

Son un tipo especial de US que se producen por la incertidumbre que la misma presenta, la cual imposibilita que pueda ser estimada y por lo tanto no cumple con la definición de listo. Una vez

resuelta la incertidumbre, la Spike se convierte en una o más US. |

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.

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.

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.

## SOFTWARE CONFIGURATION MANAGEMENT (SCM)

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.

Transversal al proyecto porque en el proyecto ejecutas la gestión de conf, se hacen mientras estas con un proyecto activo, pero para el producto a lo largo de su ciclo de vida. Si no tenes un proyecto activo no estas creando ítems nuevos, no hay ningún factor que te haga modificar la gestión de configuración.

Dentro de la ing de soft hay distintos tipos de disciplinas agrupadas por la función.  
Disciplinas técnicas que se ocupan del producto, disciplinas de gestión que se ocupan del proyecto, y disciplinas de soporte o protectoras que intentan prevenir que la calidad sea comprometida. La SCM es de soporte

Disciplinas que conforman la ingeniería de software:

- Técnicas: ayudan a construir el producto. Ej: Análisis, diseño, implementación, prueba, despliegue, etc.
- De gestión: planificación, monitoreo, control
- De soporte: gestión de configuración, métricas, aseguramiento de la calidad

La SCM es una disciplina de soporte o protectora

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:

|  |
|--|
| <b>PÉRDIDA DE COMPONENTES</b>                      |
| <b>PÉRDIDA DE CAMBIOS</b>                          |
| <b>(LA VERSIÓN DEL COMPONENTE NO ES LA ÚLTIMA)</b> |
| <b>REGRESIÓN DE FALLAS</b>                         |
| <b>DOBLE MANTENIMIENTO</b>                         |
| <b>SUPERPOSICIÓN DE CAMBIOS</b>                    |
| <b>CAMBIOS NO VALIDADOS.</b>                       |

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, iconos, etc, etc.

ítem de configuración es cualquier cosa que se pueda guardar en un disco.

## VERSIÓN

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

Las versiones se identifican únicamente. 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



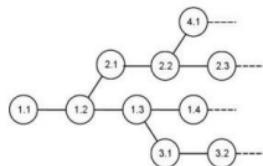
El software es volátil. Es importante incorporar un aspecto: su versión, la cual refiere a la evolución de cada IC por separado . Cada ítem con su versión en un momento de tiempo. Porque cambia la versión. La SCM busca identificar cada ítem únicamente en un momento de tiempo, eso lo hace a través de su versión. Un archivo solo con su versión es un ítem de configuración en un momento de tiempo.

## 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.

## 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

## 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.

### 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 GESTIÓN 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 única 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 existe. 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 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.**

lo conforma el equipo, no un auditor externo

#### 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 auditoría 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.<br><br>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**

La física se hace primero. Si esa sale bien se hace la funcional. La funcional ve consistencia con los reqs, si lo que estas creando se cumple. En este momento ayuda la rastreabilidad

## 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.

## 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 requisito 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 bulde, el código se compila y los test pasan.

es una extensión de la integración continua, que implica la automatización del proceso de construcción, prueba y empaquetado del software en cada cambio de código, y la entrega de estos paquetes a un entorno de pruebas o de producción para su evaluación

## **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.

## 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

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.

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 super 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.

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.

(horas ideales, sin desperdicio)

Luego de estimar el esfuerzo, estimo el **CALENDARIO**, que es justamente calendarizar el esfuerzo el calendario contesta cuando va a estar esto. Hay que ver que trabajos se pueden hacer en paralelo y cuales dependen de otro. Se calendariza para ver cuando voy a poder entregar esta versión del producto

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

### Recursos Críticos:

Recursos particulares de un proyecto. A esas cosas que nos hacen falta para desarrollar, ejemplo, si desarrollamos un software que maneja sensores de alarma para incendio, el desarrollador necesita el sensor y el de testing también. No es propio de todos los proyectos pero es necesario tener en cuenta la posibilidad de su existencia.

## 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 completa 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.

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 fórmula.  $(O+4H+P)/6$ .
- 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.

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:



Si las estimaciones se utilizan como compromisos son muy peligrosas y perjudiciales para cualquier organización.



Lo más beneficioso en las estimaciones es el “proceso de hacerlas”.



La estimación podría servir como una gran respuesta temprana sobre si el trabajo planificado es factible o no.



La estimación puede servir como una gran protección para el equipo.

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.

El foco de las estimaciones agiles es que se hace foco en la certeza, no en la precisión.

Enfoques tradicionales hacen énfasis en la precisión. Difícil de cumplir. Para llegar a esa precisión hubo que hacer suposiciones

Al igual que con los reqs no hacer un esfuerzo en estimar todo el producto sino cuando hace falta

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 la 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 puntuá 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:

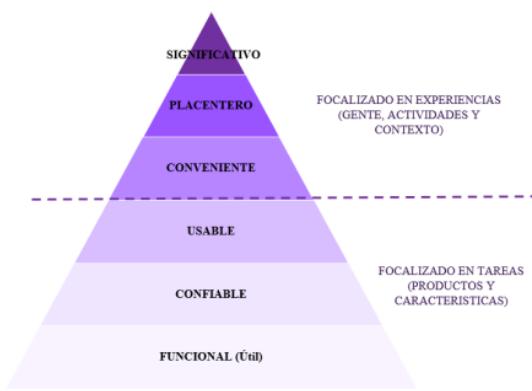
### ¿POR QUÉ CREAMOS PRODUCTOS?

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 desecharmos 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.

## **INGENIERÍA DE SOFTWARE – TESTING**

La prueba de software está en el contexto del aseguramiento de la calidad del software, pero cuando hablamos de prueba de software **no nos referimos ni a Control de Calidad de Proceso, ni a Control de Calidad de Producto**; es un tema en sí mismo.

No es aseguramiento, es parte de la disciplina, pero es control de calidad lo que hace. Una actividad de validación y verificación. Esos son los dos procesos principales. Verifica que el sistema funciona correctamente y valida que el sistema sea el sistema correcto. Hago eso con un producto construido, lo que detecto con el testing son defectos, que se trasladaron. Es control de calidad.

**TESTING DE SOFTWARE:** es un proceso **destructivo** cuyo objetivo es encontrar defectos (No asegurarse que el software funcione) ya que **nosotros asumimos que los hay**; implica ir con una actitud negativa para demostrar que algo es incorrecto.



Partiendo de la base que un test/prueba es exitoso/a cuando encuentra defectos, podemos concluir que:

**un desarrollo exitoso nos conduce a un test/prueba no exitoso/a, porque no encuentra defectos**

**Mundialmente, en el costo de un software confiable, el Testing se lleva entre el 30% al 50% del mismo.**

En la definición de la actividad estamos aceptando que el soft tiene defectos. El soft lo hacen las personas y las personas cometen errores. Errores de distinta gravedad pero de cualquier manera son errores. Si no se encuentran errores no se hizo suficiente testing o no se hizo bien.

El testing no certifica nada. No se puede asegurar que son todos los defectos que hay. Hay una relación costo-beneficio, así que en un punto hay que dejar de probar, y no siempre es suficiente.

### **ERROR vs DEFECTO**

|  |   |
|--|---|
| El error se descubre a partir de técnicas específicas que me permiten encontrar los mismos dentro de la misma etapa en la que estoy trabajando | Los defectos nos demuestran un error no detectado que se trasladó a una etapa siguiente |
|--|---|

En el testing encontramos **defectos**, ya que justamente se encuentran cosas incorrectas que se realizaron en la etapa de implementación que es la etapa anterior.

Más adelante en el contexto de revisiones técnicas e inspecciones vamos a ver más a fondo que son los errores y como encontrarlos.

Obviamente siempre es mejor encontrar errores antes que defectos.

Cuando hablamos de defectos encontrados durante el testing, debemos considerar dos aspectos que nos van a definir el accionar que vamos a realizar sobre estos:

#### SEVERIDAD

Tiene que ver con la gravedad del defecto que encontré.

Un defecto puede ser **bloqueante** y no permitirme seguir con el caso de prueba, **crítico** se refiere a un defecto que compromete la ejecución del caso de prueba y así sucesivamente va disminuyendo desde **mayor, menor**, hasta **cosmético** (que podría ser alguna cuestión de sintaxis, ortografía, visualización, etc.)

#### PRIORIDAD

Urgencia que tenemos para resolver este defecto.

Se podría intuir que, según la severidad de los defectos, va a haber tal o cual prioridad, pero esto no siempre es así.

Esto va a depender del contexto en el cual nos encontramos, es por esto la importancia de tener los dos aspectos identificados.

La prioridad la define el cliente

Desde el punto de vista de mi negocio que tan rápido lo necesito. Pensamos que los bloqueantes son urgentes, los de prioridad baja son cosméticos, pero no necesariamente es así. Puede ser bloqueante pero de una parte que no está en producción.

### NIVELES DE TESTING

Uno de los aspectos importantes a la hora de definir las pruebas que vamos a hacer es identificar un esquema o manera de hacer testing que tiene que ver con los niveles de prueba. Como lo indica su nombre, se trata de **subir escalones** o niveles, e ir abarcando más y más cosas para probar a medida que vamos subiendo.

**PRUEBAS UNITARIAS:** son aquellas en donde pruebo un componente individual, algo acotado que tiene que ver con el desarrollo que estoy realizando. Son respecto a aspectos puntuales y aislados del proyecto que se está realizando. Al probar componentes individuales y de forma independiente, normalmente se ejecutan por el mismo desarrollador. En estas pruebas se encuentran errores más que defectos. Se encuentra en el límite. Es muy fácil de automatizar.

**PRUEBA DE INTEGRACIÓN:** implica integrar los componentes ya probados en las pruebas unitarias. Se integrar para ver su funcionamiento conjunto. Normalmente requieren de un tester. Como estamos buscando verificar que las partes aisladas funcionan bien en conjunto, se hace una integración de manera incremental para lograr una identificación más correcta de los errores.

Las pruebas de integración dependen. Hay una variación con respecto a quien hace las pruebas de integración. En organizaciones las hacen los desarrolladores y terminan con un build que se puede poner en producción. Otras automatizan con integración continua, nosotros obtenemos permanentemente todo el tiempo un build que está en condiciones de ser la entrada al nivel de prueba del sistema.

Las de integración dependen de cada equipo, cada proyecto funcione y del contexto de la org decidir como se van a llevar adelante las pruebas de integración. A veces la gente de testing no está capacitada para hacer la integración, así que la suelen hacer los desarrolladores

**PRUEBAS DE SISTEMA:** Son más amplias, no solo pruebo la integración entre dos componentes, sino el sistema en toda su escala. Busca asegurarse que el sistema en su totalidad funcione de manera satisfactoria. Empiezan a impactar otras cuestiones que tienen que ver no solo con requerimientos funcionales, sino también con requerimientos no funcionales. El **ambiente** de prueba tiene que ser lo más parecido al entorno de producción.

Las pruebas de sistema también se las llama pruebas de versión en ciclos iterativos. Porque si estamos con ciclos iterativos, cada vez que generamos una versión no es el sistema completo pero a esa versión le vamos a hacer el nivel de prueba de sistema, y es a la que vamos a aceptar después. Prueba de sistema y de versión son mas o menos lo mismo. Hacemos el testing de los reqs de esa porción del producto.

Pruebas de sistema apuntan al principio del testing que dice que un desarrollador no debería probar su código sino otro. Muchos casos no es por mala intención, sino que a uno le cuesta hacer una separación y lograr la objetividad necesaria para encontrar defectos. Se recomienda que sea no la misma persona, sino incluso dentro del equipo cruzarse el código, que la user que programó uno la testeé otro. En este nivel hay que respetar esto, no vamos a tener la actitud destructiva que necesitamos.

**PRUEBAS ACEPTACIÓN:** La mayoría de las veces son ejecutadas por el usuario final o el cliente. El foco no es encontrar defectos, sino que la meta es distinta, es establecer confianza en el sistema. Independientemente de que el usuario puede encontrar defectos, no es el objetivo principal. En estas pruebas, el ambiente también tiene que ser lo más parecido posible al entorno de producción.

Finalmente las pruebas de aceptación de usuario, las hace el usuario. El usuario tiene que estar presente, porque él es el que pidió, el que sabe lo que quiere y el que tiene que hacer la aceptación de la versión del producto. Es el último nivel de prueba. La prueba de sistema se hace en el workflow de prueba, y la de aceptación en el de despliegue.

Unitarias en implementación, sistema en pruebas y de aceptación en despliegue. Las de integración depende de cada equipo, cada proyecto funcione y del contexto de la org decidir como se van a llevar adelante las pruebas de integración

## AMBIENTES PARA CONSTRUCCIÓN DEL SOFTWARE

Los ambientes son los lugares en donde se trabaja para el desarrollo de software. Los ambientes para la construcción del software se refieren a los diferentes entornos utilizados en el ciclo de vida del desarrollo de software. Cada ambiente cumple un propósito específico y se utiliza en diferentes etapas del proceso de desarrollo y despliegue

### AMBIENTE DE DESARROLLO

El ambiente de desarrollo es donde los desarrolladores crean, prueban y depuran el software. Es un entorno local o en red utilizado por los desarrolladores para escribir y probar el código antes de integrarlo con el resto del sistema. Los programadores pueden utilizar herramientas de desarrollo, depuración y pruebas para garantizar que el software cumpla con los requisitos establecidos. Este ambiente suele ser flexible y permite a los desarrolladores experimentar y probar nuevas ideas sin afectar los entornos de producción. Las pruebas unitarias se llevan a cabo en el ambiente de desarrollo

### AMBIENTE DE PRUEBA

El ambiente de pruebas es donde se llevan a cabo pruebas exhaustivas del software desarrollado. Aquí se realizan pruebas de integración, pruebas funcionales, pruebas de rendimiento y otras pruebas relevantes para verificar que el software cumpla con los requisitos establecidos y funcione correctamente. El ambiente de prueba suele ser una réplica o tener características parecidas al entorno de producción, pero sin afectar a los usuarios finales y sin TODAS las características (ya que es caro tener ambientes iguales al de producción). Se utilizan conjuntos de datos y configuraciones similares a las del entorno de producción para garantizar una prueba más precisa del software. Las pruebas de integración se realizan en el ambiente de pruebas

**Comentado [1]:** no se si las pruebas de integración se hacen en ambiente de pruebas

El entorno de prueba debe corresponder al entorno de producción tanto como sea posible para reducir al mínimo el riesgo de incidentes debidos al ambiente específicamente y que no se encontraron en las pruebas.

El ambiente de prueba es para las pruebas de sistema. Preparados con datos de prueba y todas las herramientas que te hacen falta para hacer las pruebas de sistema. La hacen los testers.

Prueba separado de desarrollo. Ambientes separados por una cuestión de injerencias y separación de intereses. Los desarrolladores no tienen acceso al ambiente del testing. Ellos integran y generan un build, y testing tiene que ejecutar un ciclo de prueba, ejecutando casos de prueba definidos para una versión, y termina cuando haces todas las pruebas o cuando una te bloquea y no te deja seguir. No hay que arreglar el error y después seguir, tiene que empezar el ciclo de prueba de nuevo. Porque cuando un desarrollador corrige un error introduce mas, por eso es ambiente separado.

#### AMBIENTE DE PRE-PRODUCCIÓN

El ambiente de preproducción, también conocido como entorno de puesta en escena o entorno de calidad, es donde se realiza una prueba final del software antes de su lanzamiento en producción. Aquí se simulan las condiciones del entorno de producción y se realizan pruebas de último minuto, como pruebas de estrés, pruebas de carga y pruebas de seguridad. El objetivo es validar y verificar que el software esté listo para ser implementado en el entorno de producción sin problemas significativos. Las pruebas de sistema se llevan a cabo en el ambiente de preproducción

La idea es tener una infraestructura de hard y de comunicaciones lo mas parecido al de producción. Mientras mas nos acercamos al ambiente de producción mas grave y mas caros.

la pre producción es donde se deberían hacer las pruebas de aceptación, y cuando pasaran debería ponerse el soft en producción que es cuando el soft ya esta disponible para que los usuarios lo usen

El problema es que no siempre se incorpora el ambiente de pre-producción por costo. Las pruebas de aceptación no se pueden hacer en prueba porque después hay que desplegarlo y ahí se hacen pruebas también.

#### AMBIENTE DE PRODUCCIÓN

El ambiente de producción es donde el software está en funcionamiento y es accesible a los usuarios finales. Es el entorno real en el que el software se utiliza para realizar las tareas y funciones para las que fue diseñado. Este ambiente suele ser altamente controlado y está configurado para ser escalable, seguro y confiable. Se implementan medidas de respaldo y recuperación ante desastres para garantizar la disponibilidad continua del software. Es el ambiente en el que el software está funcionando. Las pruebas de aceptación se llevan a cabo en el ambiente de preproducción o en un entorno similar al de producción (en el ambiente de prueba)

**Comentado [2]:** las pruebas de sistema se llevan en el ambiente de pruebas

## CASO DE PRUEBA

Tiene que ver con un conjunto de condiciones o variables que nos van a permitir determinar si el software está funcionando correctamente o no. La buena definición de casos de prueba nos ayuda a reproducir defectos. El caso de prueba tiene que ver con ejecutar una serie de pasos o acciones en una determinada funcionalidad determinando completamente con qué valores voy a hacer la ejecución para ver si el software me va a dar los resultados esperados o no.

Un caso de prueba además de especificar cuál es la acción a ejecutar, me debe especificar con qué datos debo realizarlo. Si el caso de prueba está bien definido ayuda a identificar defectos y controlarlos, solucionarlos. Esto es algo fundamental para el éxito en el testing.

El caso de prueba es el artefacto principal de testing. Son la unidad de trabajo del testing. Un CP es un escenario concreto, con datos específicos, que intenta probar un conjunto de situaciones o de criterios de aceptación o de condiciones de prueba. Para poder ejecutar un caso de prueba tenemos que tener bien especificado qué es lo que queremos probar, bien especificada la condición de seteo del sistema, en qué estado tiene que estar el sistema para poder probar lo que quiero probar, y trabajar mucho y muy bien la conformación del set de datos que se van a usar para poder ejecutar las pruebas, y bien identificado el resultado esperado.

Un defecto que no se puede reproducir no es un defecto, porque como se lo reportas al desarrollador. Tener un caso de prueba nos ayuda porque tenemos registrado el paso a paso de lo que hice para poder reportarlo

Para esto, el éxito de un buen testing está sustentado fuertemente en que tan bien diseñados estén los CP y que tan bien conformadas las BD para hacer pruebas.

Dentro de los casos de prueba, y su identificación, se nos presenta el inconveniente de que no podemos identificar o definir caso de prueba de manera infinita, debo tener algún mecanismo para lograr definir la **menor cantidad de casos de prueba**, pero que estos me permitan cubrir el testing completamente, o de la manera más completa posible.

La menor cantidad de CPS que sean lo más eficientes y efectivos posibles. Quiero la menor cantidad de CPS que me permitan probar la mayor cantidad de cosas.

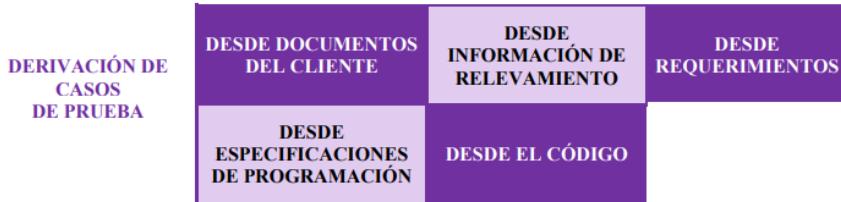
Hay una combinatoria de condiciones posibles tan grande que no se puede tener una cobertura del 100% del testing, porque es inviable en términos de costo y tiempo. La cobertura del 100% del testing es imposible, entonces uno tiene que intentar reducir la cantidad de CPS para reducir el tiempo y el costo que hacen falta para ejecutarlos después.

Para esto vamos a usar **ESTRATEGIAS DE PRUEBA**, vamos a conducir el esfuerzo hacia lo importante.

Normalmente los casos de prueba trabajan sobre la idea de que es imposible probar todo, entonces tienes que elegir. Hay que elegir métodos de prueba que ayudan a definir casos de prueba que sean eficientes y sean la menor cantidad posible. El propósito que tienen es achicar la cantidad de casos de prueba sin reducir la calidad del testing. Buscar donde están los problemas, suelen estar en el borde al límite.

Cuando vamos a definir casos de prueba, vamos a intentar apuntar a valores límites o esquinas, es decir, a los lugares que son más propensos de tener errores/defectos. Probamos en los "límites".

Los casos de prueba pueden derivar de distintos lugares, **cuanta más documentación tengamos, más fácil será especificar casos de prueba.** Cuanta menos especificación tengamos, vamos a tener que describir con mayor detalle los casos de prueba



El lugar ideal para sacar info para diseñar los casos de prueba para hacer verificación y validación es desde los reqs. Sacarlo del código no porque no podes hacer validación si diceñas un cp desde el código, no tenes forma de saber si el producto esta haciendo lo que tiene que hacer o esta haciendo otra cosa si lo derivas desde el código. Desde el código lo único que vas a poder hacer es una verificación, no hay errores de no se clava o da mensajes de error. Si estamos probando lo que el cliente espera recibir o no no lo vamos a saber. Para eso lo derivamos desde los cu o las user, y entonces voy a ver si el producto se esta comportando como dicen los reqs o no, y estoy haciendo verificación y validación. Si derivo desde el diseño detallado o desde el código no tengo forma de saber si el producto esta haciendo lo que el cliente esperaba o esta haciendo otra cosa. Por eso las tendencias apuntan a fortalecer toda la actividad de testing y que el detalle en lugar de estar en los reqs este en el testing, porque en el testing si no hay detalle no se puede probar. Por eso las user tienen descripción muy corta de la funcionalidad y hacen mucho hincapié en criterios de aceptación y las pruebas que tenemos que diseñar. De lo que se menciona ahí hay que derivar los cp correctos. Menos detalle en los reqs y mas en las pruebas, porque de esa forma hay que mantener actualizados, porqje si los cp no están actualizados no se puede probar. En cambio en los reqs la tendencia es que uno hace la versión de reqs y sale la primer versión y cambian los reqs y la ers no se actualiza. Con los cp no podes porque si no están actualizados no se puede ejecutar la prueba.

Si los requerimientos están definidos de una manera muy general y poco especificados/detallados, deberemos especificarlos ya que, si es el caso contrario, la derivación de los requerimientos hacia los casos de prueba es directa.

Muchas veces se pueden utilizar los casos de prueba como especificación de requerimientos.

Lo más importante con respecto a la generación de casos de prueba es saber que ninguna técnica es completa, las distintas técnicas atacan distintos problemas y por ende es esencial combinarlas para complementarlas ventajas de cada una y tener una especificación más detallada. Los casos de prueba dependen del código a testear y se debe tener en cuenta la conjectura de defectos. Debemos ser sistemáticos y documentar las suposiciones sobre el comportamiento o el modelo de fallas.

## CONDICIONES DE PRUEBA

Tienen que ver con el contexto del sistema que tengo que tener en cuenta para poder hacer los casos de prueba. Son la reacción esperada de un sistema frente a un estímulo particular, y estando el mismo constituido por las distintas entradas.

Es algo que se enuncia ANTES de definir los casos de prueba, defino las condiciones y en función de eso defino los casos. Una condición de prueba debe ser probada por al menos un caso de prueba.

### CAJA BLANCA



A diferencia de caja negra, lo que hace caja blanca es mirar dentro del código, revisar su estructura y ver si dentro de esa definición se encuentra algo que vaya a dar un resultado no esperado.

Se basan en el análisis de la estructura interna del software o un componente del software.

Hay distintos métodos. Algunos de ellos son:

- COBERTURA DE ENUNCIADOS O CAMINOS BASICOS
- COBERTURA DE SENTENCIAS
- COBERTURA DE DECISIÓN
- COBERTURA DE CONDICIÓN
- COBERTURA MÚLTIPLE

### CAJA NEGRA



Se llama de caja negra, ya que nosotros definimos en los casos de prueba cuales son las entradas y el resultado esperado tiene que ver con cómo se comporta el sistema frente a esas entradas, lo que nosotros desconocemos, es específicamente como se llega a esa salida. Frente a determinadas entradas esperamos obtener ciertas salidas, si las obtenemos el caso de prueba pasa.

Los métodos de caja negra pueden dividirse en dos.

#### - BASADOS EN ESPECIFICACIONES

Algunos métodos son:

- o Partición de Equivalencias
- o Análisis de valores límites

#### - BASADO EN LA EXPERIENCIA

Tiene que ver con el testing que hace el tester experimentado y sabe exactamente donde ir para encontrar los defectos. Se basa en la experiencia.

Algunos son:

- o Adivinanza de Defectos
- o Testing Exploratorio

Comentado [3]: falta decisión/condición en caja blanca

### MÉTODO PARTICIÓN DE EQUIVALENCIAS

Este método analiza cuales son las diferentes condiciones externas, todas las entradas o salidas posibles, que van a estar involucradas en el desarrollo de una funcionalidad y para cada condición externa, analizo cuales son los subconjuntos de valores posibles que pueden tomar las mismas que producen un resultado equivalente. Sigue los siguientes pasos:

#### 1. Identificar las clases de equivalencia (válidas y no válidas)

- Rango de valores continuos
- Valores discretos
- Selección simple
- Selección múltiple

ingresar un valor"; la **división en particiones de equivalencias** se basa en cuales son los resultados posibles que vas a tener.

Identificamos la entrada y luego elegimos/determinamos cuales son esos subconjuntos de valores que producen que cualquier valor que yo tome de ese subconjunto va a producir un resultado equivalente.

**CLASE DE EQUIVALENCIA:** subconjunto de valores que puede tomar una condición externa para el cual, si yo tomo cualquier miembro de ese subconjunto, el resultado de la ejecución de la funcionalidad es equivalente.

## 2. Identificar los casos de prueba

Tomo de cada una de esas condiciones externas, una clase de equivalencia en particular y voy a elegir un valor representativo para poder conformar mi caso de prueba.

En cuanto a la prioridad, van a tener prioridad alta aquellos que nos garanticen encontrar mayor cantidad de defectos y que además nos garante la salud en sí de la funcionalidad; en la prioridad alta siempre vamos a encontrar los caminos felices, los de prioridad baja son aquellos relacionados con validaciones, con valores no ingresados.

Las precondiciones son **todo el conjunto de valores o de características que debe tener mi contexto para que yo pueda llevar adelante este caso de prueba en particular**.

### TÉCNICA DE CAJA BLANCA

- Cobertura de enunciados o caminos básicos
- Cobertura de sentencias
- Cobertura de decisión
- Cobertura de condición
- Cobertura de decisión/ condición
- Cobertura múltiple

Comentado [4]: en el de vale faltaba 1

#### COBERTURA DE ENUNCIADOS O CAMINOS BÁSICOS

Busca poder garantizar que todos los caminos independientes que tiene nuestra funcionalidad, lo vamos a recorrer por lo menos una vez. Si diseñamos un conjunto de casos de prueba, que corresponda a la cobertura de enunciados o caminos básicos, podemos garantizar que nuestro código ha sido ejecutado pasando por todos los caminos independientes.

Para la prueba del camino básico:

- Se requiere poder representar la ejecución mediante grafos de flujo
- Se calcula la complejidad ciclomática
- Dado un grafo de flujo se pueden generar casos de prueba.

#### CÁLCULO DE LA COMPLEJIDAD CICLOMÁTICA

$$M = E - N + 2 * P \quad M = \text{Número de regiones cerradas} + 1 \text{ (forma visual)}$$

- ✓ M = complejidad ciclomática
- ✓ E = Número de aristas del grafo (flechitas que unen nodos)
- ✓ N = Número de nodos del grafo
- ✓ P= Número de componentes conexos, nodos de salida.

Pasos del diseño de pruebas mediante el camino básico:

1. Obtener el grafo de flujo
2. Obtener la complejidad ciclomática del grafo de flujo
3. Definir el conjunto básico de caminos independientes
4. Determinar los casos de prueba que permitan la ejecución de cada uno de los caminos anteriores
5. Ejecutar cada caso de prueba y comprobar que los resultados son los esperados.

#### **COBERTURA DE SENTENCIAS**

Una **sentencia** es cualquier instrucción tales como asignación de variables, invocación de métodos, mostrar un mensaje, lanzar una excepción.

El objetivo de la cobertura de sentencias es buscar la cantidad mínima de casos de prueba que me permiten pasar/ejecutar/evaluar/recorrer todas las sentencias.

No todos los casos van a ser tan sencillos como asignaciones de variables numéricas, sino que pueden representar el estado de un pedido, selección de una ciudad, existencia de una patente.

#### **COBERTURA DE DECISIÓN**

Una **decisión** es una estructura de control completa, son los paréntesis del if. Se nos pueden presentar estructuras de control anidadas como estructuras de control independientes.

El objetivo de la cobertura de decisión es buscar la cantidad mínima de casos de prueba que me permiten ejecutar todas las decisiones y verificar que funcionen correctamente: ver que vaya tanto para la rama del true, como para la rama del false, independientemente si tengo sentencias en una rama y no en la otra.

Cuando las decisiones no están anidadas y sus condiciones no se relacionan entre si (ejemplo de arriba) con dos casos de prueba me alcanzan, ya que independientemente que la primera decisión sea true o false, ya paso a la siguiente decisión.

#### **COBERTURA DE CONDICIÓN**

Una **condición** es una evaluación lógica que se encuentra dentro de una decisión.

El objetivo de la cobertura de condición es buscar la cantidad mínima de casos de prueba que me permiten valuar cada una de las condiciones tanto en su valor verdadero como en su valor falso independientemente de por donde salga la decisión.

Para el objetivo del Testing con método de caja blanca utilizando cobertura de condición, no le interesa hacer salvedades con respecto al cortocircuito de los conectores lógicos (si hay dos condiciones && y la primera ya da falsa, no se valúa la otra), sino que lo que busca hacer es evaluar cada condición en sus valores true y false minimizando la cantidad de casos de prueba.

#### **COBERTURA DE DECISIÓN/ CONDICIÓN**

La única variante con las anteriores es que lo que busca esta cobertura es no solamente valuar las decisiones en su valor verdadero y en su valor falso, sino también valuar todas las condiciones en su valor verdadero y en su valor falso.

#### **COBERTURA MÚLTIPLE**

La cobertura múltiple busca valuar el combinatorio de todas las condiciones en todos sus valores de verdad posible; lo que hacemos es plantear el combinatorio de los valores de verdad para toda la combinación de condiciones que tenemos disponibles.

## ELEGIR UN MÉTODO

Cada uno tiene fortalezas y debilidades particulares: un método puede ser bueno para algunas cosas, y no para otras cosas; **el mejor método es no usar un único método, usar variedad de técnicas ayudará a un Testing efectivo.**

## CICLO DE PRUEBA O CICLO DE TEST

Un ciclo de pruebas abarca la ejecución de la totalidad de los casos de prueba establecidos aplicados a una **misma versión del sistema a probar**; ejecuto todos los casos de prueba, veo cuales fallaron, corrojo los defectos, y hago otro ciclo, lo que implica correr todos los casos de prueba en la **nueva versión** que ya tiene los defectos corregidos, esto en loop hasta que **cumplamos con el criterio de aceptación que hayamos definido para el Testing**.

**Con el cliente debemos definir en qué momento vamos a dejar de probar; es imposible probar hasta que no haya defectos.**

el ciclo de prueba empieza con la corrida del primer cp y termina cuando se ejecutaron todos los cps previstos o cuando existe un error invalidante que impide que el ciclo de prueba continue. El ciclo de prueba no lo interrumpis a menos que tengas un defecto invalidante o bloqueante.

Si estas ejecutando el cp y encontrais defectos, ese es el propósito del ciclo de prueba, anotas el defecto, lo documentas y al final del ciclo de pruebas presentas el informe de defectos. Y cuando el soft viene con una nueva versión después que se supone que corrigieron todas las cosas, inicia un nuevo ciclo de prueba. El primer ciclo suele llamarse ciclo 0. Y a partir de ahí depende como este el producto de soft, veras cuantos ciclos de prueba vas a tener. Ideal seria tener dos ciclos de prueba, el 0 y el 1. Y en el 1 tener la salud del producto como para ponerlo en producción. La cantidad depende de muchos factores, no se puede prever. No asumir que vas a tener solo ciclo 0, es muy difícil. Al final de la corrida reportas todos los defectos que encontraste con la documentación mas clara posible de manera que ese error se pueda reproducir, para que la gente de desarrollo lo pueda localizar para reproducirlo. Por eso es importante tener cps diseñados y ejecutar pruebas sistemáticas basadas en un proceso, metodológicas, porque sino encontrais defectos y no podés reproducir lo que hiciste para que el error aparezca, perdes tiempo y dinero.

## REGRESIÓN

Al concluir un ciclo de pruebas, y reemplazarse la versión del sistema sometido al mismo, debe realizarse una verificación total de la nueva versión, a fin de **prevenir la introducción de nuevos defectos al intentar solucionar los detectados**.

La teoría nos dice que cuando solucionamos un defecto normalmente se introducen 2/3 defectos más. Es importante volver a verificar la nueva versión en su totalidad para asegurarnos de no haber introducido nuevos defectos. No siempre debemos hacer regresión, eso lo vamos a ir viendo a medida que testeamos.

Esta asociado al concepto de ciclos de prueba. Es decir que todos los ciclos de prueba se van a ejecutar como si fueran un ciclo 0. Cuando ejecutamos el ciclo 0, entendiendo que es el primer ciclo que se ejecuta, se ejecuta el ciclo y pruebo todos los cps. A partir de ahí el ciclo 1, si tomo el reporte de defectos que entregue a desarrollo como base para ejecutar las pruebas y solo pruebo las cosas que me dieron error, estoy haciendo una prueba sin regresión. Si tomo todos los cps y pruebo todo

como si fuera la primera vez que estoy probando, esa es una prueba con regresión. La de regresión es mas costosa, pero el precio que se paga sin regresión es errores ocultos, porque no sabemos si los programadores introdujeron errores nuevos al corregir lo que les reportamos, eso es lo que puede pasar. Por eso es mejor la regresión y por supuesto conforme nosotros vamos acostumbrándonos incorporando prácticas de automatización de testing, es mucho mas viable hacer pruebas de regresión si están automatizadas.

## PROCESO DE TESTING EN AMBIENTES TRADICIONALES

En el contexto de un **proceso definido** de testing, es el siguiente:



**PLANIFICACION:** Se trata de la parte más estratégica del testing, se genera un Plan de Prueba (hablando de un proyecto tradicional).

**IDENTIFICACION Y ESPECIFICACION:** Se definen y escriben los Casos de Prueba

Eso es diseño de las pruebas. Estas dos juntas serían lo que podríamos decir que es el diseño de los cps. Las pruebas se planifican y después se diseñan. Se diseñan los casos de prueba, prepara los datos para dejar todo listo para poder ejecutar

**EJECUCION:** Se lleva a cabo la ejecución de las pruebas (de manera manual o automatizada, o una combinación de ambas)

Ejecutar es tomar esta versión del producto que me dan y correr esos casos de prueba, y obtener como resultado el reporte de defectos

**ANALISIS DE FALLAS:** Se analiza si la solución/resultados de la ejecución de los casos de prueba fueron o no los esperados. Si obtuvimos los resultados esperados, ocurre el **FIN DE LAS PRUEBAS**, si no cumplió con lo esperado voy a hacer un análisis de fallas para corregirlo.

Este ciclo entre ejecución y análisis de fallas se ejecuta tantas veces como sea necesario para llegar al criterio de aceptación que va a determinar luego el **FIN DE LAS PRUEBAS**.

Si miramos el proceso contra los entregables, los entregables son el plan de prueba, los casos de prueba (conformado con los datos de prueba) y el informe de defectos. En algunos casos suele haber un informe final como de cierre del testing.

## VERIFICACIÓN vs VALIDACIÓN

Surgen dos conceptos importantes:

### VERIFICACIÓN

La verificación apunta a ver si estamos construyendo el sistema **correctamente**. Libre de defectos, apuntando a la perfección de su funcionalidad.

### VALIDACIÓN

La validación apunta a ver si estamos construyendo el sistema **correcto**, es decir, el sistema que cumple efectivamente con los requerimientos del cliente.

## EL TESTING EN EL CICLO DE VIDA DEL SOFTWARE

Es importante lograr involucrar las actividades del testing lo más temprano posible en el ciclo de vida del software.

Si yo tengo definidos en principio los requisitos, ya puedo empezar a definir los casos de prueba. Cuanto antes empecemos a realizar testing, podremos dar visibilidad más temprana al equipo sobre cómo se va a probar el producto y vamos a lograr disminuir costos de correcciones de defectos, ya que la idea sería encontrar **errores**, no **defectos**.

Con respecto al ciclo de vida, se prueba de forma inversa a la que se desarrolla. Se desarrolla de lo general a lo particular, el nivel de granularidad de los reqs es grueso y después vamos bajando mas detalle hasta el código que es lo mas detallado. Se prueba al revés, porque uno empieza probando a nivel de prueba de unidad que prueba cada componente por separado, después de integración, de sistema. Es una relación inversa.

Uno puede adelantar actividades del proceso de prueba mientras el código se esta construyendo. Yo estoy en condiciones de hacer planes de prueba cuando estoy planificando el proyecto y cuando tengo mas o menos los reqs puedo planificar. Cuando tengo los reqs puedo diseñar los casos de prueba, no hay nada que lo impida, y empezar a juntar los datos y a prepararlos. Cuando llega el código a testing el 50% del trabajo de testing ya lo puedo haber adelantado. Mientras se esta desarrollando la versión de producto puedo tener un plan de prueba, diseñar cps, preparar los datos y dejar todo el ambiente de prueba listo, para cuando viene la etapa de ejecución ya tengo todo listo. En términos de esfuerzo es 50 50, el 50% se puede adelantar sin ningún problema y es esperable que así sea.

## ¿CUÁNTO TESTING ES SUFICIENTE?

Es IMPOSIBLE probar todas las opciones posibles, el testing exhaustivo no es viable.

Decidir cuánto testing es suficiente depende de una evaluación de nivel de riesgo y de costos asociados al proyecto, depende del contexto del software y de su funcionalidad.

Se define un criterio de aceptación, con colaboración del cliente, y en base a este criterio se determina cuánto testing se va a realizar. Puede ser un punto específico, un costo, un tiempo, etc.

## PRINCIPIOS DEL TESTING

- ✚ El testing no es una etapa que comienza al terminar de codificar. Lo ideal es que sea temprano en el ciclo de vida del proyecto.
- ✚ No es probar que el software funciona, sino encontrar defectos/errores.
- ✚ El tester no es el enemigo del programador, si bien tienen objetivos contrapuestos, el tester lo que hace es visualizar lo que está sucediendo, muestra la presencia de defectos, no rompe el software.
- ✚ El testing ayuda a que el software sea confiable, no asegura la calidad.
- ✚ El testing exhaustivo es imposible.
- ✚ **PARADOJA DEL PESTICIDA:** Implica que cuando ejecutamos muchas veces los mismos casos de prueba, puede suceder que los casos de prueba se ejecuten sin fallas, pero que los verdaderos defectos no se verifiquen y queden ocultos.
- ✚ Es dependiente del contexto.
- ✚ Es una falacia decir que cuando se termina el testing hay **ausencia** de errores.
- ✚ Un programador debería evitar probar su propio código a excepción de las pruebas unitarias (único caso de excepción)
- ✚ Una unidad de programación no debería probar sus propios desarrollos
- ✚ No solo voy a buscar examinar lo que el software debería hacer sino también buscar examinar lo que pasa si el software no hace lo que debería hacer. (Ver qué pasa si hago un mal uso del software)
- ✚ No planificar el esfuerzo del testing sobre la suposición de que no se van a encontrar defectos, siempre ir con predisposición a encontrarlos.

## TIPOS DE PRUEBAS

### SMOKE TEST

Es la primera corrida de los test de sistema en la cual se verifica que las funcionalidades básicas de una aplicación o sistema funcionen correctamente antes de realizar pruebas más exhaustivas.

### SANITY TEST

Se realiza después de una ronda de pruebas más exhaustivas para verificar si se han corregido los errores o problemas críticos identificados previamente. El objetivo principal del sanity test es asegurarse de que las correcciones realizadas no hayan introducido nuevos problemas y que las funcionalidades clave del software sigan funcionando correctamente después de las correcciones.

### TESTING FUNCIONAL (¿Qué hace el sistema?)

Las pruebas se basan en funciones y características (descriptas en los documentos de requerimientos y entendidas por los testers) del software y su interoperabilidad con sistemas específicos. Si los requerimientos están bien planteados el caso de prueba deriva directamente de ellos

- Basado en requerimientos
- Basado en los procesos de negocio

## TESTING NO FUNCIONAL (¿Cómo lo hace el sistema?)

Las pruebas se basan en aspectos no relacionados directamente con las funciones específica de un software. Se centran en características como el rendimiento, la seguridad, la usabilidad y la escalabilidad del sistema. Acá se ve reflejada directamente la necesidad de que los ambientes de prueba sean lo más parecidos posible al entorno de producción ;.

- Performance Testing
- Pruebas de Mantenimiento
- Pruebas de Carga
- Pruebas de Fiabilidad
- Pruebas de Stress
- Pruebas de Portabilidad
- Pruebas de Usabilidad

## TDD (Test Driven Development)

Es una práctica de desarrollo de software que se enfoca en escribir pruebas antes de desarrollar código. Se centra en el desarrollo funcional.

Desarrollo guiado por pruebas de software o Test-Driven Development es una técnica avanzada que involucra dos prácticas, **Test First Development** (antes de escribir una línea de código, pienso en cómo voy a probar el correcto funcionamiento de ese código) y **Refactoring** (muchas veces voy escribiendo el código y lo desecho/modifico para que cumpla con las pruebas, pero sin modificar su funcionalidad, eso es refactoring).

Se trata de escribir las pruebas que voy a realizar primero y luego realizar el código necesario para hacer que esas pruebas pasen.

RED : test fails  
GREEN: test passes  
BLUE: refactor

Primero elijo el requerimiento que voy a probar, escribo la prueba, escribo la implementación de tal manera que logremos que el test pase. Lo importante es que el teste no falle (Green) una vez que logre que pase la prueba, hago refactorización, implemento el código de la mejor manera posible.

## TESTING EN AMBIENTES AGILES

El testing en ambientes agiles genera la tratativa del **Manifiesto de Testing Ágil**. Está planteado parecido a lo que es el Manifiesto Ágil.



El primer aspecto que aparece en el manifiesto habla de que NO se hace testing al final, sino que se hace mientras se va construyendo el software. Es responsabilidad de todo el equipo, no necesariamente hay un tester con un rol determinado, sino que cualquiera puede tomar ese rol.

El manifiesto plantea además todo lo que se tiene en cuenta en el testing en ambientes agiles. Prevenir bugs sobre encontrar bugs, es decir, prevenir los errores, no encontrar directamente defectos.

Darle lógica a lo que estamos testeando, no querer solo tildar una casilla y verificar una funcionalidad. El tester debe querer encontrar errores/defectos del sistema para construir un mejor sistema, no para romperlo.

En testing en procesos agiles hay ciertos principios que son la base para poder incorporar las prácticas de ágil.

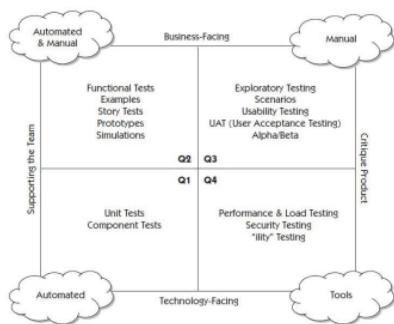
Los **PRINCIPIOS DE TESTING AGILE** son cuestiones más conceptuales.

1. **EL TESTING SE MUEVE HACIA ADELANTE EN EL PROYECTO:** No se realiza al final del proyecto, sino que se va haciendo de manera constante al finalizar cada iteración.
2. **TESTING NO ES UNA FASE:** Se hace mientras se desarrolla el código
3. **TODOS HACEN TESTING:** No hay un rol específico de tester, cualquiera dentro del proyecto puede testear.
4. **REDUCIR LA LATENCIA DEL FEEDBACK**
5. **LAS PRUEBAS REPRESENTAN EXPECTATIVAS**
6. **MANTENER EL CODIGO LIMPIO, CORREGIR LOS DEFECTOS RAPIDO:** Cuando hablamos del manifiesto hablamos de prevenir errores antes de solucionarlos
7. **REDUCIR LA SOBRECARGA DE DOCUMENTACION EN LAS PRUEBAS**
8. **LAS PRUEBAS SON PARTE DEL “DONE”:** En el criterio de DONE se incluye la realización de pruebas.
9. **DE PROBAR AL FINAL A “CONDUCIDO POR PRUEBAS”**

Luego tenemos las **PRÁCTICAS DE TESTING AGILE** que son cuestiones más concretas sobre cómo implementar testing en ambientes agiles.

- **TESTING UNITARIO O DE INTEGRACIÓN AUTOMATIZADO:** Todo lo que yo ejecuto de manera mecánica se automatiza. Gano tiempo, ahorro recursos y los utilizo en otro contexto.
- **M TESTING REGRESIVO A NIVEL DE SISTEMA AUTOMATIZADAS:** Las pruebas de regresión también podrían automatizarse.
- **TESTING EXPLORATORIO:** El testing exploratorio no se puede automatizar
- **TDD (Test Driven Development)**
- **ATDD (Desarrollo conducido por Pruebas de Aceptación):** Esta relacionado con TDD, pero tiene que ver con el desarrollo conducido por las pruebas de aceptación.
- **CONTROL DE VERSIÓN DE LAS PRUEBAS CON EL CÓDIGO:** Junto con el código tengo asociados los **Unit Tests** y los **System Tests** que tengo que correr y de esa manera es fácil plantear la automatización del testing

### CUADRANTES DEL TESTING ÁGIL



Nos hablan de cuales son todos los tipos de testing disponibles, cual es el foco de cada uno, hacia donde apuntan y cuales son aquellos que pueden automatizarse y los que deben realizarse de manera manual.

Los cuadrantes 1 y 4 tienen que ver con aspectos tecnológicos del testing, y los 2 y 3 tiene más que ver con la parte del negocio del testing.

Especificamente el cuadrante 4 tiene que ver con aspectos no funcionales como tests de performance, seguridad, etc.

Los cuadrantes 1 y 2 tienen más relación con el soporte al equipo y los 3 y 4 están más orientados a lo que hacemos sobre el producto.

## FILOSOFÍA LEAN

Es una filosofía de gestión que surge en la manufactura, en la línea de producción, no en el mundo del software. En particular surge en Toyota, donde se empieza a trabajar y a relacionarse con conceptos de Optimidad. Lean no solo se basa en optimizar la línea de producción, sino que busca satisfacer necesidades y expectativas del cliente, optimizar el consumo de recursos, minimizar/eliminar desperdicios y hacer foco en el lugar donde se crea valor. Busca a su vez, relacionado con lo mencionado anteriormente, desarrollar las capacidades relacionadas con la resolución de problemas.

Esta filosofía cuenta con 7 principios, a través de los cuales se trata de optimizar lo que tiene que ver con la gestión del cambio. Con Scrum gestionábamos proyectos, con Kanban realizamos prácticas más específicas que no tienen tanto que ver con la gestión de proyectos, sino con la gestión de un ticket/tema/requerimiento de manera individual.

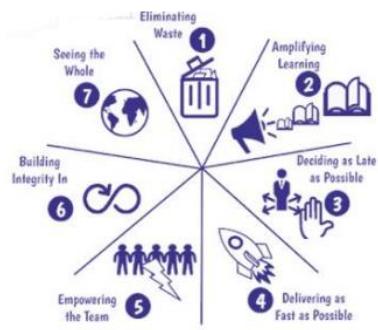
**Comentado [5]:** creo que mas bien Kanban gestiona proceso

Todo lo que tiene que ver con Lean es menos prescriptivo que en Agile, si en Agile tenemos pocas reglas, en Lean hay muchas menos.

Si bien nos da otra firma de trabajar relacionada con la **gestión del cambio**, muchos de los principios agiles tienen que ver con los principios de lean.

Una de las características que tiene que ver con el éxito en Lean, es su sentencia: "**toma lo que tengas, toma el proceso que tengas y a partir de ahí trata de aplicar estos principios y de mejorar lo que hoy ya tenes**", a diferencia de Agile, que no parte de algo que ya tenemos para ir mejorándolo, sino que es bastante prescriptivo en lo que propone.

## PRINCIPIOS LEAN



### 1. ELIMINAR EL DESPERDICIO

Si pensamos en términos de software, hablamos de **desperdicio** como aquello que no agrega valor al negocio, que implique trabajo innecesario o re trabajo.

Lean apunta a minimizar el re trabajo y no hacer actividades que no agreguen valor al negocio. Llevado al software todo lo que tiene que ver con re trabajo o cuyo desarrollo quedó en la mitad, es desperdicio. Esto surge a partir del concepto de Just in Time. Tiene que ver con el principio ágil de **Software Funcionando** y el de **Simplicidad**.

Esto implica que el tiempo que vamos a tomarnos en hacer algo que no agrega valor al sistema, debe tender a 0, tiene que ser mínimo.

Es de los principios que mas fuerza tiene en esta corriente y apunta a mejorar la productividad y consiguiente la calidad también. En el software dejar que invirtamos una excesiva cantidad de tiempo en hacer especificaciones de lo que va a ser al producto al principio y después eso cambie es una de las mayores fuentes de desperdicio que tenemos. Esta situación que dice ágil de maximizar la simplicidad y el trabajo no hecho, esta alineado con eliminar desperdicios. En nuestra industria que es intangible, hacer características en un producto de software que nadie usa es un desperdicio. La inmensa mayoría del soft que ponemos en disposición no se usa, entonces ahí se cumple lo que plantea que el 80% del valor del producto esta focalizado en el 20% de las características que nosotros entregamos, entonces si somos capaces de hacer foco en ese 20% trabajaremos mejor en muchos sentidos

### DESPERDICIOS EN TÉRMINOS DE PRODUCCIÓN



El hacer cosas de mas, producción en exceso genera desperdicio porque genera stock, almacenamiento, problemas de logística, y en el medio, dependiendo de lo que sea que se esta fabricando, se puede poner viejo, se puede echar a perder, se puede romper.

El stock es otro de los gastos que queremos eliminar. Los pasos extra, significa hacer cosas que no generan valor y por consecuencia son desperdicio. El tiempo que se pierde buscando info que hace falta para poder cumplimentar la tarea. Los defectos. Esperas, la línea esta parada porque la materia prima no llego. La logística, incide en el precio de los productos el transporte.

### DESPERDICIOS EN TÉRMINOS DE SOFTWARE

- **Características extra:** son aquellos elementos que están fuera del 20% del software que agrega valor.
- Trabajo a medias: es como trabajo no realizado
- Procesos extra
- Movimiento
- Defectos
- Esperas
- **Cambio de tareas:** con el cambio constante, se pierde el foco en la tarea. En lean se busca terminar la tarea que iniciamos antes de pasar a la que sigue.

*"Deja de empezar y comenzá a terminar"*

En el soft hay que adaptar un poco los gastos estos a la realidad del soft. Hacer reqs que nadie te pidió. Las cosas parcialmente terminada, por eso los procesos empíricos apuntan a gestiones binarias, piezas de trabajo mas chiquitas que están terminadas o no. El hecho de calcular el porcentaje de avance de una determinada tarea, ya es desperdicio, aparte es muy difícil de medir el

porcentaje de avance en un trabajo intelectual e intangible. Los pasos extra en un proceso cuando uno copia un proceso de desarrollo de otro espacio y trata de inculcarlo en la org, suele generar actividades que la gente las cuestiona porque siente que no generan valor. El tema del movimiento. Los defectos son algo endémico y costosísimo, el 50% del costo de un producto de soft es retrabajo, consecuencia de que se hizo algo mal. Las esperas en nuestro caso se dan si estoy trabajando en un equipo y necesito info que me tiene que dar otra persona, o necesito algo que esta trabajando otro equipo, tercero o un proveedor, eso demora el logro de los objetivos. En el caso del cambio de tareas también, de hacer multitasking, tener muchas cosas al mismo tiempo haciendo, eso es muy improductivo, mucho mas en el trabajo intelectual porque uno tiene un tiempo de seteo de preparar la cabeza para hacer un trabajo de un determinado tipo. Hay una definición muy clara de los enfoques de pensamiento lean de no hacer multitasking, una tarea asignada por vez, la empieza, la termina y después busca otra.

## 2. AMPLIFICAR EL APRENDIZAJE

Cultura de mejora continua, se basa en poder aprender a partir del trabajo que estamos haciendo y que ese conocimiento sea accesible a toda la organización. Esto tiene relación con el empirismo. Tienen que ver con el valor que le damos a la figura del equipo y a la autogestión del mismo, esto es gracias a la amplificación del aprendizaje y a la mejora continua a través del mismo.

Está relacionado con el principio ágil de **Autogestión** o **Auto organización** de los equipos.

La idea es que el equipo auto organizado logre un conocimiento en forma de espiral que le permita aprender en términos del equipo y compartir con el equipo para que ese proceso de aprendizaje (cómo hacer las cosas en términos de proceso, pero también incluye cuestiones técnicas) vaya amplificándose

Un proceso focalizado en crear conocimiento esperará que el diseño evoluciones durante la codificación y no perderá tiempo definiéndolo en forma completa, prematuramente.

Se debe generar un nuevo conocimiento y codificarlo de manera tal que sea accesible a toda la organización. Muchas veces los procesos "estándares" hacen difícil introducir en ellos mejoras.

Amplificar el aprendizaje. La idea es que el conocimiento que se va generando en el equipo mientras nosotros vamos trabajando, es conocimiento que tenemos que compartirlo con el resto de las personas del equipo. **Hacer explícito un conocimiento implícito.** Entonces básicamente la idea que hay atrás de lean es la misma que hay atrás de cualquier proceso empírico que es mejora continua permanente de todo lo que se identifique que se puede mejorar. Entender que el conocimiento sirve si yo lo comarto y tengo que estar predisposto a que las cosas mejoren y estar abierto a que las cosas pueden mejorar.

## 3. EMBEBER LA INTEGRIDAD CONCEPTUAL

Habla sobre contemplar todas las partes del producto/servicio de manera consistente haciendo foco en lo que tiene que ver con los requerimientos funcionales. Incluir todas las partes desde el principio, no focalizarse en una y olvidarse de las otras. Este principio incluye en términos de prácticas, a aquellas como las revisiones entre pares.

Tiene que ver específicamente con el concepto de arquitectura de software, pensando en una estructura de producto que sea escalable, que se robusto, que tenga coherencia y consistencia y que sea mantenible a medida que pase el tiempo.

En Lean, se busca pensar la arquitectura desde el principio de la construcción del producto.

*"Encastar todas las partes del producto o servicio, que tenga coherencia y consistencia (tiene que ver con los requerimientos no funcionales). La integración entre las personas hace el producto más integral"*

Este principio también apunta al concepto de **calidad**: no hacer re trabajo sino construir con calidad desde el principio. Y a su vez, busca introducir la técnica de inspecciones: sirve en gran parte para prevenir el defecto, pero también se puede utilizar luego de que los mismos ocurrán.

**INTEGRIDAD PERCIBIDA:** el producto total tiene un balance entre función, uso, confiabilidad y economía que le gusta a la gente.

**INTEGRIDAD CONCEPTUAL:** todos los componentes del sistema trabajan en forma coherente en conjunto.

**Si se quiere calidad, no inspeccione después de los hechos, y si no es posible, inspeccione luego de pasos pequeños**

Embeber la integridad conceptual, tiene que ver con la calidad, nosotros trabajamos un grupo de personas para entregar jun producto, pero la calidad tiene que estar presente, no se negocia. Básicamente tenemos que trabajar en equipo entiendo que el todo es mas que la suma de las partes, cada uno tiene que contribuir a entregar el producto que hay que entregar, pero estar atentos permanentemente a la calidad del producto. Esta implícito en el trabajo que nosotros tenemos que hacer. La calidad es algo que se va construyendo junto al producto, no es algo que lo podes dejar al final, por ejemplo haciendo mas testing. La calidad primero es una decisión de todos los que estamos involucrados, de como nosotros queremos trabajar y de si queremos hacer las cosas de la mejor manera que podamos o no.

#### 4. DIFERIR COMPROMISOS

Es una de las cosas claves de Lean.

Se refiere a tomar decisiones al último momento en el cual es necesario hacerlo. No antes, debido a la probabilidad de incertidumbre. Tiene relación directa con las US en el Product Backlog.

La idea es poder avanzar sin necesidad de tener todo definido. Lo ideal es diferir la decisión que debemos tomar hacia el final.

Si bien es difícil de lograr, muchas veces nos da ventaja competitiva ya que tiene que ver con demorar el momento de la toma de decisiones lo más que se pueda porque cuanto antes la tomemos nosotros, más incertidumbre vamos a tener y hay información que nos está faltando.

Tenemos que tomar la decisión en el último momento en donde nosotros nos aseguremos que no perdemos nada por no tomar la decisión: último momento responsable.

Ejemplo: en Agile, lo hacemos con los requerimientos, tenemos la pila en el product backlog y decidimos no escribir de manera detallada la feature que están debajo de la pila ya que probablemente al estar abajo no está dentro de las prioridades y si capaz se definen en forma detallada, en un futuro sufren cambios.

Se relaciona con el principio ágil: **decidir lo más tarde posible pero responsablemente**. No hacer trabajo que no va a ser utilizado; también se enlaza con el principio anterior de aprendizaje continuo, mientras más tarde decidimos, más conocimiento tenemos.

Diferir compromisos. Habla de tomar decisiones informadas. Sustenta la idea de los reqs agiles, de no tener un producto 100% definido sino empezar con algo y después cuando se tiene mas info ver de cambiar o completar lo que falta. Diferir hasta el ultimo momento responsable, lo tengo que diferir pero en algún punto optimo tengo que tomar la decisión, porque si no la tomo yo la toma

alguien por mi. No todas las decisiones se van a diferir porque si diferis todo nunca arrancarías a hacer nada. Ayuda con la parálisis por análisis, esta cosa de buscar lo perfecto, y no es bueno seguir pensando y que pasa si, y te quedas en esa etapa y no avanzas. Hay que tratar de contrarrestar esa situación y este principio ayuda a eso.

## 5. DAR PODER AL EQUIPO

(A.K.A equipos auto gestionados, sin roles específicos sino con asignaciones internas y estimación de roles)  
Dar poder a la gente, delegación de decisiones y posibilidades, la persona que lo va a hacer es la que va a tomar las decisiones.

En Lean, se busca que las personas que son las responsables de hacer las cosas tengan la suficiente delegación de decisiones y de responsabilidades para que ellos se puedan hacer cargo.

Se busca **respetar a la gente**: entrenar líderes, fomentar buena ética laboral y delegar decisiones y responsabilidades del producto en desarrollo al nivel más bajo posible.

Dar poder al equipo tiene que ver con que buscamos equipos que puedan tomar decisiones, equipos formados, capacitados, motivados, y que se autoorganicen y autogestionen. Eso permite cosas como que se estima en el equipo o estima el que va a hacer el trabajo. Le permitimos al equipo que tome decisiones y que avance sin tener un jefe que este todo el tiempo diciéndole que tiene que hacer, como y cuando.

## 6. VER EL TODO

Tiene que ver con tener una visión del conjunto, tener una visión holística del producto. Debe incluir el concepto de diseño arquitectónico que va más allá de funcionalidades específicas entregadas de forma independiente.

*“Tener una visión holística, de conjunto (el producto, el valor agregado que hay detrás, el servicio que tiene todos los productos como complemento)”*

Hay como un “hueco” de Scrum que tiene que ver con la definición de la arquitectura del producto, y buscamos que el producto como un todo incluya el concepto de diseño arquitectónico.

Ver el todo, esta muy relacionado con la entrega de valor. No solo ver la línea de código, la pantalla de la compu, sino entender que nosotros estamos haciendo esto para algo que es mas grande, y esto en definitiva es una herramienta. Si puedo ver esa situación voy a poder entregar un producto que tenga valor realmente para la persona que me lo pidió.

## 7. ENTREGA RÁPIDA

Agile plantea lo mismo en entregas frecuentes de software funcionando.

Plantea acortar el ciclo de desarrollo y los tiempos y entregar cuanto antes al cliente. Esto tiene que ver con el desperdicio, muchas veces se demora la entrega debido a esto.

**Entregar rápido** implica estabilizar ambientes de trabajo a su capacidad más eficiente y acotar los ciclos de desarrollo.

**Entregar rápidamente** hace que se vayan transformando “n” veces en cada iteración. Incrementos pequeños de valor; llegar al producto mínimo que sea valioso; salir pronto al mercado.

Tenes que dar retroalimentación, y esa se le da entregando rápido, preguntándole que opina, obteniendo feedback y mejorando lo que haya que mejorar. Entrega frecuente de software funcionando

## KANBAN

Una manera de aplicar Lean, es utilizar Kanban. Es el referente principal de Lean en software.

Sirve para gestionar cosas para las cuales Scrum no es muy útil.

Es un enfoque que se usa para la gestión del cambio, no es un proceso de desarrollo ni una metodología de administración de proyecto, es un proceso para gestionar el cambio en procesos de desarrollo de software, en proyectos, pero es un ENFOQUE para la gestión de cambios.



Kanban sirve para mejora continua de procesos, basada en el pensamiento lean. Fue desarrollado por David Anderson. Se usa en el contexto de procesos de desarrollo de software pero NO es una metodología, ni un proceso de desarrollo. Es menos prescriptivo que scrum, menos cosas plantea. No te va a enseñar a hacer software, simplemente es un framework para implementar mejoras continuas en las organizaciones

Es una expresión japonesa que hace referencia a señalización, es una tarjeta de señal. Es el medio por lo cual uno quiere lograr. Necesitamos lograr la visualización del trabajo, del flujo, saber en cada momento del proceso en donde esta la pieza esta que tenemos que entregarle al cliente. Con minúscula es una señalización de algo, que nos sirve para ver en que estado esta el proceso de desarrollo de la pieza que tenemos que entregarle al cliente.

Uno de los primeros principios que plantea es "Empieza por lo que tienes ahora" es sumamente importante.

Una de las cosas importantes que te dice Kanban es empeza con lo que tenes, no hagas grandes cambios ni patees el tablero, cual es el proceso que estas usando ahora tráelo, lo vamos a analizar y ver si cada paso genera valor, y ese proceso lo vamos a mapear a un tablero, y vamos a configurarlo y establecer los límites de work in progress para cada etapa del proceso, y vamos a empezar a trabajar, vamos a definir los tipos de trabajo que van a resolverse en el tablero, y que el trabajo fluya. Y ahí vemos que cosas hay que ir mejorando, como eliminamos desperdicios y como vamos haciendo la mejora continua del progreso

Otro principio de cambio que plantea tiene que ver con el liderazgo en todos los niveles, yo elijo lo que tengo que hacer y no que alguien me exija lo que tengo que hacer. Se plantea el concepto de mejora continua.

### LLEVANDOLO AL PROCESO DE DESARROLLO DE SOFTWARE

Lo primero que tenemos que hacer es poder materializar en términos concretos, cuál es el proceso que me representa, o cuál es el proceso que hoy estoy ejecutando; poder visualizar el flujo, cuál es la idea para poder tener identificadas todas las tareas/ etapas de mi proceso.

Kanban no es para la gestión de proyectos, sino más bien es para una gestión de cambios. No es para modelar un proceso de desarrollo de software, sino que nos sirve para introducir cambios en un proceso de desarrollo.

Buscamos que el proceso fluya, que los items que están a la izquierda lleguen a la derecha y para ellos nos basamos en **mejorar colaborativamente y WIP o trabajo en progreso**: buscamos que todo trabajo que esté en progreso se limite para evitar el embottellamiento y que el trabajo fluya.

## PRINCIPIOS/PRÁCTICAS GENERALES KANBAN

### 1. VISUALIZAR EL TRABAJO

Hacer el trabajo visible constantemente.

Kanban se basa en el uso de tableros visuales para representar el flujo de trabajo. Estos tableros muestran las tareas pendientes, en progreso y completadas, brindando una visión clara y transparente de todo el proceso.

### 2. LIMITAR EL TRABAJO EN CURSO (WIP):

Kanban promueve la idea de limitar la cantidad de trabajo en curso en cualquier momento dado. Esto evita la sobrecarga del equipo y ayuda a mantener un flujo de trabajo equilibrado y constante.

### 3. GESTIONAR EL FLUJO

El enfoque principal de Kanban es lograr un flujo de trabajo continuo y eficiente. Se deben identificar y resolver los cuellos de botella y los problemas que afectan el flujo de trabajo, con el objetivo de optimizarlo y garantizar una entrega rápida y constante.

### 4. HACER EXPLÍCITAS LAS POLÍTICAS

Kanban requiere que las políticas de trabajo y los procedimientos estén explícitos y sean claros para todos los miembros del equipo. Esto incluye definir los criterios de finalización de una tarea, las prioridades y cualquier otra regla que afecte el proceso.

### 5. MEJORAR DE FORMA COLABORATIVA Y EVOLUTIVA

Kanban fomenta la mejora continua a través de la colaboración y el aprendizaje. Los equipos deben buscar constantemente formas de mejorar el proceso y adaptarlo a medida que surgen nuevas ideas y desafíos.

### 6. RESPETAR LOS ROLES, RESPONSABILIDADES Y HABILIDADES EXISTENTES

Kanban reconoce y valora las habilidades y el conocimiento existentes dentro del equipo. No impone roles predefinidos, sino que permite que los equipos se organicen según sus fortalezas individuales, siempre y cuando se respeten las responsabilidades y se logre un flujo de trabajo eficiente.

Estos seis principios de Kanban ayudan a los equipos a trabajar de manera más eficiente, mejorar la colaboración y la calidad, y a adaptarse rápidamente a los cambios.

Kanban aprovecha muchos de los conceptos probados de Lean.

En el desarrollo de software, Kanban plantea una evolución de lo que ya tengo. Fomenta la evolución gradual de los procesos existentes y no pido una revolución sino un cambio gradual.

## ¿CÓMO APLICAR KANBAN?

Primero, entendiendo lo que tenemos hoy, como trabaja el equipo y que hace, poder graficarlo en un tablero. Luego, acuerdo los límites del trabajo en progreso (WIP), en cada una de las etapas voy a definir la **cantidad máxima de tarjetas que puedo tener en progreso en cada una de las etapas**, que va a estar vinculada directamente con la cantidad de recursos que yo tenga trabajando en esa parte del proceso.

- 1- **DIVIDIR EL TRABAJO EN PIEZAS:** las US son buenas para esto, pueden ser requerimientos, incidentes, etc. Luego de dividir el trabajo, podemos tipificar las piezas (tarjetas) en función de lo que estemos modelando. Podemos asignarles colores a los distintos tipos de trabajo (ej. Requerimientos correctivos con un color, requerimientos con fecha límite con otro color, etc.). En cada una de las tarjetas se usa una nota diferente.

| Story | To Do | In Progress | To Verify | Done |
|-------|-------|-------------|-----------|------|
|       |       |             |           |      |
|       |       |             |           |      |
|       |       |             |           |      |
|       |       |             |           |      |

Tenemos que identificar que tipos de trabajo son los que vamos a hacer nosotros durante el trabajo. Pueden ser user stories, particionar el producto en users y que la user sea la pieza de trabajo que hay que desarrollar, pero no es la única posibilidad, hay problemas, errores, atención al cliente, uno puede necesitar hacer distintas cosas. La idea es identificar cuales son los tipos de trabajo que se van a realizar en el tablero, y después asociarle a eso la calidad de servicio que queremos ofrecer para cada uno de esos tipos de trabajo

- 2- VISUALIZAR EL FLUJO DE TRABAJO:** utilizar nombres en las columnas para ilustrar donde esta cada ítem en el flujo de trabajo, cada una de las piezas definidas en el punto anterior van a fluir de izquierda a derecha en las columnas.
- Uno de los conceptos clave de Kanban es que cuando tengamos estas piezas sean los recursos los que se autoasignan el trabajo y no que alguien asigne el trabajo a un recurso.

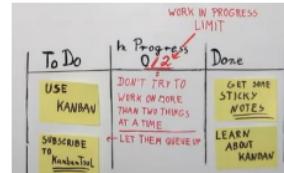


#### FUNCIONA COMO PULL, NO COMO PUSH

Hay dos tipos de columnas: de acumulación y de trabajo. Las de acumulación son todas las que dicen listo, igual que el terminado. Las que dicen en curso, haciendo, esas son las de trabajo.

Sistemas de asignación pull, no hay nadie que te empuje a vos en tu columna esto tenes que hacer, sino que vos vas y buscas el trabajo que vos consideras que vas a hacer cuando podes, que esta relacionado con el wip. En Kanban para que el trabajo fluya tiene que haber esta situación de trabajo pull, no push, que tambien esta en scrum, que esta dado por la autoorganización y autogestión de los equipos.

- 3- LIMITAR EL WORK IN PROGRESS (WIP):** Asignar límites explícitos de cuántos ítems puede haber en progreso en cada etapa del flujo de trabajo, para asegurar que el trabajo fluya. Esto ayuda a prevenir la sobrecarga y a mantener un flujo de trabajo equilibrado. Establece límites realistas basados en la capacidad del equipo y las necesidades del proyecto.



El límite de trabajo en progreso es de las cosas mas difíciles de definir y sostener. Muchas veces se definen pero no se cumplen. Significa que si yo determine en función de la capacidad de trabajo que tiene esa columna que no puede haber simultáneamente mas de dos piezas, eso es, y es lo que hay que respetar, entonces uno empieza, esto se estima, es una definición que se hace en función de la info que uno tiene, y después tenes que ajustarla, mejora continua, cada vez que se detecte que ese límite no esta funcionando bien. Nos ayuda a evitar atascamientos y cuellos de botella y a permitir que fluya el trabajo de la mejor manera posible. Si nosotros nos atoramos evitamos que el trabajo fluya. La asignación es autoasignación.

Para materializar Kanban e incluirlo en un proyecto, es importante seguir un proceso estructurado. Comienza por **comprender el contexto y los objetivos del proyecto**, identificando los problemas o desafíos específicos que se desean abordar mediante la implementación de Kanban.

Una vez que se tengan claros los objetivos, se **forma un equipo** con los miembros involucrados en el proyecto.

**Comentado [6]:** se me hace medio raro eso de seguir un proceso estructurado

A su vez, debemos **definir la unidad de trabajo** que vamos a utilizar, puede ser un requerimiento, una user story, una tarea o cualquier otra unidad de trabajo relevante para el proyecto. Se pueden utilizar colores o **clases de servicios** para categorizar y priorizar las unidades de trabajo. Estos colores o clases de servicios permiten establecer políticas de trabajo específicas según la categoría de cada unidad de trabajo, como prioridades, plazos de entrega, recursos asignados, etc.

El siguiente paso es **modelar el proceso** en el tablero Kanban, ya sea físico o digital, para **visualizar el flujo de trabajo**. Se divide el tablero en columnas que representen las etapas del proceso, desde las tareas por hacer hasta las tareas completadas. El mismo debe ser claro y comprensible para todos los miembros del equipo, y permitir tener una visualización clara del flujo de trabajo.

Una vez que el tablero esté diseñado y la unidad de trabajo está establecida, se deben **definir políticas de calidad** claras para cada columna. Se especifican los **criterios que deben cumplirse para mover una tarea de una columna a la siguiente** (evitando la acumulación de trabajo defectuoso en etapas posteriores y asegurando que se mantenga la calidad) y cuantas unidades de trabajo se está dispuesto a aceptar en cada etapa del proceso, estos **límites de trabajo en curso** (WIP) para cada columna ayudan a mantener un flujo de trabajo equilibrado, eficiente y evitar la sobrecarga del equipo.

Ahora, es el momento de implementar Kanban gradualmente. Se comienza con algunas columnas y tarjetas que reflejen el flujo de trabajo actual. A medida que el equipo se familiarice con el método Kanban, se pueden agregar más columnas y refinrar el proceso según sea necesario.

Durante todo el proceso, es importante capacitar al equipo sobre los principios y prácticas de Kanban. Fomentar la colaboración y la comunicación abierta, animando a los miembros del equipo a compartir ideas y sugerencias para **mejorar el flujo de trabajo y resolver problemas de manera conjunta**.

Es importante el monitoreo continuo del flujo de trabajo en el tablero Kanban para **identificar cuellos de botella o etapas donde se acumula demasiado trabajo**. En esos casos, se considera mover recursos de una etapa a otra para equilibrar el flujo.

Por último, se debe **establecer una cadencia para las actividades clave del proyecto**, como las releases, planificaciones y revisiones. Estas actividades deben programarse y realizarse regularmente según un calendario definido. Esto ayudara a mantener un ritmo constante y una mejor planificación y seguimiento del proyecto.

## 6. DEFINIMOS CLASES DE SERVICIO

Las clases de servicio sirven para definir cómo voy a tratar las piezas de trabajo. No es una cola FIFO. Básicamente ayudan a priorizar ciertas unidades de trabajo por sobre otras, dependiendo la prioridad de las mismas, si tenemos fechas específicas de entrega, etc.

Algunos ejemplos podrían ser:

Clases de servicio es que comportamiento van a tener las unidades de trabajo dentro del tablero. Se definen en esto de hacer expresas las políticas



Si viene algo de **alta prioridad/ expreso** debe resolverse lo antes posible. Defino para esta política que cuando llega algo expreso lo demás se pone en espera en la lista. Si llega esta clase, se puede exceder el límite del WIP. Se puede hacer una entrega especial para ponerla en producción.

pueden ser problemas que aparecen y que hay que resolver. Es una forma de plantear las urgencias. Solo puede haber una por vez.



Si tenemos una clase de servicio de **fecha fija**, no podemos modificar el WIP, pero si se permite dejar la unidad en cola hasta que sea conveniente que ingrese. Si se retrasa y la fecha de entrega está en riesgo puede promoverse a la clase de servicio **expreso**. Suelen entregarse en entregas programadas, cuidando la fecha de entrega.

Tiene un riesgo, de patearla por tener tiempo.



Una clase de servicio **estándar** usa la técnica FIFO, si no hay una clase de servicio Expreso o de Fecha Fija, el primero que entra al proceso es el primero que sale. Deben adherirse al WIP definido y son priorizados y puestos en cola con un mecanismo definido basado en el valor del negocio. Pueden analizarse también por tamaño en orden de magnitud. Son entregados en entregas programadas.



¿?

Algo así para manejar la deuda técnica. Cosas que no son fácilmente mostrables a los clientes porque les cuesta visualizarlas, pero en algún punto hay que hacerlas. Si se difieren demasiado se pueden convertir en expreso

## MÉTRICAS CLAVE

Así como en Agile y en Scrum tenemos ciertas métricas clave, en Kanban también. Son 3 métricas.

Como buscamos poner el foco en el cliente, en el valor agregado y en la entrega rápida, tenemos la métrica de **Lead Time (Vista del Cliente)** que nos indica el tiempo que demora una pieza de trabajo desde que ingresa a la cola de producto hasta que está terminada. Se suele medir en días de trabajo o esfuerzo. Es la medición más mecánica de la capacidad del proceso. La llamamos el **Ritmo de Terminación**

El **Cycle Time (Vista Interna)** es el tiempo desde que mi equipo empezó a trabajar con esa pieza del producto hasta que sale del ciclo. La diferencia es que esta métrica contiene el tiempo que la pieza se mantiene en cola. Se suele medir en días de trabajo. La llamamos **Ritmo de entrega**.

La última métrica es el **Touch Time (Tiempo de Tocado)** es el tiempo en el cual un ítem de trabajo fue realmente trabajado (o tocado) por el equipo. Se refiere a los días hábiles que pasó ese ítem en columnas de "Trabajo en Curso", en oposición con columnas de cola/buffer y estado bloqueado o sin trabajo del equipo sobre el mismo.

Se busca (obviamente) que el Touch Time sea lo más parecido posible al Cycle Time.

**Comentado [7]:** no se si no esta al revés lo de ritmo de terminación y de entrega

Scrum mide producto, Kanban mide proceso

La Eficiencia del Ciclo de Proceso se mide:

$$\% \text{ Eficiencia ciclo proceso} = \frac{\text{Touch Time}}{\text{Elapsed Time}}$$

### KANBAN CONDENSADO/COMPACTADO

Podríamos decir que las **prácticas concretas** son lo más fácil, es el trabajo concreto que voy a **hacer para poner en práctica los principios** que están planteados en el framework, la **práctica es lo seguro, no me asegura que los principios se respeten**, pero me ayuda a implementarlos.

**Los valores** por otro lado, tienen que ver con **aspectos organizaciones y de cultura de la organización**, **sin la base de los valores es muy difícil que las prácticas nos permitan implementar los principios** (Tiene que ver relacionando, con esto de “hacer ágil” y “ser ágil”).



Comentado [8]: elapsed time es el cycle

## MÉTRICAS DE SOFTWARE EN LOS DIFERENCIAS ENFOQUES DE GESTIÓN

Cuando hablamos de los componentes de un proyecto de software, uno de los aspectos mencionados era el de **monitoreo y control**, es importante poder hacerlo para detectar los desvíos y así corregirlos ya que sabemos que **los proyectos se atrasaban un día a la vez**. Más allá de las acciones puntuales del monitoreo y control, esto se materializa a través de las métricas.

*Las métricas nunca tienen que servir para medir a las personas, esto solo nos trae problemas.*

Las métricas simplifican el trabajo y pueden pasar por alto aspectos importantes, generando competencia poco saludable y desviando el enfoque del trabajo colaborativo y la calidad del producto. Además, las métricas a menudo carecen de contexto y no reflejan adecuadamente el valor que una persona aporta al equipo. También pueden generar sesgos, efectos negativos y estrés en las personas. En cambio, es preferible enfocarse en la colaboración, la comunicación abierta y la confianza en el equipo, evaluando el rendimiento y mejorando continuamente sin enfocarse exclusivamente en métricas individuales, sino en el éxito colectivo y en la mejora del proceso en general.

Una métrica tiene que tener un conjunto de características para que realmente justifique que vale la pena invertir, porque tomar métricas tiene un costo, entonces el beneficio tiene que ser mayor que el costo de obtenerlas. Todo eso es parte de la definición y del análisis.

Tener presente que una métrica es un número, un valor cuantitativo que determina la presencia de algo que yo quiero medir en el contexto en el que lo voy a medir. Un ejemplo tiene que plantear una métrica que sea medible y que se represente numéricamente para que sea objetiva.

## MÉTRICAS DE SOFTWARE EN EL ENFOQUE TRADICIONAL

**Basadas en Procesos Definidos**

En el enfoque tradicional (a diferencia de los enfoques Agile o Lean que suelen ser enfoques más simplificados que tiene que ver con una concepción más minimalista de las métricas) está basado en un proceso definido y en un conjunto de métricas a definir más amplio.

Hay más disponibilidad de métricas a definir y su definición queda a criterio del líder del proyecto, decidir qué métricas se van a tomar y elegir como va a trabajar con esas métricas.

Independientemente que este espectro de métricas en el enfoque tradicional sea bastante amplio, podemos subdividirlos en 3 conjuntos que se denominan **dominios de métricas**. Por dominio de métricas nos referimos al ámbito al que las métricas hacen referencia, al ámbito en el que las métricas sirven o son útiles. Esto de hablar de dominio, tiene sentido solo en los procesos tradicionales/definidos.

El dominio de las métricas de soft se divide en esos 3 elementos. Yo puedo elegir aspectos que ami me interese medir sobre el proceso. Uno mide porque quiere tener visibilidad sobre cierta situación, porque quiere aumentar su nivel de conocimiento sobre algún aspecto. Las métricas nos ayudan a tomar decisiones informadas. Si uno tiene información para decidir decide mejor.

### MÉTRICAS DE PRODUCTO

Tienen que ver en términos concretos con el producto de software. Son métricas que nos permiten observar que ocurre con el producto de software que estamos construyendo. Casi todas las métricas que apuntan a defectos son métricas de producto.

Cuando hablamos de producto tiene que medir cosas relacionadas al producto. Lineas de código, complejidad ciclomática (interna), cu, users, módulos, alcances, paquetes. En cuanto al tamaño del producto. Y después medir defectos. Defectos por nivel (cuantos defectos en el test unitario, de integración, etc), por severidad, densidad de defectos, cobertura del testing (es un porcentaje de lo que testee sobre la totalidad de lo que había que testear).

### MÉTRICAS DE PROYECTO (TÁCTICAS)

Tienen su aplicación en la ejecución del **proyecto** que nos permite construir un determinado producto de software. En este tipo de métricas, la utilidad de la métrica termina cuando el proyecto termina. En el contexto del proyecto, la utilidad va a tener que ver con lograr corregir algo que en el proyecto no esté funcionando bien.

Estas son de interés para el Líder de Proyecto y para el equipo, ya que permite ver justamente cual es el estado actual del proyecto, su salud, y que tenemos que hacer si algo no está funcionando bien.

Las métricas de proyecto pueden dividirse en cuatro métricas básicas:

- △ **Tamaño del producto:** Métrica del producto, medir en términos concretos que tan grande o que tan pequeño es el producto que estoy construyendo.
- △ **Esfuerzo (Horas lineales):** Métrica del proceso y proyecto.
- △ **Tiempo (Calendario):** Métrica del proyecto y proceso.
- △ **Defectos:** Métrica del producto

Medir proyecto vincularlo con la triple restricción, recursos, costos, calendarios. Las métricas tienen que ver con dar visibilidad de la situación de este producto o servicio que tenemos que entregar, como estamos funcionando para entregarlo. Esfuerzo, esfuerzo total, esfuerzo por etapas, esfuerzo de cada una de las personas, diferencias del esfuerzo estimado con el real. Lo mismo con el calendario, cuanto dijimos que iba a dura y cuanto duro, el tiempo de cada iteración.

Dentro de las métricas básicas tenemos el tamaño del producto, que es imprescindible comprender que producto y trabajo no es lo mismo. Una cosa es el tamaño del producto y otro el esfuerzo que voy a necesitar; están relacionados, producto mas grande requiere mas horas, pero no es lo mismo el tamaño del producto que horas ideales, se mide el tamaño del producto. Las líneas de código no son muy útiles, no se pueden estimar desde un principio. Reqs en la forma que cada uno maneje los reqs, se pueden contar funcionalidades, alcance, cu, opciones de menú, como cada uno le llame a ese concepto de funcionalidad, se elige eso. En el PUD el tamaño normalmente es tener métricas de cu por complejidad. Porque también contar cu solos tampoco sirve mucho porque no todos los cu son del mismo tamaño. Cuando uno cuenta cu por complejidad te das cuenta que la mayoría son simples o livianos y después algunos otros complejos, que son menos pero tienen mas peso. Por eso es bueno hacer las métricas tratando de utilizar un valor homogéneo, que es mas o menos lo que hace agile con las users y los sp. Los sp homogenizan el valor de la diferencia de puntos entre las users.

El esfuerzo se mide en horas persona lineales. Es asumir que es una persona la que hace el trabajo haciendo una cosa por vez. No se tiene en cuenta el trabajo paralelo, el índice de solapamiento y la cantidad de personas. Esa es la diferencia entre el esfuerzo y el calendario. El esfuerzo es una bolsa de horas, donde no tengo en cuenta si puedo hacer mas de una cosa a la vez ni cuantas personas voy a tener. Cuando incorporo mas gente a trabajar tengo que tener en cuenta que agrego una complejidad que da la interacción entre las partes. No se puede derivar desde el esfuerzo el calendario. El esfuerzo son horas personas lineales. Asumo que nouento tiempo muerto, eso no se tiene en cuenta, y asumo que trabaja una sola persona haciendo solo una cosa por vez. Ese esfuerzo va a tomarse como base para derivar el calendario, donde se tienen en cuenta índice de solapamiento, cuanta gente va a trabajar, etc, entonces se arma el calendario, y dice para esta fecha estaría entregando el producto. El tiempo responde al cuando, el esfuerzo al como, y el tamaño al que. Uno se usa como base para derivar al siguiente. Es una relación casi lógica. Cuando elegimos métricas para cada uno tenemos que tener claro que queremos medir en cada uno

El calendario se mide de acuerdo a la cultura organizacional. Algunos lo miden en horas pero eso es confuso con respecto al esfuerzo. Mas bien medirlo en semanas, meses, dependiendo de la envergadura.

Los defectos miden sobre el producto cuales son las cosas que se detectaron que no son coincidentes con lo que se espera. Pasa algo parecido que con los cu. Tener en cuenta la severidad. Cantidad de defectos por severidad, densidad de defectos. No es lo mismo un error bloqueante que uno de cosmética.

### MÉTRICAS DE PROCESO (ORGANIZACIONALES O ESTRATÉGICAS)

Las métricas de proceso son una despersonalización de las métricas de proyecto, permiten ver en términos organizacionales como trabajamos en el conjunto de nuestros proyectos. Si para nosotros una métrica de proyecto es ver el desvió del calendario en función de lo planificado, en términos de proceso lo que hacemos es tomar muchas métricas de proyecto despersonalizando la métrica del proyecto en particular y apuntando a la organización completa, es decir, midiendo en promedio cual fue el desvio de los proyectos de la organización.

Busco ver si los proyectos de la organización terminan en tiempo y forma. Si tengo un desvio general de los proyectos, todo lo que tenga que ver con las actividades de planificaciones van a tener modificaciones para ajustar ese desvio y hacer que los proyectos terminen en tiempo y forma.

Estas métricas le permiten a la organización saber dónde pueden aplicar mejoras, saber cómo trasladar esto de la mejora continua.

tradicional yo puedo usar la experiencia de otros proyectos para los que vienen. Con esa línea usan las métricas de proceso. Las de producto son privadas, las de proceso son publicas. Uno las construye para que toda la org tenga visibilidad de la situación de la org. Puedo armar métricas de proceso desde las de proyecto o desde las de producto. Necesito hacer un trabajo de consolidación y despersonalización de los datos. Significa que para que una métrica sea publica y de proceso, no tiene que tener atribución a ningún proceso ni proyecot, es una métrica despojada de vinculación. Si quiero decir la cantidad de defectos por severidad que tiene la org relacionado a sus productos, es de proceso. No estoy hablando de ningún producto ni proyecto en particular. La org tiene una desviación de lo estimado en función de lo ejecutado del 80%, estoy hablando de la org, no de un proyecto o producto. Entonces tomo datos de muchos proyectos, saco promedio, despersonalizo y publico. Se pueden plantear también métricas específicas para proceso, que midan su eficiencia, aspectos de capacidad o madurez del proceso. Tengo métricas de proceso despersonalizando y consolidando las de producto o las de proyecto.

Las métricas del proyecto y del proceso podemos decir que son las mismas mostradas de distinta manera. Las métricas de proyecto pueden convertirse en métricas de proceso.

La diferencia es hacia qué apunto.

### EJEMPLOS DE MÉTRICAS:

| Desarrollador   | Equipo de Desarrollo  |
|---|---|
| 1. Esfuerzo   | 1. Tiempo del producto  |
| 2. Esfuerzo y duración estimada y actual de una tarea.          | 2. Desvió entre estimado y actual entre las tareas más importantes.     |
| 3. % de cobertura por el unit test.                             | 3. Niveles de staffing reales y establecidos.                           |
| 4. Número y tipo de defectos encontrados en el unit test.       | 4. Nro. de tareas planificadas y completadas.                           |
| 5. Número y tipo de defectos encontrados en revisión por pares. | 5. Distribución del esfuerzo  |
| Organización  |   |
| 1. Tiempo Calendario  | 6. Status de requerimientos.  |
| 2. Performance actual y planificado de esfuerzo.                | 7. Vulnerabilidad de requerimientos.                                    |
| 3. Performance actual y planificado de presupuesto              | 8. Nro. de defectos encontrados en la integración y prueba de sistemas. |
| 4. Precisión de estimaciones en Schedule y esfuerzo             | 9. Nro. de defectos encontrados en peer review.                         |
| 5. Defectos en Fallos   | 10. Status de distribución de defectos.                                 |
|   | 11. % de test ejecutados.   |

#### **TIPS A TENER EN CUENTA:**

La óptica con la que nosotros vemos, si tenemos un proyecto que estimamos desde su inicio a su final tenemos planificado un cronograma de 10 meses, no tiene sentido medir su desvío todos los días. El esfuerzo de hacer esa medición, por sobre lo que voy a evaluar no tiene sentido.



Si estas a miles de distancia de tu destino, no tiene sentido medir en milímetros

No sirve tener tantas métricas porque se pierde mucho tiempo en recolectarlas y después no llegan a analizarse. El tiempo que me demoro para obtener la métrica debe ser el mínimo indispensable. No todas las métricas útiles para un proyecto significan que sean útiles para otros.

Al momento de elegir una métrica, debemos preguntarnos en el contexto actual:

¿Nos da más información que la que tenemos ahora? ¿Es esta información de beneficio práctico?  
¿Nos dice lo que queremos saber?

Uno podría crear muchísimas métricas. Pero sirven a alguien para un propósito y las necesidades de métricas varían de acuerdo al que este trabajando en el proyecto y el rol que debe cumplir. Metricas por perfil, ej al desarrollador le interesan ciertas métricas, a la organización otras. Cuando uno va a armar un esquema de métricas para una organización tiene que interiorizarse de las necesidades de info y cuales son las expectativas de la gente, siempre teniendo en cuenta que esto debe mantenerse simple. No muy complejas las métricas, ni cosas rebuscadas. Metricas que se puedan representar con un número, que se puedan medir. Si definis métricas con costo muy alto tampoco sirven. Hay que tener siempre un foco en mantenerlo simple y que las métricas que vamos a definir sean métricas que la gente necesita y que va a usar.

Es importante considerar también el **esfuerzo** involucrado en la obtención de métricas. No se puede dedicar la mayor parte de las horas productivas a recopilar métricas, ya que esto puede afectar negativamente la eficiencia del equipo y el cumplimiento de los plazos. Es necesario encontrar un equilibrio y evaluar el esfuerzo requerido para recopilar las métricas en comparación con los beneficios que se obtendrán.

En ocasiones, es preferible utilizar un enfoque selectivo, **analizando solo unas pocas métricas relevantes** que nos obliguen a profundizar en los aspectos clave que deseamos medir. Esto nos permite centrarnos en los datos más significativos y evitar la sobrecarga de información.

Al mismo tiempo, es importante **recordar la triple restricción** del proyecto: alcance, tiempo y recursos. Estos elementos deben medirse y gestionarse como líder de proyecto. Las métricas pueden ser útiles para evaluar el progreso del proyecto, identificar posibles desviaciones y tomar decisiones informadas para garantizar que el proyecto avance de manera exitosa.

## **MÉTRICAS DE SOFTWARE EN AMBIENTES ÁGILES**

**Basadas en Procesos Empíricos**

En la filosofía Agile todo es minimalista, y por lo tanto **sólo mido lo estrictamente necesario y nada más**. A diferencia de la gestión tradicional, donde se debe reflexionar sobre qué aspectos medir, en el enfoque ágil se establecen pautas claras sobre qué métricas son relevantes.

La visión sobre las métricas en agile es distinta. No hay actividades específicas vinculadas a quien toma las métricas, sino que esta integrando, es una consecuencia o salida mas del trabajo. Una visión liviana de que medir. Sobre todo por el principio que dice que la mejor métrica de progreso es el software funcionando.

Hay dos Principios Agiles que guían la elección de las métricas:

**“El software funcionando es la principal medida del progreso”**

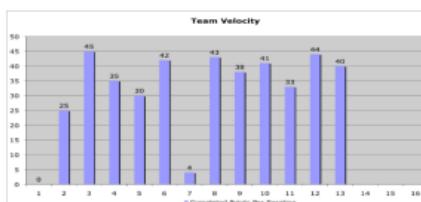
Este principio resalta que la mejor manera de evaluar el avance de un proyecto es a través del software que está efectivamente funcionando y entregando valor. En lugar de enfocarse únicamente en la planificación y las métricas teóricas, se valora la realidad tangible y tangible del software en funcionamiento como indicador principal de progreso.

**“La mayor prioridad es satisfacer al cliente por medio de entregas tempranas y continuas de software valiosos, funcionando”**

Este principio subraya la importancia de entregar rápidamente software de calidad que brinde valor al cliente. En lugar de esperar a tener un producto totalmente terminado, se busca realizar entregas tempranas y frecuentes de software funcional y valioso. Esto permite obtener retroalimentación temprana del cliente y adaptar el producto en consecuencia.

#### VELOCITY (VELOCIDAD)

Es una medida de la cantidad de trabajo que un equipo puede completar en un período de tiempo determinado. Se calcula sumando el esfuerzo estimado de las historias de usuario o tareas completadas en cada iteración o sprint. La velocidad se utiliza para **predecir la capacidad del equipo** y estimar la duración futura del proyecto.



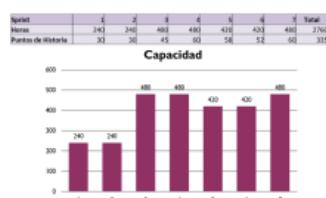
Son la cantidad de puntos de historia que el equipo terminó y que el PO aprobó en la review.

La principal métrica que tiene ágil es la velocidad. No asumir que la experiencia es repetible. La velocidad no necesariamente es la misma para cada sprint. La velocidad es una métrica que mide cuánto producto pudimos entregar al final de una iteración y el po los acepto. La cantidad de sp aceptados por el po en un sprint. Esa es la unidad básica. Se mide en puntos de historia, porque medimos producto. No se mide en horas porque las horas son esfuerzo. La velocidad no se estima, se calcula al final del sprint con la cantidad de sp de las users que el po me acepto.

La velocidad a medida que se avanza en los sprint debería tender a estabilizarse. Es normal que en los primeros sprint la velocidad sea 0 o muy baja, ya que el equipo se está conociendo, a medida que el equipo trabaja juntos y adquiere un ritmo la velocidad debería tender a estabilizarse. Esta velocidad que se estabiliza nos va a servir para ayudarnos a determinar la **CAPACIDAD** del equipo.

#### CAPACIDAD

La capacidad si bien está en la lista de métricas es una **estimación** de cuánto trabajo (puntos de historia) va a poder completar el equipo en el próximo sprint. Por ello, cuando tengo una velocidad estable y tengo en claro los recursos que tengo disponibles, es fácil determinar la capacidad, y en la planning definir qué features voy a incluir en el sprint backlog. En los primeros sprints, cuando no está clara la velocidad del equipo, puedo definir la capacidad en horas lineales.



Capacidad, si se puede estimar. Para determinar cuánto nos podemos comprometer como equipo en un sprint. Se puede medir en horas ideales o en sp. Es aceptable en ambos casos que no hay una tabla de conversión, tantas horas son un sp, eso no existe. No hay una relación directa entre puntos de historia y horas ideales. Depende lo que se sienta cómodo el equipo. Equipos más maduros

estiman directamente en sp. De lo contrario hacen el calculo de capacidad viendo cuantas horas puede trabajar cada persona del equipo día por día de la duración del sprint.

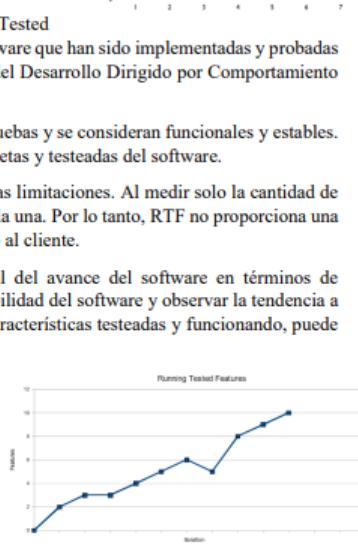
**RUNNING TESTED FEATURES (RTF):** La métrica "Running Tested Features" (RTF) se refiere a la cantidad de características del software que han sido implementadas y probadas de manera exitosa. Es una medida que se utiliza en el contexto del Desarrollo Dirigido por Comportamiento (BDD) y Agile.

RTF se enfoca en contar las características que han pasado las pruebas y se consideran funcionales y estables. Representa el progreso en términos de las funcionalidades completas y testeadas del software.

Sin embargo, es importante tener en cuenta que RTF tiene algunas limitaciones. Al medir solo la cantidad de características, no se considera el tamaño o la complejidad de cada una. Por lo tanto, RTF no proporciona una representación precisa del trabajo realizado o del valor entregado al cliente.

La métrica de RTF se utiliza para obtener una visión general del avance del software en términos de características completas y testeadas. Es útil para evaluar la estabilidad del software y observar la tendencia a lo largo del tiempo. Si se mantiene una tendencia creciente de características testeadas y funcionando, puede indicar un progreso constante en el desarrollo.

Sin embargo, es recomendable complementar la métrica de RTF con otras medidas, como la velocidad, para obtener una evaluación más completa del progreso y el valor entregado. Esta métrica sola, lo único que me permitiría ver es que si mantengo una tendencia creciente de software funcionando a lo largo del tiempo. También es importante considerar el contexto completo del proyecto y tener en cuenta las limitaciones inherentes a esta métrica.



RTF, no se usa mucho. Cuenta la cantidad de características que están funcionando. Features por iteración. Contando cuantas características entregue por sprint. Es parecida a la velocidad, reemplazando features por sps. No hay un acuerdo claro de que es feature, si es mas grande o mas pequeño que una user. Mismo problema cuando contábamos cu y no cu por complejidad. Cuando contas características y no tenes en cuenta el tamaño o la complejidad de las características, es una métrica pero que indicador puedo construir de avance, el valor de esto por donde pasa, muchas features pequeñas y una grande y da distinto el grafico.

## MÉTRICAS EN KANBAN PARA PROCESOS EMPIRICOS CON ENFOQUE LEAN

**Basadas en Procesos Empíricos**

(Ya las vimos antes, son Cycle Time, Lead Time y Touch Time)

*En resumen, en término de gestión de proyectos*

- |  |  |   |
|--|--|---|
| <ul style="list-style-type: none"><li>▪ Tradicionales</li><li>▪ Esfuerzo</li><li>▪ Tiempo</li><li>▪ Costos</li><li>▪ Riesgos</li></ul> | <ul style="list-style-type: none"><li>▪ Ágiles</li><li>▪ Velocidad</li><li>▪ Capacidad</li><li>▪ Running Tested Features</li></ul> | <ul style="list-style-type: none"><li>▪ Lean</li><li>▪ Lead Time</li><li>▪ Cycle Time</li><li>▪ Touch Time</li><li>▪ Eficiencia Proceso</li></ul> |
|--|--|---|

### *Resumen, en términos de gestión de productos*



**El análisis de las métricas no tiene que ver con juzgar o encontrar errores en las personas, el foco es mejorar,** porque además mido porque si no mido no sé dónde estoy parado, Ejemplo: Si hago una dieta, sino me peso, o medido las diferentes variables no sé si voy mejorando o no.

## Definición de Scrum

Scrum es un marco de trabajo liviano que ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptativas para problemas complejos.

Scrum es un framework para gestión porque básicamente no apunta a ninguna actividad de ingeniería, no habla de software. Se quiere extender a cualquier ámbito de trabajo.

En pocas palabras, Scrum requiere un *Scrum Master* para fomentar un entorno donde:

1. Un *Product Owner* ordena el trabajo de un problema complejo en un *Product Backlog*.
2. El *Scrum Team* convierte una selección del trabajo en un *Increment* de valor durante un *Sprint*.
3. El *Scrum Team* y sus interesados inspeccionan los resultados y se adaptan para el próximo *Sprint*.
4. *Repita*

El marco de trabajo Scrum es incompleto de manera intencional, solo define las partes necesarias para implementar la teoría de Scrum. Scrum se basa en la inteligencia colectiva de las personas que lo utilizan. En lugar de proporcionar a las personas instrucciones detalladas, las reglas de Scrum guían sus relaciones e interacciones.

## Teoría de Scrum

Scrum se basa en el empirismo y el pensamiento *Lean*. El empirismo afirma que el conocimiento proviene de la experiencia y de la toma de decisiones con base en lo observado. El pensamiento *Lean* reduce el desperdicio y se enfoca en lo esencial.

Scrum emplea un enfoque iterativo e *Incremental* para optimizar la previsibilidad y controlar el riesgo. Scrum involucra a grupos de personas que colectivamente tienen todas las habilidades y experiencia para hacer el trabajo y compartir o adquirir dichas habilidades según sea necesario.

Scrum combina cuatro eventos formales para inspección y adaptación dentro de un evento contenedor, el *Sprint*. Estos eventos funcionan porque implementan los pilares empíricos de Scrum de transparencia, inspección y adaptación.

### Transparencia

El proceso y el trabajo emergentes deben ser visibles tanto para quienes realizan el trabajo como para quienes lo reciben. Con Scrum, las decisiones importantes se basan en el estado percibido de sus tres

artefactos formales. Los artefactos que tienen poca transparencia pueden llevar a decisiones que disminuyan el valor y aumenten el riesgo.

La transparencia permite la inspección. La inspección sin transparencia es engañosa y derrochadora.

### Inspección

Los artefactos de Scrum y el progreso hacia los objetivos acordados deben inspeccionarse con frecuencia y con diligencia para detectar variaciones o problemas potencialmente indeseables. Para ayudar con la inspección, Scrum proporciona cadencia en forma de sus cinco eventos.

La inspección permite la adaptación. La inspección sin adaptación se considera inútil. Los eventos Scrum están diseñados para provocar cambios.

## Adaptación

Si algún aspecto de un proceso se desvía fuera de los límites aceptables o si el producto resultante es inaceptable, el proceso que se aplica o los materiales que se producen deben ajustarse. El ajuste debe realizarse lo antes posible para minimizar una mayor desviación.

La adaptación se vuelve más difícil cuando las personas involucradas no están empoderadas ni se autogestionan. Se espera que un *Scrum Team* se adapte en el momento en que aprenda algo nuevo a través de la inspección.

## Valores de Scrum

El uso exitoso de Scrum depende de que las personas se vuelvan más competentes en vivir cinco valores:

### ***Compromiso, Foco, Franqueza, Respeto y Coraje***

El *Scrum Team* se compromete a lograr sus objetivos y a apoyarse mutuamente. Su foco principal está en el trabajo del *Sprint* para lograr el mejor progreso posible hacia estos objetivos. El *Scrum Team* y sus interesados son francos sobre el trabajo y los desafíos. Los miembros del *Scrum Team* se respetan entre sí para ser personas capaces e independientes, y son respetados como tales por las personas con las que trabajan. Los miembros del *Scrum Team* tienen el coraje de hacer lo correcto, para trabajar en problemas difíciles.

Estos valores dan dirección al *Scrum Team* con respecto a su trabajo, acciones y comportamiento. Las decisiones que se tomen, los pasos que se den y la forma en que se use Scrum deben reforzar estos valores, no disminuirlos ni socavarlos. Los miembros del *Scrum Team* aprenden y exploran los valores mientras trabajan con los eventos y artefactos Scrum. Cuando el *Scrum Team* y las personas con las que

trabajan incorporan estos valores, los pilares empíricos de Scrum de transparencia, inspección y adaptación cobran vida y generan confianza.

## **Scrum Team**

La unidad fundamental de Scrum es un pequeño equipo de personas, un *Scrum Team*. El *Scrum Team* consta de un *Scrum Master*, un *Product Owner* y *Developers*. Dentro de un *Scrum Team*, no hay subequipo ni jerarquías. Es una unidad cohesionada de profesionales enfocados en un objetivo a la vez, el Objetivo del Producto.

Los *Scrum Teams* son multifuncionales, lo que significa que los miembros tienen todas las habilidades necesarias para crear valor en cada *Sprint*. También se autogestionan, lo que significa que deciden internamente quién hace qué, cuándo y cómo.

El *Scrum Team* es lo suficientemente pequeño como para seguir siendo ágil y lo suficientemente grande como para completar un trabajo significativo dentro de un *Sprint*, generalmente 10 personas o menos. En general, hemos descubierto que los equipos más pequeños se comunican mejor y son más productivos. Si los *Scrum Teams* se vuelven demasiado grandes, deberían considerar reorganizarse en múltiples *Scrum Teams* cohesivos, cada uno enfocado en el mismo producto. Por lo tanto, deben compartir el mismo Objetivo del Producto, el *Product Backlog* y el *Product Owner*.

El *Scrum Team* es responsable de todas las actividades relacionadas con el producto, desde la colaboración de los interesados, la verificación, el mantenimiento, la operación, la experimentación, la investigación y el desarrollo, y cualquier otra cosa que pueda ser necesaria. Están estructurados y empoderados por la organización para gestionar su propio trabajo. Trabajar en *Sprints* a un ritmo sostenible mejora el enfoque y la consistencia del *Scrum Team*.

Todo el *Scrum Team* es responsable de crear un *Increment* valioso y útil en cada *Sprint*. Scrum define tres responsabilidades específicas dentro del *Scrum Team*: los *Developers*, el *Product Owner* y el *Scrum Master*.

## **Developers**

Las personas del *Scrum Team* que se comprometen a crear cualquier aspecto de un *Increment* utilizable en cada *Sprint* son *Developers*.

Las habilidades específicas que necesitan los *Developers* suelen ser amplias y variarán según el ámbito de trabajo. Sin embargo, los *Developers* siempre son responsables de:

- Crear un plan para el *Sprint*, el *Sprint Backlog*;
  - Inculcar calidad al adherirse a una Definición de Terminado;
  - Adaptar su plan cada día hacia el Objetivo del *Sprint*; y,
- 
- Responsabilizarse mutuamente como profesionales.

## **Product Owner**

El *Product Owner* es responsable de maximizar el valor del producto resultante del trabajo del *Scrum Team*. La forma en que esto se hace puede variar ampliamente entre organizaciones, *Scrum Teams* e individuos.

El *Product Owner* también es responsable de la gestión efectiva del *Product Backlog*, lo que incluye:

- Desarrollar y comunicar explícitamente el Objetivo del Producto;
- Crear y comunicar claramente los elementos del *Product Backlog*;
- Ordenar los elementos del *Product Backlog*; y,
- Asegurarse de que el *Product Backlog* sea transparente, visible y se entienda.

El *Product Owner* es una persona, no un comité. El *Product Owner* puede representar las necesidades de muchos interesados en el *Product Backlog*. Aquellos que quieran cambiar el *Product Backlog* pueden hacerlo intentando convencer al *Product Owner*.

## **Scrum Master**

El *Scrum Master* es responsable de establecer Scrum como se define en la Guía de Scrum. Lo hace ayudando a todos a comprender la teoría y la práctica de Scrum, tanto dentro del *Scrum Team* como de la organización.

El *Scrum Master* es responsable de lograr la efectividad del *Scrum Team*. Lo hace apoyando al *Scrum Team* en la mejora de sus prácticas, dentro del marco de trabajo de Scrum.

Los *Scrum Masters* son verdaderos líderes que sirven al *Scrum Team* y a la organización en general.

El *Scrum Master* sirve al *Scrum Team* de varias maneras, que incluyen:

- Guiar a los miembros del equipo en ser autogestionados y multifuncionales;
- Ayudar al *Scrum Team* a enfocarse en crear *Increments* de alto valor que cumplan con la Definición de Terminado;

- Procurar la eliminación de impedimentos para el progreso del *Scrum Team*; y,
- Asegurarse de que todos los eventos de Scrum se lleven a cabo y sean positivos, productivos y se mantengan dentro de los límites de tiempo recomendados en esta Guía.

El *Scrum Master* sirve al *Product Owner* de varias maneras, que incluyen:

- Ayudar a encontrar técnicas para una definición efectiva de Objetivos del Producto y la gestión del *Product Backlog*;
- Ayudar al *Scrum Team* a comprender la necesidad de tener elementos del *Product Backlog* claros y concisos;
- Ayudar a establecer una planificación empírica de productos para un entorno complejo; y,
- Facilitar la colaboración de los interesados según se solicite o necesite.

El *Scrum Master* sirve a la organización de varias maneras, que incluyen:

- Liderar, capacitar y guiar a la organización en su adopción de Scrum;
- Planificar y asesorar implementaciones de Scrum dentro de la organización;
- Ayudar a los empleados y los interesados a comprender y aplicar un enfoque empírico para el trabajo complejo; y,
- Eliminar las barreras entre los interesados y los *Scrum Teams*.

## Eventos de Scrum

El *Sprint* es un contenedor para todos los demás eventos. Cada evento en Scrum es una oportunidad formal para inspeccionar y adaptar los artefactos Scrum. Estos eventos están diseñados específicamente para habilitar la transparencia requerida. No operar cualquier evento según lo prescrito resulta en la pérdida de oportunidades para inspeccionar y adaptarse. Los eventos se utilizan en Scrum para crear regularidad y minimizar la necesidad de reuniones no definidas en Scrum.

Lo óptimo es que todos los eventos se celebren al mismo tiempo y en el mismo lugar para reducir la complejidad.

### El *Sprint*

Los *Sprints* son el corazón de Scrum, donde las ideas se convierten en valor.

Son eventos de duración fija de un mes o menos para crear consistencia. Un nuevo *Sprint* comienza inmediatamente después de la conclusión del *Sprint* anterior.

Todo el trabajo necesario para lograr el Objetivo del Producto, incluido la *Sprint Planning*, *Daily Scrums*, *Sprint Review* y *Sprint Retrospective*, ocurre dentro de los *Sprints*.

Durante el *Sprint*:

- No se realizan cambios que pongan en peligro el Objetivo del *Sprint*;
- La calidad no disminuye;
- El *Product Backlog* se refina según sea necesario; y,
- El alcance se puede aclarar y renegociar con el *Product Owner* a medida que se aprende más.

Los *Sprints* permiten la previsibilidad al garantizar la inspección y adaptación del progreso hacia un Objetivo del Producto al menos cada mes calendario. Cuando el horizonte de un *Sprint* es demasiado largo, el Objetivo del *Sprint* puede volverse inválido, la complejidad puede crecer y el riesgo puede aumentar. Se pueden emplear *Sprints* más cortos para generar más ciclos de aprendizaje y limitar el riesgo de costo y esfuerzo a un período de tiempo menor. Cada *Sprint* puede considerarse un proyecto corto.

Existen varias prácticas para pronosticar el progreso, como el trabajo pendiente (*burn-downs*), trabajo completado (*burn-ups*) o flujos acumulativos (*cumulative flows*). Si bien han demostrado su utilidad, no reemplazan la importancia del empirismo. En entornos complejos, se desconoce lo que sucederá. Solo lo que ya ha sucedido se puede utilizar para la toma de decisiones con miras al futuro.

Un *Sprint* podría cancelarse si el Objetivo del *Sprint* se vuelve obsoleto. Solo el *Product Owner* tiene la autoridad para cancelar el *Sprint*.

El tiempo del sprint no se puede cambiar durante el sprint, porque ahí rompes el time box, si te puedes pasar que cancelas un sprint, si un incremento ya no lo necesitas mas no lo seguis.

### *Sprint Planning*

La *Sprint Planning* inicia el *Sprint* al establecer el trabajo que se realizará para el *Sprint*. El *Scrum Team* crea este plan resultante mediante trabajo colaborativo.

El *Product Owner* se asegura de que los asistentes estén preparados para discutir los elementos más importantes del *Product Backlog* y cómo se relacionan con el Objetivo del Producto. El *Scrum Team* también puede invitar a otras personas a asistir a la *Sprint Planning* para brindar asesoramiento.

La *Sprint Planning* aborda los siguientes temas:

Tema uno: ¿Por qué es valioso este *Sprint*?

El *Product Owner* propone cómo el producto podría *Incrementar* su valor y utilidad en el *Sprint* actual. Luego, todo el *Scrum Team* colabora para definir un Objetivo del *Sprint* que comunica por qué el *Sprint* es valioso para los interesados. El Objetivo del *Sprint* debe completarse antes de que termine la *Sprint Planning*.

Tema dos: ¿Qué se puede hacer en este *Sprint*?

A través de una conversación con el *Product Owner*, los *Developers* seleccionan elementos del *Product Backlog* para incluirlos en el *Sprint* actual. El *Scrum Team* puede refinar estos elementos durante este proceso, lo que aumenta la comprensión y la confianza.

Seleccionar cuánto se puede completar dentro de un *Sprint* puede ser un desafío. Sin embargo, cuanto más sepan los *Developers* sobre su desempeño pasado, su capacidad actual y su Definición de Terminado, más confiados estarán en sus pronósticos para el *Sprint*.

#### Tema tres: ¿Cómo se realizará el trabajo elegido?

Para cada elemento del *Product Backlog* seleccionado, los *Developers* planifican el trabajo necesario para crear un *Increment* que cumpla con la Definición de Terminado. A menudo, esto se hace descomponiendo los elementos del *Product Backlog* en elementos de trabajo más pequeños de un día o menos. La forma de hacerlo queda a criterio exclusivo de los *Developers*. Nadie más les dice cómo convertir los elementos del *Product Backlog* en *Increments* de valor.

El Objetivo del *Sprint*, los elementos del *Product Backlog* seleccionados para el *Sprint*, más el plan para entregarlos se denominan juntos *Sprint Backlog*.

La *Sprint Planning* tiene un límite de tiempo de máximo ocho horas para un *Sprint* de un mes. Para *Sprints* más cortos, el evento suele ser de menor duración.

### Daily Scrum

El propósito de la *Daily Scrum* es inspeccionar el progreso hacia el Objetivo del *Sprint* y adaptar el *Sprint Backlog* según sea necesario, ajustando el trabajo planificado entrante.

La *Daily Scrum* es un evento de 15 minutos para los *Developers* del *Scrum Team*. Para reducir la complejidad, se lleva a cabo a la misma hora y en el mismo lugar todos los días hábiles del *Sprint*. Si el *Product Owner* o *Scrum Master* están trabajando activamente en elementos del *Sprint Backlog*, participan como *Developers*.

Los *Developers* pueden seleccionar la estructura y las técnicas que deseen, siempre que su *Daily Scrum* se centre en el progreso hacia el Objetivo del *Sprint* y produzca un plan viable para el siguiente día de trabajo. Esto crea enfoque y mejora la autogestión.

Las *Daily Scrums* mejoran la comunicación, identifican impedimentos, promueven la toma rápida de decisiones y, en consecuencia, eliminan la necesidad de otras reuniones.

La *Daily Scrum* no es el único momento en el que los *Developers* pueden ajustar su plan. A menudo se reúnen durante el día para discusiones más detalladas sobre cómo adaptar o volver a planificar el resto del trabajo del *Sprint*.

En la daily hay inspección y adaptación. El propósito de la daily todos hablan y lo que dicen es concreto, que hiciste en que estas trabajando que pensar hacer. El equipo tiene libertad de hablar o preguntar de otra manera. Para que no se resista el equipo a la daily. La necesidad sigue existiendo, no sacarlas, sino aprender a hacerlas bien, y en ese contexto la daily va a servir.

## **Sprint Review**

El propósito de la *Sprint Review* es inspeccionar el resultado del *Sprint* y determinar futuras adaptaciones. El *Scrum Team* presenta los resultados de su trabajo a los interesados clave y se discute el progreso hacia el Objetivo del Producto.

Durante el evento, el *Scrum Team* y los interesados revisan lo que se logró en el *Sprint* y lo que ha cambiado en su entorno. Con base en esta información, los asistentes colaboran sobre qué hacer a continuación. El *Product Backlog* también se puede ajustar para satisfacer nuevas oportunidades. La *Sprint Review* es una sesión de trabajo y el *Scrum Team* debe evitar limitarla a una presentación.

La *Sprint Review* es el penúltimo evento del *Sprint* y tiene un límite de tiempo de máximo cuatro horas para un *Sprint* de un mes. Para *Sprints* más cortos, el evento suele ser de menor duración.

## **Sprint Retrospective**

El propósito de la *Sprint Retrospective* es planificar formas de aumentar la calidad y la efectividad.

El *Scrum Team* inspecciona cómo fue el último *Sprint* con respecto a las personas, las interacciones, los procesos, las herramientas y su Definición de Terminado. Los elementos inspeccionados suelen variar según el ámbito del trabajo. Se identifican los supuestos que los llevaron por mal camino y se exploran sus orígenes. El *Scrum Team* analiza qué salió bien durante el *Sprint*, qué problemas encontró y cómo se resolvieron (o no) esos problemas.

El *Scrum Team* identifica los cambios más útiles para mejorar su efectividad. Las mejoras más impactantes se abordan lo antes posible. Incluso se pueden agregar al *Sprint Backlog* para el próximo *Sprint*.

La *Sprint Retrospective* concluye el *Sprint*. Tiene un tiempo limitado a máximo tres horas para un *Sprint* de un mes. Para *Sprints* más cortos, el evento suele ser de menor duración.

La retrospectiva vemos que hicimos bien que hicimos mal, con el propósito de mejoras sobre el proceso. En la review se mejor producto, en la retrospectiva el proceso. Usamos los pilares del empirismo: transparencia, adaptación, introspección.

## **Refinamiento del pb**

Esta última se utiliza en demanda, cuando se necesita. Es el refinamiento del producto backlog. Antes se llamaba grooming. Es continua, todo el tiempo estamos haciendo esto, on demand, vamos a armar una reunión de refinamiento cada vez que lo necesitemos.

El refinamiento del pb esta planteada en términos porcentuales. No hay un momento exacto fijo determinado como si el resto de las ceremonias. No es opcional. Se hace en cualquier momento, cuando haga falta, esta definida como una actividad continua. Tienen un porcentaje sobre la duración del sprint. Hay que dejar un espacio en la jornada del trabajo para refinar el pb

## Artefactos de Scrum

Los artefactos de Scrum representan trabajo o valor. Están diseñados para maximizar la transparencia de la información clave. Por lo tanto, todas las personas que los inspeccionan tienen la misma base de adaptación.

Cada artefacto contiene un compromiso para garantizar que proporcione información que mejore la transparencia y el enfoque frente al cual se pueda medir el progreso:

- Para el *Product Backlog*, es el Objetivo del Producto.
- Para el *Sprint Backlog*, es el Objetivo del Sprint.

10

- Para el *Increment* es la Definición de Terminado.

Estos compromisos existen para reforzar el empirismo y los valores de Scrum para el *Scrum Team* y sus interesados.

### Product Backlog

El *Product Backlog* es una lista emergente y ordenada de lo que se necesita para mejorar el producto. Es la única fuente del trabajo realizado por el *Scrum Team*.

Los elementos del *Product Backlog* que el *Scrum Team* puede dar por Terminados dentro de un *Sprint* se consideran preparados para ser seleccionados en un evento de *Sprint Planning*. Suelen adquirir este grado de transparencia tras las actividades de refinamiento. El refinamiento del *Product Backlog* es el acto de dividir y definir aún más los elementos del *Product Backlog* en elementos más pequeños y precisos. Esta es una actividad continua para agregar detalles, como una descripción, orden y tamaño. Los atributos suelen variar según el ámbito del trabajo.

Los *Developers* que realizarán el trabajo son responsables del dimensionamiento. El *Product Owner* puede influir en los *Developers* ayudándolos a entender y seleccionar sus mejores alternativas.

### **Compromiso: Objetivo del Producto**

El Objetivo del Producto describe un estado futuro del producto que puede servir como un objetivo para que el *Scrum Team* planifique. El Objetivo del Producto está en el *Product Backlog*. El resto del *Product Backlog* emerge para definir "qué" cumplirá con el Objetivo del Producto.

*Un producto es un vehículo para entregar valor. Tiene un límite claro, personas interesadas conocidas, usuarios o clientes bien definidos. Un producto puede ser un servicio, un producto físico o algo más abstracto.*

El Objetivo del Producto es el objetivo a largo plazo del *Scrum Team*. Ellos deben cumplir (o abandonar) un objetivo antes de asumir el siguiente.

### **Sprint Backlog**

El *Sprint Backlog* se compone del Objetivo del *Sprint* (por qué), el conjunto de elementos del *Product Backlog* seleccionados para el *Sprint* (qué), así como un plan de acción para entregar el *Increment* (cómo).

El *Sprint Backlog* es un plan realizado por y para los *Developers*. Es una imagen muy visible y en tiempo real del trabajo que los *Developers* planean realizar durante el *Sprint* para lograr el Objetivo del *Sprint*. En consecuencia, el *Sprint Backlog* se actualiza a lo largo del *Sprint* a medida que se aprende más. Debe tener suficientes detalles para que puedan inspeccionar su progreso en la *Daily Scrum*.

### **Compromiso: Objetivo del Sprint**

El Objetivo del *Sprint* es el único propósito del *Sprint*. Si bien el Objetivo del *Sprint* es un compromiso de los *Developers*, proporciona flexibilidad en términos del trabajo exacto necesario para lograrlo. El Objetivo del *Sprint* también crea coherencia y enfoque, lo que alienta al *Scrum Team* a trabajar en conjunto en lugar de en iniciativas separadas.

El Objetivo del *Sprint* se crea durante el evento *Sprint Planning* y se agrega al *Sprint Backlog*. Mientras los *Developers* trabajan durante el *Sprint*, tienen en mente el Objetivo del *Sprint*. Si el trabajo resulta ser diferente de lo que esperaban, colaboran con el *Product Owner* para negociar el alcance del *Sprint Backlog* dentro del *Sprint* sin afectar el Objetivo del *Sprint*.

### **Increment**

Un *Increment* es un peldaño concreto hacia el Objetivo del Producto. Cada *Increment* se suma a todos los *Increments* anteriores y se verifica minuciosamente, lo que garantiza que todos los *Increments* funcionen juntos. Para proporcionar valor, el *Increment* debe ser utilizable.

Se pueden crear múltiples *Increments* dentro de un *Sprint*. La suma de los *Increments* se presenta en la *Sprint Review* apoyando así el empirismo. Sin embargo, se puede entregar un *Increment* a los interesados antes del final del *Sprint*. La *Sprint Review* nunca debe considerarse una puerta para liberar valor.

El trabajo no puede considerarse parte de un *Increment* a menos que cumpla con la Definición de Terminado.

### **Compromiso: Definición de Terminado**

La Definición de Terminado es una descripción formal del estado del *Increment* cuando cumple con las medidas de calidad requeridas para el producto.

En el momento en que un elemento del *Product Backlog* cumple con la Definición de Terminado, nace un *Increment*.

La Definición de Terminado crea transparencia al brindar a todos un entendimiento compartido de qué trabajo se completó como parte del *Increment*. Si un elemento del *Product Backlog* no cumple con la Definición de Terminado, no se puede publicar ni presentar en la *Sprint Review*. En su lugar, vuelve al *Product Backlog* para su consideración futura.

Si la Definición de Terminado para un *Increment* es parte de los estándares de la organización, todos los *Scrum Teams* deben seguirla como mínimo. Si no es un estándar organizacional, el *Scrum Team* debe crear una Definición de Terminado apropiada para el producto.

Los *Developers* deben adherirse a la Definición de Terminado. Si hay varios *Scrum Teams* trabajando juntos en un producto, deben definir y cumplir mutuamente la misma Definición de Terminado.

Dor determina si algo entra a la planning al sprint backlog. Dod es un criterio que determina si esta característica va a entrar al producto que yo le voy a mostrar al po para que me acepte, después de la aceptación se determina si va a producción o no.

El dor de base tiene que cumplir con el invest model, y a partir de ahí para arriba lo que el equipo decida agregar.

El dod tiene su checklist también. Los criterios están relacionados con la calidad del producto y el nivel de exigencia esta directamente relacionado con lo que la organización define, con la característica del producto que se construye. Hay toda una serie de criterios para ver que tan estricto va a ser. Hay cosas que no deberían ser negociables. Lo que se pone ahí hay que cumplirlo. No poner cosas por las dudas. Si no se cumple no puede entrar en la review.



Time box. Todas las actividades tienen que tener duración fija. El time box dice que la duración del sprint es fija y la acuerda el equipo pero tiene un máximo de 30 días, de preferencia menos. Todas las ceremonias tienen time box. La daily 30 minutos y todos tienen que hablar y asistir. Hay time box en todas. Esta la recomendación de la duración de cada una de las ceremonias.

El tiempo es complejo de administrar porque no lo pueden controlar, entonces hay que gestionarse para aprovechar mejor el tiempo. Es difícil porque hay mucho distractor, mucha fuente que interrumpe la concentración, mucha necesidad de inmediatez. Entonces un poco la idea de ayudar a organizarse forzadamente con el tiempo es el time box.

No discutir un sprint entero las estimaciones de las user. El concepto de time box nos obliga a ser respetuosos con nuestro tiempo y el de los demás. El agilismo no es legalizar el quilombo.

Si tenemos la cultura arraigada la práctica es más fácil de sostener. Las mejoras de proceso hay que sostenerlas hasta que se hace el cambio cultural. Una de las restricciones de scrum es el time box. Se terminó el tiempo y no llegamos con todo, entregamos menos. Restringe la manera que vamos a trabajar. A partir de ese time box uno va a incorporar lo que dicen las ceremonias.

Iterativo incremental de duración fija y ahí es donde engancha el time box

Timeboxing es aprender a negociar en otros términos. Que la variable de negociación no sea el tiempo. La variable es el alcance. Aprender a hacer foco. Cuando estamos hablando del compromiso no solo te ayuda con el foco, con lo de eliminar el yagni, sino también el hecho de entender que si no nos alcanza el tiempo en vez de mover la fecha de entrega estamos entregando menos. Estamos cambiando la variable de negociación. A su vez como beneficio colateral nos volvemos más confiables, el equipo tiene un ritmo de trabajo sostenible, me ayuda a generar confianza, el po sabe que en tal determinado momento va a

recibir una entrega aunque no sea todo lo que habíamos dicho, y eso desde el lado del cliente es muy bien visto. No es un tema menor y todas las ceremonias son timebox.

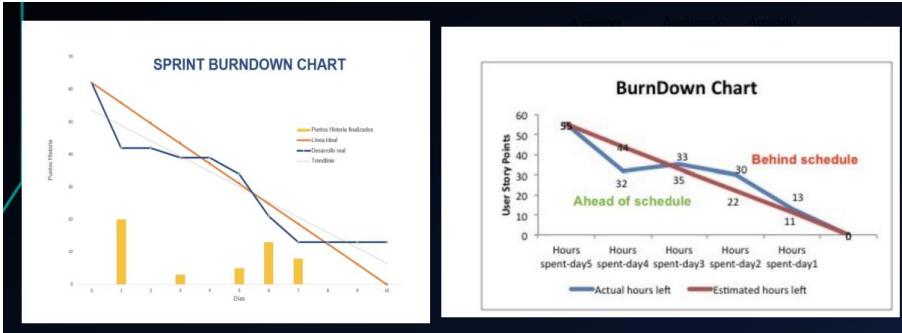
| Story                    | To Do   | In Process                                      | To Verify        | Done  |
|--------------------------|---|---|------------------|---|
| As a user, I... 8 points | Code the... 9<br>Code the... 2<br>Test the... 8 | Test the... 8<br>Code the... 8<br>Code the... 4 | Test the... SC 6 | Code the...<br>Test the... SC 6<br>Test the... SC 6<br>Test the... SC 6<br>Test the... SC 6 |
| As a user, I... 5 points | Code the... 8<br>Code the... 4                  | Code the... DC 8                                |                  | Test the... SC 6<br>Test the... SC 6<br>Test the... SC 6                                    |

La herramienta fundamental que elige scrum para visualizar el trabajo es el tablero o board. La característica de estos tableros es que duran lo que dura el sprint. Se configuran inicialmente en el sprint planning y luego al final se borran para finalizar el sprint para el siguiente. Tiene 3 columnas el To do, del producto backlog aparecen las users estimadas en story points y ese es el to do, las cosas que tengo que hacer. Después el doing, ahí están las asignaciones de trabajo, yo voy a buscar a la columna de to do lo que me comprometo a hacer, paso la user de la columna de to do al doing y le pongo mis iniciales. La hacer el propio developer, es un compromiso que asume.

Entra al to cuando cumple el DoR, se puede mover del doing al done cuando cumple el DoD. En la daily se actualiza el tablero. En el to do tenes todo lo que tenes que hacer en el sprint. En el doing tiene una sola cosa a tu nombre, y cuando la terminas vas a otra, no hacer multitasking.

Esa es la configuración básica. Solo se puede hacer si tenes developers en condiciones de agarrar una user, empezarla y terminarla. Sino usas configuraciones alternativas. Story del backlog para acá, en la primera parte de la planning. El equipo descompone en tareas y aparece el to do, cada user tiene varias tareas. Separa todas las tareas que tenes que hacer en esa user para cumplir con el DoD y las estimas en horas ideales. De ahí ya se puden asignar a personas, las tareas. Pasa a in process. Después a in verify para ver si ya podes juntar las tareas y llevarlas al done. Done Done, mueve la user de story al final cuando la user esta terminada. No hay una sola configuración del tablero, depende de lo que el equipo considere. La granularidad del tablero tiene que ser fina. Si es grande no tenes margen de maniobra. Users grandes hay riesgos de no terminarla en el sprint y además no tenes margen de monitoreo en el tablero.

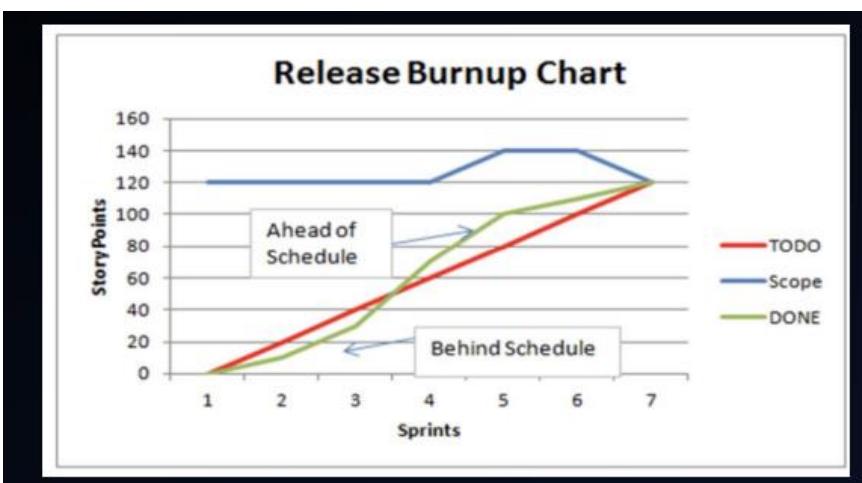
La descomposición en tareas de la user se hace en la planning. Un primer momento en que esta el po que es el del que y otro que ya no se requiere que este que es el como.



Burndown chart sirve para visualización, del trabajo. Eje y, situación ideal es que vayan puntos de historia y en x los días del sprint. Idealmente el grafico se actualiza en la daiyi en función del avance que se reporta.

El problema de la gestión de horas es que gestiona trabajo, esfuerzo. Sumar tiempo u horas trabajados no es una indicación de que hiciste el trabajo. Si ves que no baja la curva se termina cancelando el sprint

Los charts se borran al final de cada sprint. Llego el ultimo día, veo que termine y que quedo pendiente, calculo la velocidad a partir de esto (cuantos puntos de historia sumados de las user que el po acepto) y esa velocidad se guarda como una variable regferencial al momento de la planificación del sprint siguiente. Tal velocidad en un sprint no garantiza que se va a tener en el sprint siguiente, pero sirve de referencia. La experiencia es eso. Esa referencia le sirve al equipo nomas, la experiencia no se transfiere. Scrum plantea que la experiencia es valida para la gente que esta trabajanod en este proyecto, en este contexto.



El otro grafico es el burnup. Es permanente. Me hace el seguimiento de como voy trabajando en un producto a lo largo del trabajo de los sprint. Es al revés. Es una curva que

sube porque vamos sumando sp en los sucesivos sprints. Es fijo. Muestra cuanto trabajo voy terminando.



Ningún producto de soft se puede terminar en un sprint. Hay niveles de planificación diferente. El nivel mas pequeño es la daily. Ahí sincronizamos y vamos viendo la necesidad de hacer cambios ajustes o adaptaciones en la diaria para poder cumplir el compromiso que nosotros establecimos en el sprint. El siguiente nivel es el sprint planning, planificación a nivel de iteración, yo planifico un sprint (iteración en scrum). Por arriba de eso esta la planificación de reléase. El reléase es una versión de producto que tiene un conjunto de características incluidas ahí que se libera, se pone en producción. La planificación del reléase implicaría definir cuantos sprint voy a necesitar yo para poder entregar esta cantidad de características del producto que defini que integran el reléase. El concepto de planificación de reléase en términos concretos implica esa definición. Por encima esta la planificación de producto, es la sumatoria de los releases que voy a ir generando a lo largo del tiempo. La planificación del producto implica decisiones de que funcional y no funcional quiero ofrecer en el producto, y en que reléase pienso ir entregando estas características, es una planificación mas larga en horizonte de tiempo. Planificación de portfolio es cuando tengo muchos productos, cuando uno hace planificación a nivel de portfolio, se prioriza a que recurso le voy a dar mas productos, a cual menos, cual voy a descontinuar. Los horizontes de planificación en estos casos van cambiando. Mientras mas alto es el nivel de planificación, el horizonte es a mas largo plazo, el foco es mas amplio y quienes toman decisiones también es distinto. A partir del reléase ya hay un equipo que interviene

El backlog de portfolio es la sumatoria de todos los productos que la empresa tiene. La prioridad de desarrollo la da el orden del backlog. Se recomienda estimar en tales de remera en ese nivel

En el producto backlog hay mas una tendencia de estimar por story points. Depende mucho la decisión de cada equipo, pero mientras mejor estén estimados los ítems del pb es mas fácil la priorización, a veces una decisión de priorización esta basada en los sp y puede cambiar una decisión en función de los puntos de historia que requieran

El sprint backlog van las user con el dor estimadas en sp. Después el equipo decide si dividirla en tareas o no. Las tareas se estiman en horas ideales. Las users en punto. Es decisión de cada equipo si dividir o no en tarea.

Planificación del producto en términos de reléase. Cuantos releases voy a necesitar, cuantos releasess al año voy a entregar de este producto. Cuanto es lo que quiero hacer, que presupuesto tengo y que fechas estoy manejando. Yo tengo una visión de producto, esa visión me lleva a configurar el producto backlog. El pb nunca esta terminado, no hace falta que este 100% definido. Tiene que tener la cantidad suficiente de reqs para que pueda comenzar a andar. No es la ers. Las características del pb para meter en el sprint backlog desaparecen del pb si se terminaron, no vuelven a ingresar. No es un contenedor de todas las características de producto porque nunca va a tener todas las car de producto, porque algunas las termine y no están mas, o algunas no se me ocurrieron todavía.

Tengo el pb y tengo una definición de reléase, estas caracteristicas que quiero liberar al mercado. Busco saber cuantos sprint voy a necesitar para alcanzar este reléase. Es planificación de producto, cuantos releases tengo que sacar.

Los agiles si planifican, están revisando su planificación constantemente.

La salida de planificación de reléase. Que rango de características va a entrar en cada sprint, cuantos necesito para terminar este reléase que son essta bolsa de características que acorde que necesito, sobre una base de condiciones determinadas. Definir cuanto va a durar el sprint, la capacidad estimada del equipo, los objetivos de cada sprint, y en función de eso vos ha<<ces la planificación, sobre la base de cambia alguna situación, entonces el plan se va a tener que cambiar.

La empresa decide como quiere armar su reléase. En algunos casos es igual al sprint, termina el sprint y liberan a producción. Cd cada característica que terminan la ponen a producción. Algunos liberan una reléase al año. La cadencia la define cada organización, no son decisiones del equipo.

## ASEGURAMIENTO DE CALIDAD DE PROCESO Y PRODUCTO

El **aseguramiento de calidad de procesos y productos** se centra en la implementación de medidas preventivas para garantizar que un producto alcance la máxima **calidad** posible. Esto implica tomar acciones anticipadas y realizar actividades antes de que el producto esté terminado, con el objetivo de asegurar que la calidad esté integrada en todas las etapas del proceso. En lugar de detectar y corregir errores o defectos después de que se hayan producido, el enfoque se centra en prevenirlos desde el principio.

Que distinción hay entre hacer aseguramiento de calidad y acciones para trabajar de manera preventiva sobre la calidad, y que diferencia hay entre eso y hacer testing, que llega tarde porque el producto ya esta hecho. El espíritu del aseguramiento de calidad tiene que ver con hacer cosas antes, en trabajar como prevención para que el producto tenga embebida la mejor calidad que se pueda

La **CALIDAD** se refiere a todos los aspectos y características de un producto o servicio que permiten que este cumpla con todas las necesidades, tanto las que se expresan de manera clara y directa como las que están implícitas o no se mencionan explícitamente. La calidad implica la capacidad de un producto o servicio para satisfacer y superar las expectativas y requisitos del cliente.

Estos aspectos y características están relacionados con la capacidad del producto o servicio para cumplir con su propósito, funcionar de manera confiable, brindar eficiencia y durabilidad, y ofrecer beneficios y características adicionales que generen valor para el cliente.

Calidad es muy difícil de medir. Es subjetivo, tiene mucho que ver con las expectativas y necesidades de cada una de las personas. Tiene que ver si para mi cumple con mis necesidades y expectativas. Y la más difícil de las dos son las expectativas, porque son implícitas. Las expectativas son las cosas que espero que este producto o servicio tenga, pero no las manifiesto explícitamente. No digo quiero que el sistema sea lindo o tenga una interfaz amigable, porque lo considero obvio o por alguna razón, pero no lo manifiesto, entonces cuando no ocurre me siento disconforme. La calidad está directamente relacionada con la persona, sus circunstancias, su contexto, su edad. Además esa visión de calidad en un momento de tiempo puede cambiar después por alguna circunstancia.

**La visión de calidad que tenemos en un momento en el tiempo puede cambiar en otro, ya que la calidad es relativa a las personas.**

#### **¿QUÉ COSAS OCURREN FRECUENTEMENTE EN LOS PROYECTOS DE DESARROLLO DE SOFTWARE?**

- △ Atrasos en las entregas
- △ Costos Excedidos
- △ Falta cumplimiento de los compromisos
- △ No están claros los requerimientos
- △ El software no hace lo que tiene que hacer
- △ Trabajo fuera de hora
- △ Fenómeno del 90-90, 90% hecho 90% faltante, el líder de proyecto le pregunta a su equipo como van y cuánto falta, y su equipo le dice que bien y que falta poco.
- △ ¿Dónde está ese componente?

Estas situaciones concretas evidencian la falta de calidad o la NO calidad. La calidad no solo tiene que ver con el producto, sino también con el proyecto y con el equipo involucrado en el mismo.

El **aseguramiento de calidad de procesos** implica establecer y mantener estándares y procedimientos claros para llevar a cabo las actividades de manera consistente y eficiente. Esto incluye definir las mejores prácticas, documentar los procesos, capacitar al personal en la ejecución correcta de las tareas y establecer controles para monitorear y medir el desempeño.

Por otro lado, el **aseguramiento de calidad de productos** se centra en garantizar que los productos cumplan con los estándares de calidad establecidos. Esto implica realizar inspecciones, pruebas y evaluaciones durante el proceso de fabricación para identificar y corregir cualquier defecto o problema antes de que el producto final sea entregado al cliente. También se pueden implementar sistemas de gestión de calidad, como ISO 9001, para asegurar que se cumplan los requisitos y se sigan las mejores prácticas.

La calidad del software muchas veces se ve reducida por:

- Demoras en los tiempos de entrega
- Costos excesivos
- Falta de cumplimiento de los compromisos
- El software no hace lo que se pide

En contraposición, un software de calidad ofrece:

- Cumplimiento de las expectativas del cliente
- Cumplimiento de las expectativas del usuario
- Cumplimiento de las necesidades de la gerencia.
- Cumplimiento de las necesidades del equipo de desarrollo y de mantenimiento

### **PRINCIPIOS DE ASEGURAMIENTO DE CALIDAD**

**Sustentan la necesidad de asegurar la calidad**

Hay principios básicos que sustentan la necesidad del aseguramiento de calidad. Acá lo importante es si uno quiere o no hacer un producto que tenga calidad. Los principios agiles hablan de calidad todo el tiempo y de que no es negociable. La calidad debe ser algo que se concibe y se decide desde el momento cero. No es que hago las cosas mal y después al final le agrego la calidad, o le agrego la calidad con el testing, el testing no agrega nada, visibiliza que tan bien o mal este hecho el producto. No vas a conseguir que el producto tenga calidad con el testing, vas a conseguir que el producto tenga calidad haciendo el producto con la calidad

- La calidad no se inyecta ni se compra:** La calidad no puede ser agregada o comprada después de la construcción del producto o servicio, debe estar incorporada/embebida en el producto o proceso desde el momento 0 que comienza su construcción.
- La calidad conlleva un esfuerzo de todos:** Asegurar la calidad no es solo responsabilidad de un departamento o individuo, requiere el compromiso y esfuerzo de todos los miembros del equipo.
- Las personas son clave para lograr la calidad:** La capacitación del equipo en calidad es fundamental. El éxito en el desarrollo de software depende en gran medida de las habilidades y conocimientos del personal involucrado.
- Se necesita un sponsor a nivel gerencial:** Es importante contar con el respaldo y liderazgo de un patrocinador o sponsor a nivel gerencial que considere el desarrollo de un producto de calidad como algo fundamental y lo respalde con *recursos y apoyo*.
- Se debe liderar con el ejemplo:** Los líderes deben ser ejemplos de compromiso y calidad, demostrando prácticas y comportamientos que fomenten la cultura de aseguramiento de calidad en toda la organización.
- La calidad se mide mediante el testing:** Las pruebas son una herramienta fundamental para medir y controlar la calidad del software. Sin pruebas adecuadas, no se puede tener un control efectivo sobre la calidad. **No se puede controlar lo que no se mide.**
- Simplicidad, empezar por lo básico:** En lugar de complicar los procesos, se debe buscar la simplicidad y comenzar por los fundamentos básicos del aseguramiento de calidad.
- El aseguramiento de calidad debe planificarse:** Es necesario contar con un plan de aseguramiento de calidad que establezca los objetivos, las actividades y los recursos necesarios para garantizar la calidad del producto o servicio.
- El aumento de las pruebas NO aumenta la calidad automáticamente:** Simplemente aumentar la cantidad de pruebas realizadas no garantiza automáticamente una mayor calidad. Se deben aplicar técnicas y enfoques adecuados para obtener resultados efectivos.
- Debe ser razonable para mi negocio:** El aseguramiento de calidad debe adaptarse a las necesidades y características específicas de cada negocio, siendo realista y viable en términos de recursos y objetivos alcanzables.

## ¿CALIDAD PARA QUIÉN?

Cuando hablamos de calidad, ya sea la calidad del proceso para construir el producto, la calidad del producto en sí misma, la calidad que se espera que el producto tenga, debemos saber **desde qué perspectiva**.

Tenemos que **integrar todas las perspectivas** (visiones), desde cuáles son las expectativas de calidad que pretende el usuario, hasta las del equipo de desarrollo (que muy probablemente no sean las mismas). El usuario probablemente quiera una buena interfaz, que sea fácil de usar, que sea rápido, mientras que el equipo de desarrollo el producto va a tener calidad si es fácil de mantener y si tiene un nivel de cohesión y acoplamiento adecuado.



La **visión trascendental** hace referencia a responder la pregunta de **¿para qué quiero construir software?** Tiene que ver con lograr cosas más allá de lo que uno se imagina que puede hacer con el software.

Para mí tiene calidad si es fácil de mantener, pero eso al usuario no le importa, quiere que sea rápido, que sea fácil de usar, etc.

| CALIDAD PROGRAMADA   | CALIDAD NECESARIA   | CALIDAD LOGRADA  |
|--|---|--|
| Se refiere a las expectativas y estándares de calidad que se establecen para el producto durante su planificación y diseño. Es la calidad que se busca alcanzar y que se establece como objetivo a lo largo del proceso de desarrollo. | Es el nivel mínimo de calidad requerido para que el producto sea considerado correcto y pueda satisfacer los requerimientos y expectativas del usuario. Corresponde al conjunto de características y funcionalidades esenciales que el producto debe cumplir para cumplir con su propósito. | Se refiere a la calidad real que se ha alcanzado en el producto final una vez completado el proceso de desarrollo y las pruebas correspondientes. Es el resultado concreto y medible de las actividades de desarrollo, aseguramiento y control de calidad. |

Al lograr que estas tres perspectivas de calidad coincidan disminuimos efectivamente el riesgo de desperdicio y de insatisfacción de los clientes.

## Calidad en el Software

Para desarrollar software de manera efectiva, es fundamental contar con una **estructura o guía** que oriente al equipo en la construcción del producto deseado. Esta estructura se conoce como **proceso**.

Al crear o adaptar un proceso en una organización, ya sea basado en estándares definidos o en enfoques empíricos, es recomendable **basarse en Modelos de Referencia** (Modelos para crearlos o Modelos de calidad). Estos modelos proporcionan **lineamientos y mejores prácticas** para la creación y gestión de procesos, así como para **garantizar la calidad del software**.

Una vez que se ha establecido un proceso en la organización, incorporando los modelos de referencia pertinentes, es importante buscar una mejora continua del mismo. Para lograr esto, se pueden adoptar **Modelos de Mejora de Procesos**, los cuales brindan un marco de trabajo para identificar áreas de mejora y establecer proyectos que impulsen dichas mejoras.

Para hacer software o cualquier cosa, la gente necesita una guía, una idea de para llegar de acá hasta allá necesito hacer estas cosas, y esas cosas hechas así me van a permitir ami lograr el objetivo. En el ámbito del soft es necesario un proceso. Que elijas un empírico no significa que no uses proceso, sino que la concepción es distinta. Proceso tenemos que tener siempre. Ninguna empresa toma un libro y dice el proceso va a ser tal cual como dice el libro. Primero poque no hay ninguno completo, entonces uno se basa en modelos para tomar de referencia para poder definir un proceso para una organizacipn. Una vez que esos modelos están incorporados y creamos nuestro proceso, si queremos funcionar en un ciclo de mejora continua, motivación que existe independiente de los procesos empíricos y definidos, la mejora continua esta fuertemente enfocada a los procesos (y también al producto), entonces para eso aparecen modelos que nos ayudan a mejorar los procesos. Nos ayudan a definir proyectos para mejorar los procesos. Aparecen modelos como el IDEAL. Kanban es un framework para mejorar procesos. IDEAL esta en la línea de los procesos definidos. El propósito es dar un marco de referencia a los equipos o a las orgs que quieren mejorar sus procesos. Los modelos de evaluación intentan ver el grado de adherencia del proceso al modelo que tomaron de referencia, se usan para certificaciones, presiones comerciales, evaluación en calidad, ley de soft, etc.

Además, es posible formalizar y certificar el cumplimiento de un proceso mediante **Modelos de Evaluación**. Estos modelos se utilizan para medir y evaluar el grado de adherencia del proceso a los estándares y modelos de referencia establecidos.

**El proceso se materializa en proyectos, los cuales son la unidad de trabajo que da vida a dicho proceso.** Al iniciar la ejecución de un proyecto, es esencial incorporar actividades de aseguramiento de calidad, como revisiones técnicas y auditorías, para evaluar la aplicación de los modelos seleccionados.

**El proyecto a su vez es el ámbito donde uno trabaja para obtener el producto,** la versión de producto que yo someto a evaluación se va generando en el contexto de un proyecto. Para el contexto del producto, existen técnicas y herramientas tanto para el aseguramiento de calidad, como la revisión de pares o las auditorías (De configuración, auditorías físicas y funcionales), como para el control de calidad, como es el Testing.

**Cuando yo hago actividades de aseguramiento de calidad tanto de proceso como de producto, se hace en el contexto de un proyecto en específico.**

Es importante distinguir el **Aseguramiento de calidad de Proceso** y el **Aseguramiento de calidad de Producto**, por eso acá una breve comparación:

El aseguramiento de calidad de proceso se enfoca en el proceso de desarrollo de software y busca mejorar la forma en que se trabaja, mientras que el aseguramiento de calidad de producto se centra en evaluar y garantizar la calidad del producto de software resultante. Ambos aspectos son fundamentales para lograr un desarrollo de software exitoso y de alta calidad.

|            | ASEGURAMIENTO DE CALIDAD DE PROCESO  | ASEGURAMIENTO DE CALIDAD DE PRODUCTO   |
|------------|--|--|
| FOCO       | Se concentra en evaluar y mejorar el proceso de desarrollo de software en sí mismo, es decir, la forma en que las actividades se llevan a cabo.  | Se enfoca en evaluar y garantizar la calidad del producto de software resultante, es decir, los artefactos y entregables generados durante el proceso de desarrollo.   |
| OBJETIVO   | Busca establecer prácticas efectivas, eficientes y consistentes para el desarrollo de software, con el fin de optimizar la productividad y minimizar los errores o problemas en el proceso.              | Busca asegurar que el producto de software cumpla con los requisitos establecidos, sea confiable, funcione correctamente y sea adecuado para su uso previsto.  |
| ACT. CLAVE | Involucra definir estándares y buenas prácticas, realizar revisiones técnicas y auditorías de proceso, medir y monitorear indicadores de calidad del proceso, y gestionar la configuración del software. | Incluye realizar pruebas de software (como pruebas funcionales, de rendimiento o de seguridad), realizar revisiones de código, realizar auditorías de producto, verificar el cumplimiento de estándares de calidad y realizar validaciones con los usuarios o stakeholders del producto. |

## CALIDAD EN PROCESOS DEFINIDOS vs CALIDAD EN PROCESOS EMPIRICOS

El problema de visión entre los procesos definidos y empíricos se debe a diferentes enfoques sobre la relación entre la calidad del proceso y la calidad del producto.

En el caso de los procesos definidos, se argumenta que si el proceso en sí mismo es de alta calidad y se respeta en el contexto del proyecto, entonces el producto resultante también será de alta calidad. Esta perspectiva se basa en la premisa de que un proceso sólido y bien estructurado conducirá a resultados consistentemente buenos.

*"El proceso tiene que tener calidad y si el proceso tiene calidad y la gente en el contexto del proyecto lo respeta, como consecuencia el producto que se obtenga también va a tener calidad"*

Sin embargo, es importante tener en cuenta que esta premisa no debe tomarse de manera literal. Aunque un proceso pueda ser de alta calidad, eso no garantiza automáticamente que las personas que trabajan en él cumplirán con todas las tareas y responsabilidades adecuadamente. Así como tener un buen auto no garantiza que alguien sea un buen conductor y aproveche todas las funciones del vehículo, o tener una habitación llena de libros no implica necesariamente que alguien sea un lector ávido.

Por otro lado, aquellos que trabajan con enfoques empíricos enfatizan que la calidad del producto depende de las personas que participan en el desarrollo. En esta perspectiva, se sostiene que si el equipo realiza adecuadamente sus tareas y cumple con sus responsabilidades, el producto final tendrá calidad. Se pone énfasis en la colaboración, las habilidades y el compromiso del equipo de desarrollo.

*"La calidad del producto depende de las personas que participan, si el equipo hace lo que tiene que hacer el producto va a tener calidad"*

## Calidad en el Producto

**La calidad del producto** No se puede sistematizar, no hay modelos en la industria generalizados para evaluar calidad de producto que se puedan aplicar como una plantilla a todos los productos igualmente.

Su **aseguramiento** se hace mediante las **Revisões Técnicas** (Revisões de pares) y **Auditórias**, mientras que su **control** se hace mediante el **Testing**.

No hay modelos para certificar calidad de producto, porque es muy complejo porque cada producto tiene sus características, y la calidad del producto está directamente relacionada con quienes van a usar el producto.

## MODELOS DE CALIDAD DE PRODUCTO

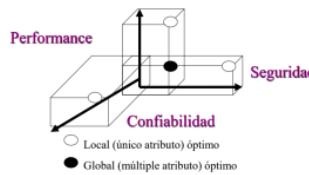
Comentado [9]: No hace falta saberlo

Los modelos de calidad de producto son marcos o estructuras conceptuales que se utilizan para evaluar y medir la calidad de los productos de software. Estos modelos proporcionan un conjunto de características, criterios y métricas que se utilizan para determinar la calidad de un producto de software.

### 1. Modelo de Barbacci

Este modelo ofrece una perspectiva amplia y completa de la calidad del producto de software, centrándose en varios aspectos clave.

La calidad del producto se mide mediante su confiabilidad, performance y seguridad. Debemos trabajar para lograr un equilibrio entre estas tres cosas.



## 2. Modelo de McCall

La calidad depende de tres factores determinantes que causalmente tiene que ver con las visiones de calidad mencionadas anteriormente:

**Revisión del producto:** Es la capacidad de verificación del producto. Es la facilidad de mantenimiento, flexibilidad y prueba.

**Operación del producto:** Es la calidad del producto en uso, lo que más le importa al usuario final. Corrección, fiabilidad y usabilidad.

**Transición del producto:** Es la facilidad con la que el producto puede expandirse y crecer.



NUNCA se puede hacer esa comparación solo contra el modelo, primero se debe definir qué es lo que se supone que el cliente quiere como características de calidad del producto, en términos de requerimientos. La evaluación de calidad sobre un producto se hace en base a que tanto se acerca a esos requerimientos más que lo que se acerca a cumplir con los modelos teóricos.

## Calidad en el Proceso

El proceso es el único factor controlable para mejorar la calidad del Software y el rendimiento como organización.



El producto, las personas, la tecnología, las características del cliente, las condiciones de negocio y el entorno de desarrollo son incontrolables.

A raíz de un proceso que adopte la calidad, se puede instanciar un proyecto que tenga la calidad como factor embebido, lo que lleva a un **Software de calidad**.

El Aseguramiento de calidad de Software implica insertar, en cada una de las actividades, acciones que detecten lo más temprano posible oportunidades de mejora, tanto sobre el producto como sobre el proceso.

Implica la **definición de estándares y procesos de calidad apropiados**, y el aseguramiento de que los mismos sean respetados.

Debe ayudar a desarrollar una **cultura de calidad**, donde la calidad es vista como una responsabilidad de todos y cada uno.

## DEFINICIÓN DE UN PROCESO DE SOFTWARE

**Proceso:** Secuencia de pasos ejecutados para un propósito dado (IEEE)

**Proceso de Software:** Un conjunto de actividades, métodos, prácticas y transformaciones que la gente usa para desarrollar o mantener software y sus productos asociados (Sw-CMM)

El proceso no es solamente las tareas escritas, sino que se define como una mesa de 3 patas, donde sí es cierto que debe haber lineamientos de cómo vamos a trabajar, pero debe haber personas con habilidades con entrenamiento y motivación, y el mejor soporte de herramientas automatizadas y equipamiento.

### ¿Cómo es un proceso para Construir Software?



La **ingeniería** se le llama a la etapa de requerimientos, análisis y diseño. **Implementación**, es la codificación, el testing y el despliegue. Ahora incorporamos en un proceso de desarrollo de software disciplinas de gestión y de soporte estas son:

- Planificación y Seguimiento de Proyectos
- Administración de Configuraciones
- Aseguramiento de la Calidad

En un proceso de soft hay mas que disciplinas técnicas. Hay disciplinas de gestión y de soporte

Son disciplinas transversales. Empezamos a trabajar y empiezan a funcionar esas disciplinas. Están en background, segundo plano. Mientras hacemos soft hacemos calidad.

### ASEGURAMIENTO DE CALIDAD DE SOFTWARE

**“Lo que no está controlado no está hecho”**

Concerniente con asegurar que se alcancen los niveles requeridos de calidad para el producto de software. Implica la definición de estándares y procesos de calidad apropiados y asegurar que los mismos sean respetados. Debería ayudar a desarrollar una “cultura de calidad” donde la calidad es vista como una responsabilidad de todos y cada uno.

**Implica insertar en cada una de las actividades acciones tendientes a detectar lo más tempranamente posible oportunidades de mejora sobre el producto y sobre el proceso.**

Aseguramiento de calidad de soft, es insertar en cada una de las actividades acciones tendientes a detectar lo mas tempranamente posible oportunidades de mejora sobre el producto y sobre el proceso. Tiene que estar en la voluntad y la cultura de todos. Para materializarlo cuando una org quiere un grupo de aseguramiento de calidad, hay algunas consideraciones, por ejemplo que no debe reportar la gente que hace calidad al mismo gerente que hace los proyectos (porque le quita independencia y objetividad porque es el jefe). El grupo de aseguramiento de calidad tiene que tener un reporte independiente al reporte de los proyectos, y además ese reporte tiene que ser lo mas cerca posible a depender directamente de la gerencia del primer nivel, porque esa es la manera que garantizamos independencia. La gente de calidad tiene en un primer momento definir los

estándares y los procesos y los modelos contra los cuales va a hacer las comparaciones. Tanto las revisiones como las auditorías son actividades de comparación y tengo que tener contra qué comparar, sino no puedo hacer alguna actividad de calidad, por ejemplo la revisión técnica de la arq tengo que tener los reqs para ver si los cumple, so hago una revisión de que tanto un proyecto cumple un proceso el proceso tiene que estar definido. Despues tiene que planificar la calidad, a este proyecto le voy a hacer estas revisiones en este momento, estas auditorias en este otro. El de testing suele ser un plan aparte, pero se lo referencia desde el de calidad. Y despues ejecutar esas actividades para ver en qué situación está el proyecto.

Grupo de calidad es el rol, puede ser una persona, un área, etc, depende de la necesidad de una empresa.

Aseguramiento de calidad es insertarse en el proceso de desarrollo mientras el soft se está construyendo. No esperar a que el producto esté terminado para ponerle cal

#### **Reporte del grupo de aseguramiento de calidad (GAC)**

Para materializarlo en forma concreta, cuando una organización quiere un grupo de aseguramiento de la calidad hay que tener en cuenta las siguientes consideraciones.

- No se debe reportar al líder de proyecto, como forma de mantener la independencia y la objetividad. El grupo de aseguramiento de calidad tiene que tener un **reporte independiente** al reporte de los proyectos y tiene que ser **lo más cerca posible a depender directamente del nivel más alto de la organización**.
- No debe existir más de un nivel de gerencia entre la Dirección y el GAC, **el reporte debe ser directo a la gerencia de dirección**.
- El GAC debe reportar a alguien realmente interesado en la calidad del Software.

#### **Actividades de administración de Calidad del Software**

**ASEGURAMIENTO DE CALIDAD:** Establecer estándares y procedimientos de calidad. Elegir contra qué estándares se van a efectuar las comparaciones. (Las revisiones técnicas y las auditorías son actividades de comparación)

**PLANIFICACIÓN DE CALIDAD:** Se seleccionan los procedimientos y estándares de calidad para un proyecto y particular, y se los modifica si fuera necesario. Ejemplo, en qué momento se van a hacer las revisiones, las auditorías.

**CONTROL DE CALIDAD:** Se ejecuta el control de calidad de acuerdo a los procedimientos y estándares de calidad elegidos. Se ejecutan las actividades definidas en la planificación para ver en qué situación está el proyecto.

**Aseguramiento de calidad es insertarse en el proceso de desarrollo de software mientras el software se está construyendo.**

### Funciones del Aseguramiento de Calidad de Software

- Prácticas de Aseguramiento de Calidad
- Desarrollo de herramientas adecuadas, técnicas, métodos y estándares que estén disponible para realizar las revisiones de Aseguramiento de Calidad.
- Evaluación de la planificación del Proyecto de Software
  - Evaluación de Requerimientos
  - Evaluación del Proceso de Diseño
  - Evaluación de las prácticas de programación
  - Evaluación del proceso de integración y prueba de software
  - Evaluación de los procesos de planificación y control de proyectos
  - Adaptación de los procedimientos de Aseguramiento de calidad para cada proyecto.

## PROCESOS BASADOS EN CALIDAD

El trabajo de las personas encargadas de la calidad en el desarrollo de software implica definir y establecer los procesos que se seguirán para garantizar la calidad del producto final. Estos procesos incluyen actividades como la planificación, el diseño, la implementación, las pruebas y la entrega del software.



Una vez que se han establecido los procesos, es importante evaluar continuamente su efectividad y calidad. Esto se logra a través de la evaluación del producto de software resultante. Se analizan los artefactos y entregables generados para determinar si cumplen con los estándares de calidad y si satisfacen los requisitos establecidos.

En función de los resultados de la evaluación del producto, se toman decisiones sobre la mejora del proceso. Si se identifican deficiencias o problemas en el producto, se realiza una revisión exhaustiva del proceso correspondiente y se implementan mejoras para corregir los errores y evitar que se repitan en el futuro.

Por otro lado, si el producto de software cumple con los estándares de calidad y se considera exitoso, el proceso utilizado se estandariza. Esto implica documentar y comunicar de manera efectiva los pasos, las actividades y las buenas prácticas empleadas durante el desarrollo del producto. Estos procesos estandarizados se distribuyen al resto de la organización para que puedan ser seguidos y aplicados de manera consistente en proyectos futuros.

### MODELOS DE MEJORA (Modelos para mejorarlos)

Los modelos de mejora son recomendaciones o estándares para encarar un proyecto de mejora de un proceso. El propósito de un modelo de mejora es tomar el proceso de una organización y elaborar un proyecto, cuyo resultado va a ser un proceso mejorado.

Sirven para armar un proyecto que nos va a permitir a nosotros tener un nuevo proceso mejorado para aplicar en los nuevos proyectos de desarrollo de la organización.

- △ SPICE.
- △ IDEAL (INITIATING, DIAGNOSING, ESTABLISHING, ACTING, LEARNING)

## IDEAL

Modelo de mejora que nos sirve para definir en una organización, un proyecto que ayude a mejorar el proceso que esa empresa tiene.

**INITIATING (Inicio):** Se busca un sponsorero en la organización. Esto es importante, puesto que los proyectos de mejoras de proceso no suelen ser tomados como prioridad a menudo en las organizaciones.



es muy importante porque este tipo de proyectos nunca son críticos, siempre hay otra cosa mas urgente, entonces si no tengo un aval para ejecutar este proyecto inclusive en niveles altos de la org, no solo por el financiamiento sino por garantizar que la gente realmente se ponga las pilas y le dedique tiempo, sin ese sponsorero no terminan bien porque siempre hay algún otro proyecto mas urgente y consume los recursos.

**DIAGNOSING (Diagnóstico):** Se realiza un diagnóstico para determinar cuál es el punto de partida del proyecto. Se realiza un **Análisis de brecha:** se analiza en donde esta parada la organización y a qué objetivo quiere apuntar.

Análisis de brecha, gap análisis. Yo quiero llegar a algún lado a tener un proceso que sea compatible con el nivel 2 de CMMI, entonces tenes que hacer todo esto con tu proceso para tener tu proceso mejorado

**ESTABLISHING (Establecimiento):** Se planifica y se establece un **plan de mejora** detallado. Se definen los objetivos, se identifican las actividades necesarias y se asignan los recursos necesarios.

Son planes de acción, una vez definidos y aprobados empieza la etapa de definir los procesos, escribir los roles, las actividades, todo lo que va a regular el proceso, y a partir de ahí que prácticas quiero que mi gente haga, eso es libre lo define cada org como quiere, el como la org va a hacer análisis, reqs etc lo define la org, las herramientas que quiere usar, que es lo que va automatizar, si va a usar prácticas continuas, si va a usar users, eso lo define la org

**ACTING (Actuar):** Se ejecuta el **plan de acción** y las estrategias definidas. Probamos el proceso definido en algún proyecto (pruebas pilotos).

Este modelo plantea planear y ejecutar pilotos, que ese proceso se ponga a prueba en algunos proyectos, no en toda la org al mismo tiempo, para poder tener una supervisión mas fina de como esta funcionando el proceso

**LEARNING (Aprender):** Si el resultado es positivo, lo extrapolamos a toda la organización. **Se documentan los resultados** del plan de acción y se vuelve a ejecutar el modelo con las lecciones aprendidas y para identificar oportunidades de mejora.

**Es un proceso cíclico ya que está orientado a la mejora continua**

Cuando hacemos los **análisis de brecha** entran los Modelos de Calidad

### MODELOS DE CALIDAD (Modelos para crearlos)

## CMMI (Capability Maturity Model Integration)

Surgió específicamente para empresas, organizaciones que hacen específicamente software. Modelo que se usa de referencia en base a los objetivos que quieren alcanzarse. El CMMI dice el **qué** no el **cómo**, es un modelo **descriptivo** no prescriptivo.

El objetivo principal de CMMI es proporcionar una guía para mejorar la capacidad y madurez de los procesos organizacionales. El modelo se basa en buenas prácticas recopiladas de la industria y está diseñado para ayudar a las organizaciones a alcanzar niveles superiores de calidad, eficiencia y satisfacción del cliente.

CMMI se estructura en niveles de madurez y áreas de proceso. Los niveles de madurez representan etapas de mejora evolutiva en la organización, desde un nivel inicial ad hoc hasta un nivel de optimización continuo. Las áreas de proceso se enfocan en aspectos específicos del desarrollo y gestión de software, como la planificación, el monitoreo y control, la gestión de requisitos, el diseño, la implementación y la evaluación.

CMMI proporciona un conjunto de prácticas y criterios que las organizaciones pueden seguir para mejorar sus procesos. Estos criterios son evaluados mediante una evaluación formal realizada por profesionales capacitados, lo que permite obtener una medida objetiva de la madurez y capacidad de los procesos organizacionales.

Al seguir las recomendaciones de CMMI, las organizaciones pueden lograr beneficios como la reducción de defectos, la mejora de la productividad, la gestión de riesgos más efectiva, la alineación con los objetivos del negocio y la mejora en la entrega de productos y servicios.

Es importante destacar que CMMI es un modelo flexible y adaptable, que se puede personalizar según las necesidades y características de cada organización.

Tiene 3 dominios o ámbitos de mejora (Constelaciones):

**CMMI DEV:** Provee la guía para medir, monitorizar y administrar los procesos de desarrollo. Se enfoca específicamente en el desarrollo de software

**CMMI SVC:** Provee la guía para entregar servicios, internos o externos.

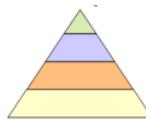
**CMMI ACQ:** Provee la guía para administrar, seleccionar y adquirir productos o servicios.

## REPRESENTACIONES CMMI

Aparecen dos formas de mejorar, evolucionar con el proceso.

### 1. Por etapas:

En esta representación, se organizan los niveles de madurez de la organización en etapas predefinidas. Cada nivel de madurez representa un conjunto de prácticas y capacidades específicas que deben alcanzarse para avanzar al siguiente nivel. Los niveles de madurez en la representación por etapas son:



- △ Nivel 1: Inicial: La organización tiene procesos ad hoc y no estándarizados.
- △ Nivel 2: Gestionado: La organización establece prácticas de gestión básicas y utiliza enfoques más disciplinados para el desarrollo de proyectos.
- △ Nivel 3: Definido: La organización establece y documenta procesos estandarizados y repetibles.
- △ Nivel 4: Cuantitativamente gestionado: La organización establece mediciones cuantitativas para el control y mejora de los procesos.
- △ Nivel 5: Optimizado: La organización enfoca en la mejora continua y la optimización de los procesos.

(Lo vemos detallado más abajo \*1)

Divide a las organizaciones en dos tipos: maduras o inmaduras

Las organizaciones **inmaduras** son las organizaciones de nivel 1.

Las **organizaciones maduras** son las organizaciones de nivel 2 a 5.

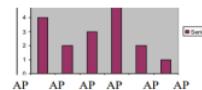
Mientras más madura es la organización, **más capacidad tiene para cumplir con los objetivos**, entonces mejoraba la calidad de sus productos y disminuía sus riesgos.

Esta representación tiene como ventaja que provee una única clasificación que facilita las comparaciones entre organizaciones. Habla de la organización, entendiendo como organización el área de la empresa que quiero evaluar, no necesariamente es toda la empresa.

La representación por etapas proporciona una estructura clara y lineal para mejorar la madurez de la organización, donde cada nivel construye sobre los logros del nivel anterior.

### 2. Continua

Esta representación lo que hace es elegir áreas de proceso dentro de las que el modelo te ofrece (son 22 en la última versión) y yo elijo cuales son los procesos quiero mejorar por separado, entonces en lugar de medir la madurez de toda la organización, se mide la capacidad de un proceso en particular.



Esta representación mide la CAPACIDAD de las distintas áreas de proceso para poder cumplir los objetivos.

#### Mide áreas individuales.

En vez de medir las MADUREZ de la organización, mide la CAPACIDAD de un proceso en particular.

Cada área de proceso tiene metas y prácticas específicas asociadas. La evaluación de la capacidad se realiza en una escala de 0 a 5, donde cada nivel de capacidad indica el grado de implementación y desempeño de las prácticas específicas de esa área.

La representación por capacidad permite a las organizaciones seleccionar y mejorar áreas de proceso específicas de acuerdo con sus necesidades y prioridades. Se puede trabajar en paralelo en diferentes áreas de proceso sin tener que seguir una secuencia fija.

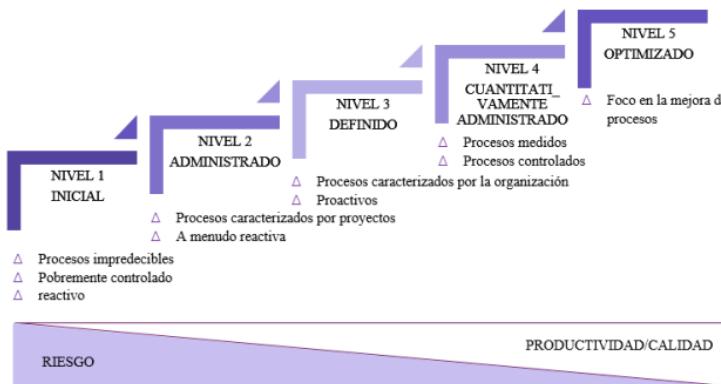
#### CMMI ¿ROLES Y GRUPOS?

Cuando CMMI se refiere a grupos se refiere a la existencia de roles que cubren ciertas tareas. Esos roles se adaptan al tamaño de la organización y a las expectativas de madurez que la organización quiere llegar.

Lo importante es que exista alguien responsable de cubrir las actividades de cada uno de los roles o grupos.

CMMI permanentemente habla de debe existir un grupo de tal cosa. Roles que cumplan ciertas tareas, que se adaptan al tamaño de la org, la cantidad de gente que tiene, las expectativas de madurez que la org quiere alcanzar. La palabra grupo suena a manada pero no, cuando dice que debe existir un grupo de ingeniería de software, es porque tiene que haber gente que haga el soft, cubrir el rol de analista, desarrollador, tester, etc. Dice tiene que haber un grupo de scm, es porque alguien tiene que asumir el rol que implica las actividades de ser gestor de configuración de proyecto, no importa si son muchas personas o una, lo que haga falta en la organización. Lo importante es que exista la responsabilidad definida y que alguien la asuma.

#### \*1 Representación por Etapas



##### 1. NIVEL 1: INICIAL

No existe visibilidad respecto del proceso. No se sabe cuándo termina, cuánto cuesta ni la calidad de lo que se obtiene.

Son organizaciones caracterizadas por “apagar incendios”. Estas organizaciones tienen actitud reactiva: buscan como atacar el problema una vez sucedió, en vez de buscar prevenirlo. Son organizaciones inmaduras.

##### 2. NIVEL 2: ADMINISTRADO

En este nivel, la organización comienza a tener controles básicos para la gestión de proyectos y procesos. Se busca una mayor estandarización y se inicia la recopilación de métricas para el seguimiento y la toma de decisiones.

Disciplinas de gestión y de soporte. Una de las cosas que se advirtió cuando surgió el SW CMM es que las empresas no gestionaban sus proyectos de ninguna manera y no había scm. Ese análisis que se hizo y es una característica de las orgs inmaduraz, tienen que ir a atender los problemas que tienen, entonces la idea del nivel 2 cuyo foco es tener un proceso administrado significa incorporar en las orgs la capacidad de gestión y de soporte

### 3. NIVEL 3: DEFINIDO

En este nivel, la organización ha establecido y documentado procesos definidos y estandarizados. Se definen roles y responsabilidades, y existe retroalimentación.

El nivel 3 es el que mas áreas de proceso tiene con 11. Se encarga de la ingeniería, practicas que tienen que ver con la calidad del producto

### 4. NIVEL 4: CUANTITATIVAMENTE ADMINISTRADO

En este nivel, la organización tiene la capacidad de cuantificar y medir su rendimiento. Se establecen métricas, y se recopilan datos para analizar el desempeño de los procesos y procedimientos.

### 5. NIVEL 5: OPTIMIZADO

En este nivel, la organización se enfoca en la mejora continua y en la optimización de sus procesos.

**Nosotros hacemos foco en el nivel 2**, que es el de administración. Cómo se puede observar son los temas que hemos visto. Son disciplinas de gestión y de soporte.

*Si alcanzo un nivel 2 de CMMI puedo decir que mi organización tiene madurez para administrar sus proyectos y que el resultado de sus proyectos va a ser un producto de software de lo que se sabe que se espera de él.*

En resumen, a todo lo anterior, si se quiere mejorar el proceso de una organización y se elige **IDEAL** como modelo de mejora y marco de referencia para implementar la mejora y queremos alcanzar un modelo de calidad **CMMI**. Cuando el proceso esté listo, definido y que haya proyectos que lo hayan usado, entonces aparecen los Modelos para evaluar, en el cual un grupo de personas evalúan lo realizado (son instancias de auditoria o certificaciones), el método formal para evaluar y que una empresa sepa si adquirió un determinado nivel o capacidad de CMMI se llama **SCAMPI**.

Si quiero llamar a una evaluación necesito que ese proyecto se use en los proyectos y generar evidencia. Se genera evidencia de dos tipos: objetiva y subjetiva. Subjetiva es lo que la gente dice, en un proyecto de evaluación de scampi haces entrevistas a la gente que trabaja en los proyectos y le preguntas que hace y como. Despues eso se contrasta con la evidencia que va dejando el como consecuencia de cada tarea que hizo, que es evidencia objetiva. Objetiva y subjetiva se contrastan. Despues de juntar la evidencia, presentas haces el contraste contra lo que el modelo pide y ves si el modelo se cumple o no. Se tienen que cumplir todos los objetivos de todas las áreas de proceso del nivel. Si un área de proceso se cae (no cumple algún objetivo), se cae el nivel, y si no sos nivel 2 sos nivel 1. Si una org llama a acreditación de nivel 5 pero tiene problema con un área de nivel 2, cae a nivel 1. Si no sos nivel 2 no sos nada.

## INTERESES FUNDAMENTALES DE LA GESTIÓN DE CALIDAD

A nivel de **organización**, la gestión de calidad se ocupa de establecer un marco de proceso y estándares de organización que conducirán a software de mejor calidad. Esto supone que el equipo de gestión de calidad debe tener la responsabilidad de definir los procesos de desarrollo del software a usar, los estándares que deben aplicarse al software y la documentación relacionada, incluyendo los requerimientos, el diseño y el código del sistema.

A nivel del **proceso**, la gestión de calidad implica la aplicación de procesos específicos de calidad y la verificación de que continúen dichos procesos planeados; además, se ocupa de garantizar que los resultados del proyecto estén en conformidad con los estándares aplicables a dicho proyecto.

A nivel del **proyecto**, la gestión de calidad se ocupa también de establecer un plan de calidad para un proyecto. El plan de calidad debe establecer metas de calidad para el proyecto y definir cuáles procesos y estándares se usarán.

### Puntos clave

- △ El software puede analizarse desde varias perspectivas: como proceso, como producto, la calidad también.
- △ La calidad del software es difícil de medir.
- △ El software como proceso es el fundamento para mejorar la calidad.
- △ Trabajar con calidad es más barato.
- △ La mejora de procesos exitosa requiere compromiso organizacional y cambio organizacional.
- △ Existen varios modelos disponibles para dar soporte a los esfuerzos de mejora.
- △ La mejora de procesos en el software ha demostrado retornos de inversión sustanciales.

## CMMI CARA A CARA CON ÁGIL

CMMI y Ágil son enfoques diferentes para el desarrollo de software y la mejora de procesos. A continuación, se presenta una comparación cara a cara entre CMMI y Ágil:

### 1. Filosofía y enfoque:

- **CMMI:** CMMI se basa en un enfoque más tradicional y orientado a procesos. Se enfoca en establecer prácticas estandarizadas, mejorar la madurez de los procesos y lograr la calidad a través de la planificación y el control rigurosos.
- **Ágil:** Ágil se basa en principios y valores ágiles, como la adaptabilidad, la colaboración y la entrega de valor incremental. Se enfoca en la flexibilidad, la respuesta al cambio y la entrega temprana de software funcional.

### 2. Gestión de proyectos:

- **CMMI:** CMMI se centra en la gestión de proyectos a través de procesos planificados y controlados. Se pone énfasis en la definición y el seguimiento de requisitos, la planificación detallada, el monitoreo del progreso y el control de cambios.
- **Ágil:** Ágil promueve la gestión de proyectos mediante iteraciones cortas y frecuentes. Se enfoca en la colaboración con el cliente, la retroalimentación continua y la adaptabilidad a medida que se desarrolla el producto. La planificación es flexible y se ajusta a medida que se obtiene más información y se responden a los cambios.

### 3. Entrega de software:

- **CMMI:** CMMI tiende a enfocarse en la entrega final del software, centrándose en los aspectos de calidad, cumplimiento de requisitos y procesos establecidos.
- **Ágil:** Ágil se centra en la entrega temprana y continua de software funcional y de alta calidad. Se valora la retroalimentación del cliente y la adaptación del producto a medida que se desarrolla, lo que permite una mayor satisfacción del cliente y la capacidad de responder rápidamente a los cambios en los requisitos.

### 4. Flexibilidad y adaptabilidad:

- **CMMI:** CMMI proporciona un marco estructurado y detallado de prácticas y procesos que deben seguirse. Si bien se puede personalizar, tiene un enfoque más rígido y menos adaptable a los cambios.
- **Ágil:** Ágil se basa en la adaptabilidad y la flexibilidad para responder a los cambios en los requisitos y las necesidades del cliente. Los equipos ágiles tienen la capacidad de ajustar su enfoque y prioridades según sea necesario.

Es importante tener en cuenta que tanto CMMI como Ágil tienen sus propias fortalezas y áreas de aplicación. Algunas organizaciones pueden optar por una combinación de ambos enfoques o adaptarlos según sus necesidades específicas. La elección entre CMMI y Ágil dependerá de factores como el tipo de proyecto, la cultura organizativa, la industria y las preferencias del equipo.

El modelo dice que hacer no como. Entonces podés incorporar prácticas ágiles al proceso. Para agile con CMMI ambas tienen que ceder un poco. Agile habla de gestión ágil de reqs con un pb que se va llenando y vaciando constantemente y no te pide que tengas un control de que reqs tiene el producto a cada momento de tiempo, pero CMMI si. Podés trabajar con un pb pero tenes que ver una manera de implementar la trazabilidad de que esta user se programó acá, se diseño acá, se probó acá, y tenes que tener una manera de ir sacándole fotitos al pb para ir guardando la evolución. Entonces aceras las diferencias, uno cede un poquito y otro un poco. CMMI cede en no te voy a pedir que dejes registro de las daily, y agile se dedica por guardar la evolución del pb a lo largo del tiempo. Desde lo estricto son difíciles de compatibilizar. A nivel 2 funciona muy bien el mapeo, agile tiene que ceder en

auditorias, y CMMI cede en algunas otras cosas. Tenes que tener un evaluador que entienda el negocio y sea flexible