



Unidad 1: Ingeniería de Software en Contexto

Introducción:

Ingeniería de Software

Introducción:

La **crisis del software** es un concepto que surge en los años **60 y 70** cuando se empezaron a evidenciar graves problemas en el desarrollo de 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 recalcó la dificultad para generar software libre de defectos, fácilmente comprensibles y que sean verificables.

A medida que los sistemas de software crecían en **complejidad y magnitud**, los proyectos comenzaron a sufrir **retrasos, sobrecostes y fallos** en cumplir con las expectativas iniciales.

Con respecto a las **principales causas**:

- Se puede decir que el **hardware avanzó muchísimo** (debido al avance de los circuitos integrados) mientras que el software se quedó atrás en ese momento, debidos a que esta es una actividad humana.

- **Demandas crecientes:** a medida que la tecnología y las técnicas de desarrollo evolucionan, la demanda de sistemas más grandes y complejos aumenta. Entonces, durante esa época, el desarrollo de software experimentó un auge debido al creciente uso de computadoras en diversos sectores, desde la industria hasta la ciencia.
- En ese momento, se utilizaban metodologías, que en su mayoría eran **ad hoc** y carecían de rigor estructural, no podían gestionar adecuadamente el desarrollo de sistemas cada vez más grandes y críticos.

Entre los principales problemas que comenzaron a surgir se encuentran:

- **Retrasos en la entrega de proyectos:** Los desarrollos de software frecuentemente se extendían más allá de los plazos originalmente estimados, afectando tanto la economía de las empresas como la eficiencia operativa de los usuarios.
- **Sobrecostes:** Los presupuestos iniciales de los proyectos eran a menudo subestimados, lo que resultaba en incrementos significativos de los costos, que en algunos casos duplican o triplican los valores previstos.
- **Calidad deficiente:** El software que se entregaba a menudo estaba plagado de errores o defectos, lo que provocaba fallos frecuentes, mal rendimiento y una experiencia de usuario insatisfactoria.
- **Mantenibilidad limitada:** A medida que el software evoluciona, se hacía más difícil modificarlo o mantenerlo debido a la falta de documentación adecuada, el uso de técnicas no sistemáticas y la poca modularidad en su diseño.

¿Qué es una metodología Ad Hoc?

Cuando hablamos de metodologías ad hoc nos referimos a las prácticas de desarrollo de software en ese momento eran improvisadas, informales y dependían en gran medida de la experiencia individual de los programadores. En otras palabras, cada equipo o empresa desarrolla software a su manera, sin seguir procesos bien definidos, lo que lleva a problemas.

¿Cómo nos encontramos hoy en día?

El avance de la **Ingeniería de Software** como disciplina ha permitido abordar muchos de estos desafíos a través de la introducción de metodologías, herramientas y técnicas más avanzadas.

Algunos de los avances que nos han permitido gestionar un poco la crisis son:

- Entornos integrados de desarrollo (IDE)
- Testing
- SCM y lenguajes como git

Concepto:

La ingeniería de software es el establecimiento y uso de principios fundamentales de la ingeniería con objeto de desarrollar de forma económica software que sea confiable y que trabaje con eficiencia en máquinas reales

Es importante entender que debe hacerse ingeniería con el software en todas sus formas y a través de todos sus dominios de aplicación

La ingeniería de Software es **importante** por dos razones:

1. Cada vez con mayor frecuencia, los individuos y la sociedad se apoyan en los avanzados sistemas de software. Por ende, se requiere producir económica y rápidamente sistemas confiables.
2. A menudo resulta más barato a largo plazo usar métodos y técnicas de ingeniería de software para los sistemas de software, que sólo diseñar los programas como si fuera un proyecto de programación personal. Para muchos tipos de sistemas, la mayoría de los costos consisten en cambiar el software después de ponerlo en operación.

Disciplinas que conforman la Ing. de Software:

- **Disciplinas técnicas:** Nos referimos a actividades que aportan al desarrollo de software como producto. Ellas son:
 - Toma de requerimientos
 - Análisis de requerimientos

- Diseño de software
 - Implementación de software
 - Prueba de software
 - Despliegue de un software
 - Documentación de software
 - Capacitaciones a usuarios
- **Disciplinas de gestión:** Hacen referencia a actividades de planificación, monitoreo y control del proyecto de desarrollo de software.

Ejemplo: Dentro del marco Lean, tenemos a Kanban el cual utiliza tableros para monitorear el proyecto

- **Disciplinas de soporte:** Son disciplinas transversales a los procesos de software, que permiten verificar la integridad y calidad de un producto de software. Entre ellas se incluyen:

- Gestión de configuración de Software (SCM)
- Toma de métricas
- Aseguramiento de calidad (QA)
 - Calidad en el producto
 - Calidad en el procesos
 - Pruebas de software (Testing)

¿Y qué se entiende por **software**?

Colección de programas necesarios para convertir a una computadora (de propósito general) en una máquina de propósito especial diseñada para una aplicación de dominio particular, incluyendo documentación.

Componentes de Proyecto de Software

Proceso de desarrollo de Software

Concepto:

Primero que nada se entiende por **proceso**, a la secuencia de pasos ejecutados para un propósito dado (IEEE)

Un proceso de software es un conjunto de actividades, métodos, prácticas y transformaciones que la gente utiliza para desarrollar o mantener software y sus productos asociadas

De acuerdo con Pressman, el proceso de software tiene un conjunto de **actividades estructurales** genéricas, las cuales son:

1. **Comunicación:** Antes de comenzar el laburo técnico, es importante comunicarse con el cliente para entender los objetivos del proyecto y reunir los requerimientos que me ayuden a definir el software.
2. **Planeación:** Como todo viaje, si no tenemos un mapa nos perdemos. Acá se diseña el *plan de proyecto de software*, en el cual se describen las tareas técnicas a realizar, los riegos, los recursos y los productos que se tendrán
3. **Modelado:** Como lo dice el nombre se crea un modelo (simplificación abstracta de la realidad) del producto, en donde se visualiza arquitectónicamente el mismo y cómo se relacionan las partes que lo conforman
4. **Construcción:** Acá intervienen la generación de código y pruebas que descubren los errores
5. **Despliegue:** El software como entidad completa es entregado al consumidor quien lo evalúa y le da retroalimentación

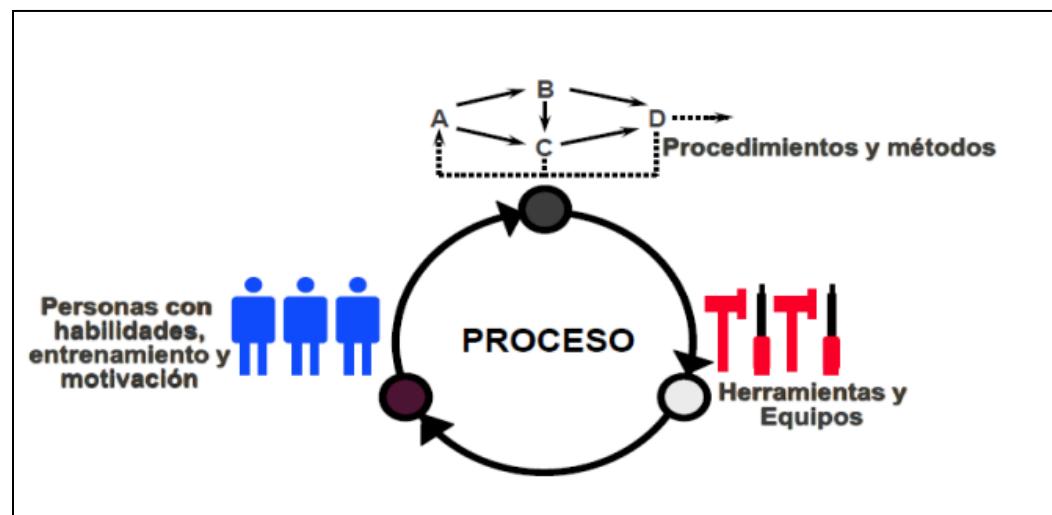
El **orden y la relación** entre las diferentes actividades está determinada por la elección del ciclo de vida.

Es importante entender que en el contexto de la ingeniería de software, un proceso no es una **prescripción rígida** acerca de cómo elaborar software. Por el contrario, es un **enfoque adaptable** que permite que las personas que hacen el trabajo (el equipo de software) busquen y elijan el conjunto apropiado de acciones y tareas para el trabajo.

En un proceso hay **3 factores determinantes**:

- **Procedimientos y métodos:** Representan cómo se hacen las cosas, es decir, el conjunto de métodos, prácticas, normas o estándares que guían el trabajo.
- **Personas capacitadas y motivadas:** Es factor primordial para obtener la calidad del producto del proceso, ya que éste se determina del esfuerzo de las personas. Sin ellas, no se puede lograr construir ningún producto. Por ello, deben estar capacitadas y con habilidades para realizar sus tareas asignadas de la manera correcta y motivados, para lograr aún mayor eficiencia. **Recordar que el software es una actividad humano-intensiva.**
- **Herramientas y Equipo:** Son todos los recursos que soportan el desarrollo del proceso de software. Esto va desde herramientas de software como IDE, herramientas de gestión de proyectos como Jira hasta el hardware necesario como servidores, etc

Estas herramientas ayudan a automatizar tareas, aumentar la productividad y reducir errores.

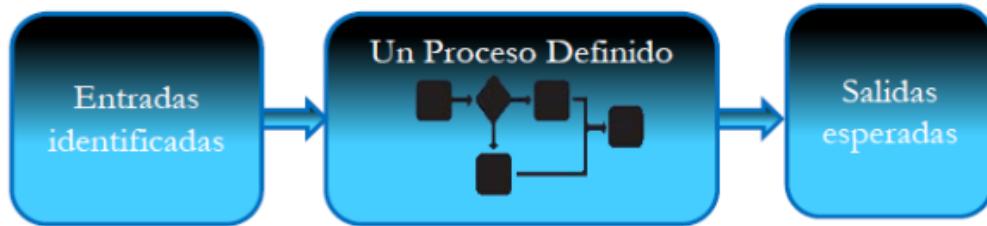


Proceso definido:

Los procesos definidos son considerados **deterministas**: ante la **misma entrada** se pretende que se obtendrá la **misma salida**. Asumen que, si se aplican una y otra vez, se obtendrán siempre los mismos resultados, a pesar de cambiar de equipo.

Algunas de las **características** pueden ser:

- Están inspirados en las **líneas de producción**
- La **repetición** y la **previsibilidad** son claves en estos procesos
- Puedo usar **cualquiera de los ciclos de vida**



Estos procesos intentan ser **completos** en el sentido que describen todos lo esperable en el proceso, y los artefactos que se esperan obtener en cada etapa (Como lo es el PUD)

La **administración y control del proceso**, en muchos de los casos, toman más importancia que el avance del software en sí (Termina siendo más importante la documentación que el avance del proyecto en sí mismo).

Proceso empírico:

Los procesos empíricos se conforman en **base a la experiencia interna y externa** de las personas en un contexto particular.

Dado que el factor diferencial es la experiencia sensible, en estos procesos se pueden obtener diferentes resultados dependiendo del contexto en el cual se apliquen

Como lo dice el nombre, el **empirismo = experiencia**. Hay que aprender de manera constante de la experiencia del equipo → Generar conocimiento

Algunas **características son:**

- Son procesos que trabajan bien con problemas **creativos y complejos**
- Se basan en **ciclos cortos de inspección y adaptación** (retroalimentación), porque ante un análisis, se ajustan de mejor forma en aquellos dominios complejos donde prima la creatividad y/o complejidad
- Se basa en **ciclos de entrega cortos** para así poder generar retroalimentación
- La **administración y el control** es por medio de inspecciones frecuentes y adaptaciones para lograr buenas prácticas.
- No se pueden combinar con cualquiera de los tipos de ciclos de vida vistos, se suele recomendar que sea el **modelo iterativo-incremental**, y el que se suele prohibir es el modelo secuencial debido a que no es viable después de 2 años (lo dice una teoría).
- **La experiencia no es extrapolable** → Esto quiere decir que los conocimiento y la experiencia que yo haya generado con un grupo de trabajo en una empresa X, no serán los mismos que los que yo tenga en una empresa Y con otro grupo.

Ahora bien, los principios empíricos se basan en **3 pilares fundamentales**:

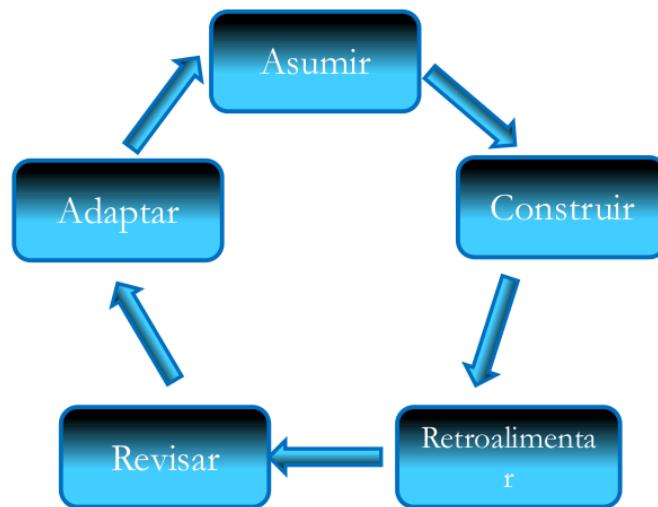
- **Transparencia:** Se centra en la **comunicación abierta y sin obstáculos**.
La transparencia **es la base de la confianza** y la colaboración, ya que **promueve** un **intercambio de información** claro y sincero entre todas las partes interesadas del proyecto.

Transparencia en el procesos y en el producto

- **Inspección:** Los equipos deben **identificar las desviaciones mediante evaluaciones periódicas**, lo que fomenta la mejora y mantiene la trayectoria hacia el éxito del proyecto.

- **Adaptación:** Una vez que el equipo ha inspeccionado el producto y los procesos, adapta sus estrategias en función de los conocimientos adquiridos. A medida que los equipos descubren nueva información y entienden mejor la dinámica de sus proyectos, pueden corregir el rumbo de una forma ágil.

El **patrón del conocimiento** de los procesos empíricos se basa en:



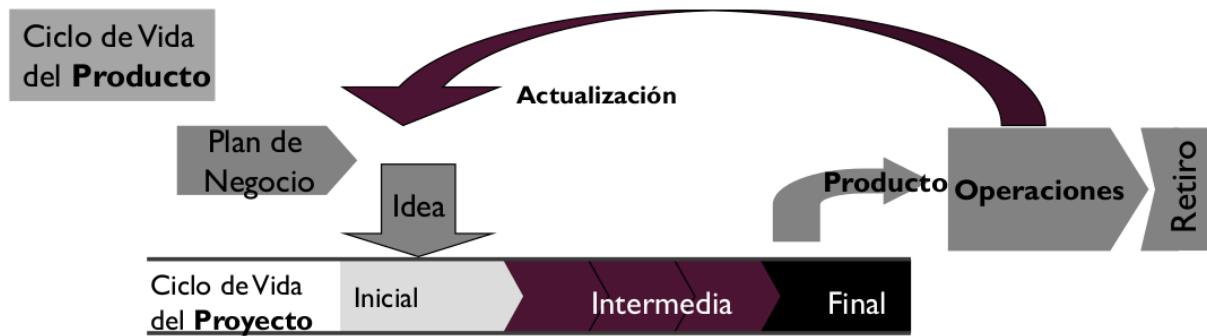
Nota: Es posible que las partes del gráfico este no se entienda mucho pero básicamente vos asumis algo, es decir que partís de una hipótesis, luego la construimos y en base a eso obtener feedback retroalimentación y luego de esto realizamos retrospectivas (revisar). Por último adaptas es decir ves cómo podrías mejorar tu proceso

Ciclos de Vida:

Concepto:

Se entiende por **ciclo de vida**, a una serie de pasos a través de los cuales el producto o proyecto progresan. Los productos y proyectos tienen su ciclo de vida.

Es decir, que es una **representación gráfica simplificada** de un proceso desde una perspectiva en particular, el cuál define **elementos del proceso** (actividades estructurales, productos del trabajo, tareas, etc.) y el **flujo del proceso** (o flujo de trabajo), que especifica la relación y el orden de dichos elementos.



El ciclo de vida del producto siempre es mayor que el ciclo de vida del proyecto, ya que el ciclo de vida del proyecto dura lo que dura el desarrollo del software, mientras que el ciclo de vida del producto dura hasta que el software se deje de utilizar, dependiendo esto de la madurez que tenga el producto en base a las necesidades del mercado.

Es posible que un producto tenga varios proyectos en su ciclo de vida, debido a constantes cambios y/o actualizaciones que se vayan realizando. Por lo que, dentro del ciclo de vida del producto, se pueden desarrollar varios ciclos de vida de proyectos.

Nosotros vamos a **hacer enfoque en los ciclos de vida del proyecto de software**

Tipos de ciclos de vida:

Hay tres tipos básicos de Ciclos de Vida para un proyecto de desarrollo de software:

1. Secuencial:

- Este modelo dispone de las **actividades de forma lineal**, es decir, que el proyecto progresó a través de una **secuencia ordenada de pasos** (fases) y una actividad no puede iniciar sin que la precedente haya sido finalizada.

El avance entre las fases se da tras una **revisión al final** de cada una de estas para determinar si el software está listo para avanzar. Un **hito** es un punto de control importante dentro del proceso de desarrollo, que marca la finalización de una fase y la verificación de los entregables antes de avanzar a la siguiente etapa.

- Se suele utilizar en aquellos sistemas donde los requerimientos se comprenden bien y son exhaustivos ya que éstos se **congelan** al finalizar la etapa de toma de requerimientos.
- Generalmente, este tipo de modelos está **dirigido por documentos**, es decir, el trabajo principal del producto es la **documentación del software** entre las distintas fases.
- El modelo en cascada es el ejemplo por defecto
- ¿Y qué dificultades se presentan?
 - Los proyectos raramente siguen un flujo secuencial
 - El modelo secuencial exige que los requerimientos sean completamente explícitos desde un principio y esto en la realidad no suele suceder. **No puede lidiar con la incertidumbre.**
 - El **cliente obtiene una versión funcional del producto en etapas muy avanzadas del proyecto.**
 - Encontrar un defecto implica un gran rediseño en la solución, ya que está prácticamente todo hecho.
 - **Dependencia entre los equipos de trabajo.** Un equipo no puede comenzar hasta que el otro no haya finalizado.
 - El desarrollo está dirigido por un plan, y cada etapa general documentación para realizar un monitoreo constante contra el plan, por lo que esa **documentación puede llegar a ser muy burocrática y excesiva.**



Frase Clave:

En el modelo secuencial, los errores técnicos se corrigen en pruebas, pero los errores en los requisitos son muy costosos de solucionar, porque el modelo **no está pensado para cambios** una vez que se congelan los requisitos.

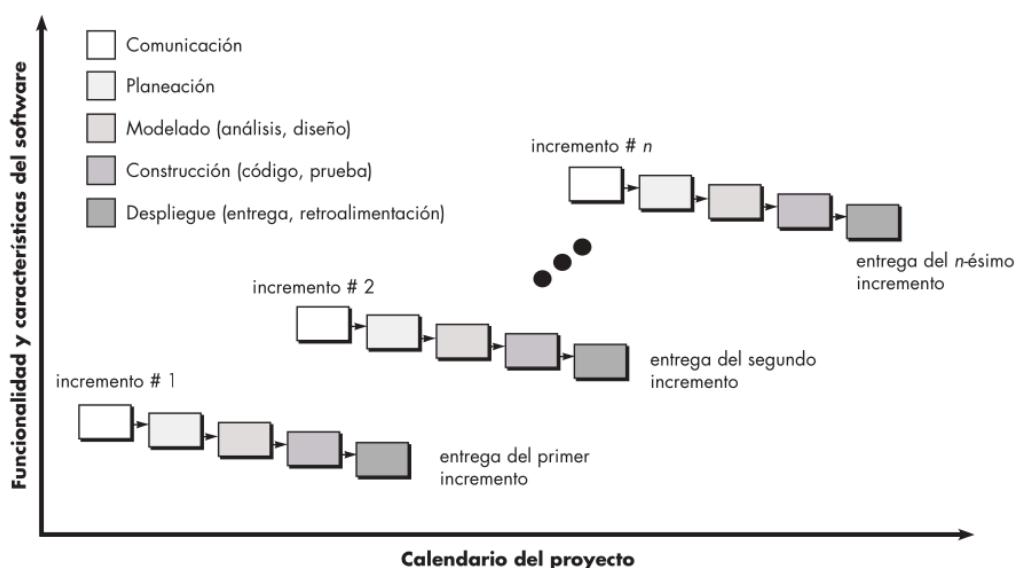
2. Iterativo / Incremental:

Este modelo aplica sucesivas **iteraciones en forma escalonada** a medida que avanza el calendario de actividades. Cada iteración produce un incremento de software funcional potencialmente entregable.

Fíjate que la **iteración** sería una repetición de un conjunto de actividades del proceso de desarrollo (como análisis, diseño, codificación y pruebas) con el objetivo de refinar, mejorar o evolucionar el producto de software.

El **incremento** sería el resultado de la iteración en donde se obtiene una versión funcional cada vez más completa del software

Usualmente los primeros incrementos incluyen las funciones básicas/críticas que más requiere el cliente. Este enfoque vincula las actividades de especificación, desarrollo y validación. El sistema se desarrolla como una serie de versiones (incrementos) y cada una añade funcionalidades a la versión anterior.



¿Y qué ventajas tenemos con respecto al modelo anterior?

- Se reduce el **costo** de adaptar los requerimientos cambiantes del cliente y el retrabajo es menor ya que en cada incremento se obtiene una retroalimentación que permite adaptar el modelo a las necesidades del cliente.
- El **cliente** es parte del proceso de desarrollo. En el modelo secuencial, el cliente debía esperar hasta la última instancia para recibir el producto y no era capaz de juzgar el avance del producto a través de documentos de diseño de software.
- La entrega es mucho más **rápida**. Los clientes tienen la posibilidad de usar y ganar valor del software más temprano de lo que sería posible con un proceso en cascada.

Con respecto a las desventajas:

- Se **invisibiliza** el proceso
 - *Como no hay una documentación tan exhaustiva como antes, quizás es complicado para personas externas al proceso entender cómo funciona (stakeholders)*
- La **estructura del software** tiende a degradarse frente a una cantidad considerable de incrementos
- **Alto costo** de documentar cada incremento
- A menudo genera mucha fricción con **organizaciones altamente burocráticas** que necesitan mucha documentación y pasos definidos

Un ejemplo de este ciclo de vida es el PUD.

3. Recursivo

El modelo recursivo es utilizado para gestionar los riesgos del desarrollo de **sistemas complejos** a gran escala. Requiere de la intervención del cliente.

¿Y en qué se diferencia con el iterativo - incremental?

EL iterativo - incremental realiza entre el producto de a incrementos o partes, pero que cada parte es funcional, es decir que cada una otorga algún tipo de valor, por ejemplo si estoy desarrollando Microsoft Word, los incrementos podrían ser:

1. Motor de texto
2. Barra de tareas
3. Barra de tareas avanzadas
4. Soporte a colaboración online

Por otro lado el recursivo, toma **todo el alcance del producto**, y lo va refinando en cuanto a terminarlo, pero la versión productiva no llega sino hasta el final.

Acá no hay versiones intermedias, solamente se va refinando el producto

Ejemplo: Tendríamos que estar ante un proceso sumamente crítico y complejo en el que la gestión de riesgos se clave → Un software para un satélite

Desventajas desde un punto de vista administrativo:

- Puede ser más costoso en tiempo y dinero adaptarlos para utilizarlos para diferentes proyectos.
- La tecnología puede ser obsoleta
- Pueden Carecer de mantenimiento o documentación, lo cual dificulta mucho las tareas
- Se genera una versión del producto recién al final

Administración de proyectos segun ciclo de vida:

La elección de un ciclo de vida afecta de forma directa a la administración que se debe realizar sobre el proyecto, ya que cada uno de los modelos de proceso poseen características y enfoques distintos.

Por ejemplo, si se realiza un análisis de riesgo y se determina que los requerimientos pueden variar mucho a lo largo del desarrollo, se debería elegir un ciclo de vida en el cual el cambio sea permisible, como los iterativos. En este caso, la administración y la planificación del proyecto se ve afectada en la elección del ciclo de vida.

Además, durante el desarrollo del proyecto, la gestión de este se realiza en consecuencia de esta elección, ya que no es lo mismo gestionar un proyecto que se desarrolla de manera iterativa, que uno que se desarrolla de manera secuencial o en cascada.

1. Si los requerimientos son claros y exhaustivos, en un contexto de baja incertidumbre en el que el cliente no requiere obtener el producto rápidamente, es posible aplicar el modelo secuencial.
2. Si la situación anterior no se presenta y el cliente desea obtener resultados lo antes posible, con las funciones principales del sistema, entonces sería conveniente la elección del modelo iterativo- incremental.
3. Si el objetivo es disminuir los riesgos presentes en el desarrollo de sistemas grandes y complejos, entonces el modelo recursivo puede ser una buena opción.

Algunos modelos de proceso o ciclos de vida:

Estos modelos de proceso han sido cortesía del libro Pressman y de Rapid Development.

Esta definición no se si esta bien pero los modelos de proceso serían como **implementaciones particulares** que uno realiza a partir de un tipo ciclo de vida

Modelo de Cascada Puro:

El modelo de la cascada, a veces llamado ciclo de vida clásico, sugiere un enfoque sistemático y secuencial para el desarrollo del software, que comienza con la especificación de los requerimientos por parte del cliente y avanza a través de planeación, modelado, construcción y despliegue, para concluir con el apoyo del software terminado



Es importante entender esto, en el modelo de cascada **si se puede volver para atrás**, no es que no esté permitido, simplemente es demasiado costoso y complejo.

Supongamos que hemos diseñado un auto y cuando estamos construyendolo viene el cliente y te dice “*Cuando el auto hace marcha atras quiero que las luces se prendan de color rojo*”. Bueno no es que tenemos que esperar a terminar el auto y hacer otro nuevamente con esa modificación, **podemos ir para atrás si**, pero eso involucra una cantidad enorme de **retrabajo**.

- **Modelo en cascada con fases solapadas (Sashimi):**

Mientras que el modelo clásico sigue una secuencia estricta donde cada fase debe completarse antes de iniciar la siguiente, el modelo con fases solapadas **permite que las fases se inicien antes de que la fase anterior haya concluido por completo**.

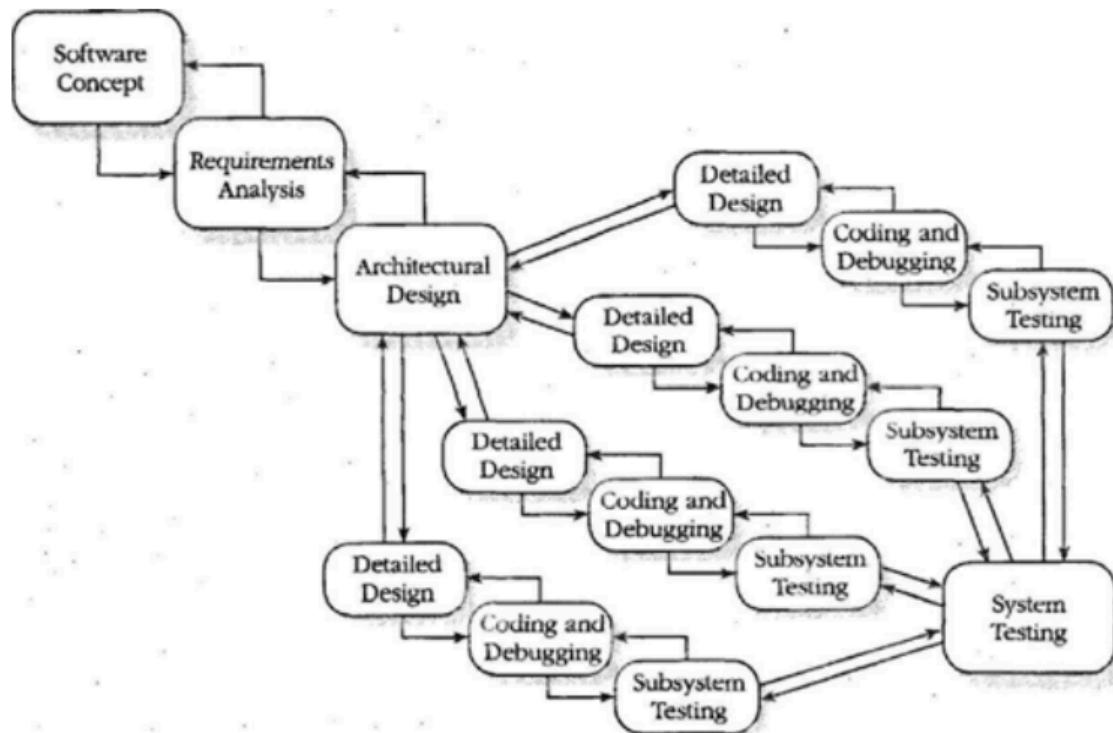
Una de sus principales desventajas es que al ser solapadas las fases, los **hitos son muy ambiguos**, es complicado trackear el progreso.

- **Cascada con subproyectos:**

Uno de los problemas del modelo de cascada puro es que se supone que debes terminar muy detalladamente el diseño antes de pasar a la fase de codificación y debuggear.

Entonces surge la pregunta → *Por qué retrasar la implementación de áreas que son fáciles de diseñar sólo porque estamos esperando al diseño de áreas difíciles*

Entonces lo que se hace es dividir el sistemas en **subsistemas lógicamente independientes**, de manera que cada uno puede diseñarse e implementarse a la vez



EL riesgo principal de este modelo son las interdependencias que no son previstas

Modelo Code - and - Fix:

Este modelo es comúnmente utilizado pero **rara vez útil**. Es un modelo que se caracteriza por ser informal, se empieza con una idea general de lo que se quiere hacer (se pueden tener especificación como no), luego se va directo a un diseño informal, código, debug y testing.

Cuenta con **2 ventajas** principales:

- No cuenta con gastos iniciales: Básicamente no se gasta tiempo y plata en la planificación, documentación, aseguramiento de calidad etc lo cual nos permite **saltar directo al código** lo que genera signos de avance inmediatos
- Se requiere muy poco expertiz: Básicamente cualquiera que sepa hacer un simple programa para la compu ya está familiarizado con este modelo

Es evidente que utilizar este modelo resulta **peligroso**, no hay algún tipo de medida de progreso o calidad, identificación de riesgos, etc. Solamente se codea hasta que se termina.



Modelo en espiral:

Si el modelo code and fix era la simplicidad en su máxima expresión ahora nos vamos a la otra punta (complejidad)

El modelo en espiral es un modelo **orientado al riesgo** que divide el proyecto en una serie de **miniproyectos**. Cada miniproyecto se ocupa de uno o más riesgos importantes

El concepto de **riesgo** es algo muy amplio, puede referirse a:

- Requerimientos pobremente entendidos
- Arquitectura pobremente entendida
- Errores de rendimientos
- Tecnología que no entendemos

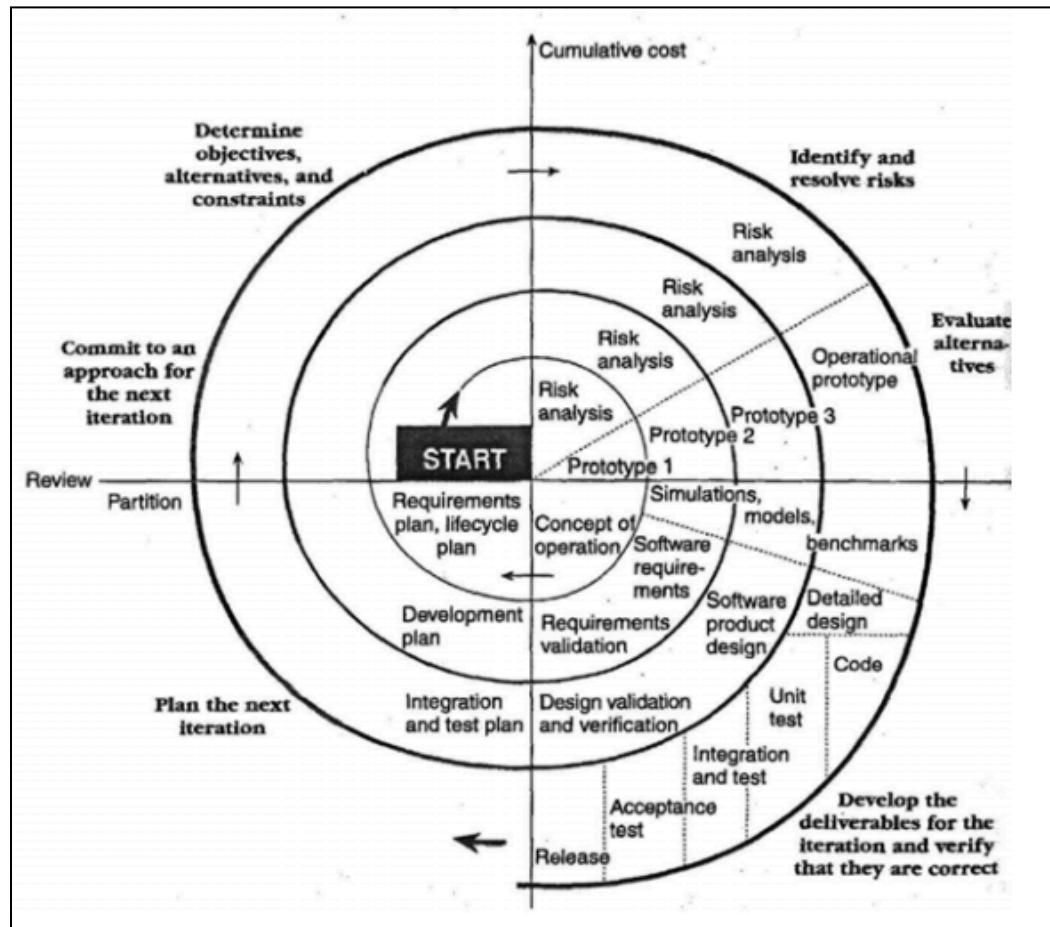
Una vez que todos los riesgos importantes han sido solucionados, el modelo en espiral termina como si lo haría un modelo en **cascada**

Fíjate que sobre cada uno de estos mini-proyectos se producen las siguientes **iteraciones** que consisten en 6 pasos:

1. Determinar objetivos, alternativas y restricciones
2. Identificamos y resolveremos riesgos
3. Evaluamos las alternativas
4. Desarrollamos entregables para cada iteración y verificamos si son correctos
5. Planear la próxima iteración
 - Se planifica el testing, el desarrollo, etc.
6. Comprometerse con un enfoque para la próxima iteración

En el modelo en espiral las iteraciones tempranas son las más baratas. Básicamente a medida que el modelo avanza a medida que el costo aumenta el riesgo disminuye

La única **desventaja** del modelo por así decirlo es que es complicado.

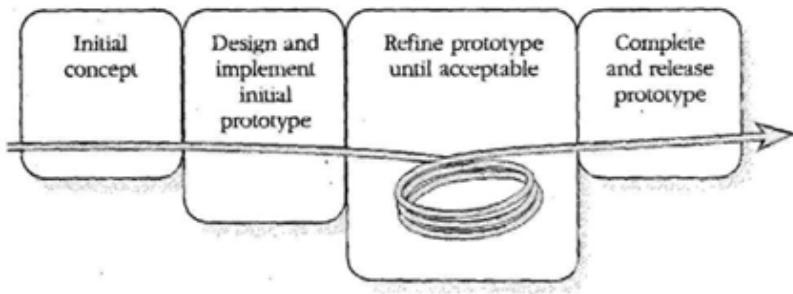


Prototipado evolutivo

El prototipado evolutivo es un modelo de ciclo de vida en el que se desarrolla el concepto del sistema a medida que se avanza en el proyecto.

Normalmente se empieza desarrollando los aspectos más **visibles** del sistema. Esto luego es presentado al cliente y luego continuamos desarrollando el prototipo en base al **feedback** del cliente

Va a llegar un momento en el que el cliente diga “*bueno esto es suficiente*” y recien ahí podemos empezar a desarrollar el producto final



Este modelo de proceso es **útil** cuando los requerimientos cambian rápido y el cliente es reacio a comprometerse a una serie de requerimientos.

La **desventaja** es que es imposible de conocer al principio del proyecto cuánto tiempo va a tardar en crearse el producto

Criterios para la selección de un ciclo de vida:

Ciclo de vida	Procesos de desarrollo que lo admite	Características que lo hacen elegible
Secuencial	Definidos	<ul style="list-style-type: none"> - Alta certeza - La organización trabaja de manera tradicional - No se puede realizar entregas parciales del software - Eliminar riesgos de planificación
Iterativo / Incremental	Definidos o Empírico	<ul style="list-style-type: none"> - Existe incertidumbre - Volatilidad de los requerimientos - Posibilidad de entregas parciales - Uso en etapas tempranas del producto
Recursivo	Definidos	<ul style="list-style-type: none"> - Mucho control de riesgos en cada iteración

		- No se pueden realizar entregas parciales
--	--	--

Proyecto:

Un proyecto de software es un **esfuerzo temporal** que requiere del acuerdo de un conjunto de especialidades y recursos para la obtención de un determinado resultado

Es una **unidad organizativa** para adaptar el marco teórico del proceso, dependiendo de las personas y recursos con los que se cuente.

Define el proceso y tareas que se van a realizar, el personal que se encargará de las actividades y los mecanismos que se implementarán para valorar riesgos, controlar el cambio y evaluar la calidad.

Características de los proyectos:

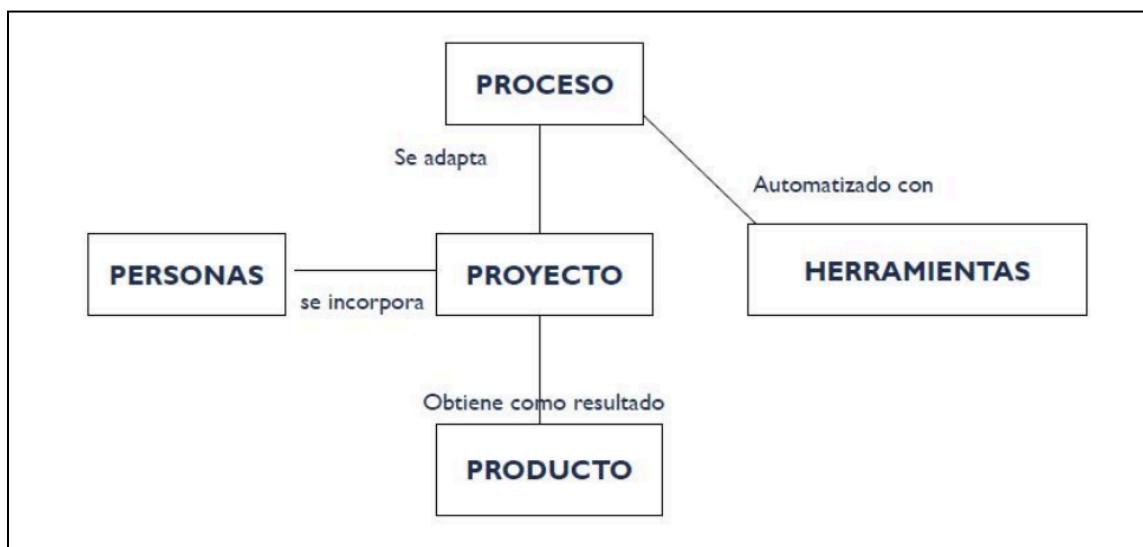
- **Orientación a objetivos:**
 - Los proyectos están dirigidos a obtener resultados y ello se refleja a partir de los objetivos
 - Los objetivos guían al proyecto
 - Los objetivos no deben ser **ambiguos**, y no solo esto sino que deben ser **alcanzables**
- **Duración Limitada:** → Característica distintiva
 - Los proyectos son temporales, cuando se alcanzan los objetivos, el proyecto se termina
 - Una línea de producción no es un proyecto
- **Tareas interrelacionadas basadas en esfuerzos y recursos:**

Básicamente necesito definir cuales son los recursos que necesito en términos de plata, personas, hardware que necesito para ejecutar el software, etc

- **Son únicos:**

- Cada proyecto es una instancia única e irrepetible de un proceso.

Vinculo proceso - proyecto - producto en la gestión de un proyecto de sw:



Se dice que el proceso, automatizado con herramientas, se adapta al proyecto, al cual se incorporan personas, y de este se obtiene como resultado un producto.

- **Personas:** Los principales autores de un proyecto Software son los arquitectos, desarrolladores, ingenieros de prueba y el personal de gestión
- **Producto:** Es evidentemente lo que se crea gracias al proyecto. Aca la diferencia principal entre producto y proyecto, es que el proyecto es **temporal** es decir que empieza y termina mientras que el producto no; él mismo sobrevive al proyecto

- **Proceso:** Su definición está más arriba en el resumen. Lo que si agregamos acá es que el proceso es una plantilla, una descripción abstracta que se materializa al definir un proyecto
 - https://www.youtube.com/watch?v=Om08nxC9NoQ&list=PLsl9mUiT17yxu5x12_pGNW_Afuf_IY4x2&index=2 (Min 7:00)
- **Herramientas:** Software que se utiliza para automatizar las actividades definidas en el proceso
 - Para **control de versiones:**
 - Github
 - Para **gestionar el proyecto:**
 - Jira
 - Asana
 - Trello
 - Para **gestionar pruebas:**
 - Selenium (pruebas automáticas web)

Nota:

- SaaS = Software as Service
- SaaP = Software as Product

Administración de Proyectos (Tradicional):

Administración de Proyecto es la aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades del proyecto para satisfacer los requerimientos del proyecto.

Es una parte fundamental para que el proyecto sea exitoso

- *Es tener el trabajo hecho en tiempo, presupuesto acordado y con los requerimientos satisfechos*

Una buena gestión **no puede garantizar** el éxito del proyecto (Pero si el fracaso si el proyecto no es administrado). Un proyecto planificado también puede fracasar, lo importante es que planificaste y no el resultado en sí. Importa que vos te sentaste y te pusiste a pensar en objetivos, riesgos, alcances, estimaciones, etc.

Administrar un proyecto incluye:

- Identificar los requerimientos
- Establecer objetivos claros y alcanzables
- Adaptar las especificaciones, planes y el enfoque a los diferentes intereses de los involucrados (stakeholders)

Metas importantes de la administración de proyectos:

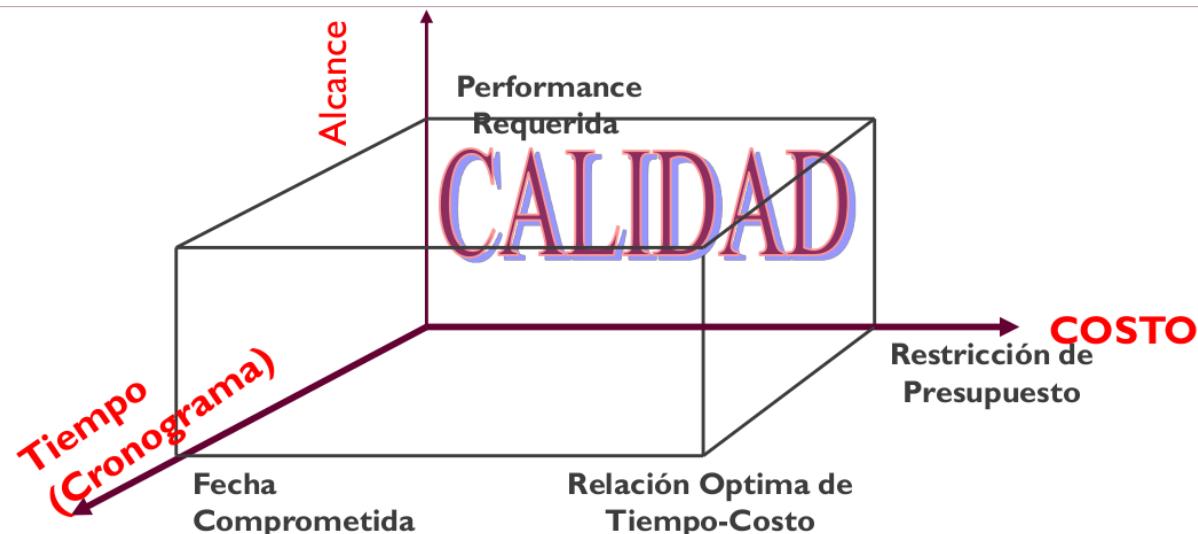
- Entregar el software al cliente en el tiempo acordado
- Mantener los costos dentro del presupuesto
- Entregar un software que cumpla con las expectativas
- Mantener un equipo de desarrollo óptimo y con buen funcionamiento

La restricción triple:

- **Objetivos del proyecto:** ¿Qué está el proyecto tratando de alcanzar?
- **Tiempo:** ¿Cuánto tiempo nos debería llevar completarlo?
- **Costos:** ¿Cuánto debería costar

El balance de estos factores afecta directamente a la **calidad del proyecto:** *Proyectos de alta calidad entregan el producto requerido, el servicio o resultado, satisfaciendo los objetivos en el tiempo estipulado y con el presupuesto planificado.*

Es responsabilidad de **Líder del Proyecto** balancear estos tres objetivos (que a menudo compiten entre ellos)



¿Y cuáles son los elementos principales en la **Adm. de un proyecto tradicional**?

Líder del proyecto

En un enfoque tradicional, el líder de proyecto en primera instancia define el **alcance del producto**, luego ejerce la **toma de decisiones** en su totalidad sin la inclusión de la opinión del equipo.

El líder **asigna las tareas** que deben realizar los integrantes del equipo, y además, maneja todas las relaciones con todos los **stakeholders** del proyecto (clientes, gerencias, contratistas, stakeholders en general).

Dentro de este enfoque, como líderes tenemos la responsabilidad de realizar **estimaciones, planificaciones y el seguimiento** de lo que está pasando.

También se encarga de la identificación de **riesgos** y generaciones de **reportes**

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.

Equipo de Proyecto

Un equipo de proyecto es un grupo de personas comprometidas en alcanzar un conjunto de objetivos de los cuales se sienten mutuamente responsables.

Informa cuando ya está terminado, informa cuánto se demoró, informa el porcentaje de avance y si hay una desviación respecto a los planes informar al líder para que tome ciertas correcciones

Entre sus características se encuentran:

- Diversos conocimientos y habilidades
- Posibilidad de trabajar juntos efectivamente / desarrollar sinergia
- Usualmente es un grupo pequeño
- Tienen sentido de responsabilidad como una unidad

Stakeholders:

Son los interesados del proyecto, incluye el equipo de proyecto, el equipo de dirección, el líder del proyecto y el patrocinador

El plan de proyecto:

El plan del proyecto es un artefacto que cumple la función de hoja de ruta en un proyecto.

El plan de proyecto documenta:

- ¿Qué es lo que hacemos?
- ¿Cuándo lo hacemos?
- ¿Cómo lo hacemos?

- ¿Quién lo va a hacer?

¿Qué implica la planificación de proyectos de SW?

- Definición del alcance del proyecto
- Definición del Proceso y Ciclo de vida
- Estimación
- Gestión de riesgos
- Asignación de recursos
- Programación de proyectos
- Definición de controles
- Definición de métricas

A continuación vamos a detallar cada una de estas **actividades**:

1. Definición del alcance del proyecto:

Acá es muy importante diferenciar:

- **Alcance del producto:** Son todas las características que pueden incluirse en un producto o servicio.

Es la sumatoria de todos los requerimientos funcionales y no funcionales que el producto tiene que satisfacer, esto se define mediante la ERS (Especificación de Requerimientos de Software)

- **Alcance del proyecto:** Es todo el trabajo y solo el trabajo que debe hacerse para entregar el producto o servicio con todas las características y funciones especificadas.

Este alcance se define mediante el Plan de Desarrollo de software (Plan del proyecto); este plan cuenta con la definición del ciclo de vida, la estimación de, la gestión del riesgo, la definición de métricas, etc

¿Cómo se mide el alcance?

- El cumplimiento del alcance del producto → Se mide contra la especificación de los requerimientos

- El cumplimiento del alcance del proyecto → Se mide contra el plan del proyecto (o plan de desarrollo de software)

2. Definir un ciclo de vida y el proceso a utilizar:

Bueno acá no hay mucho que decir, simplemente hay que elegir un ciclo de vida que más se adapte a tu proyecto

Sin embargo, es importante aclarar lo siguiente. Es una confusión muy común pensar que los ciclos de vida iterativo son exclusivos de Ágil o Scrum.

Esto es mentira ya que los procesos definidos también utilizan ciclos de vida iterativos

La diferencia sustancial entre el marco ágil y los procesos como el PUD, radica en que en este último, lo que se encuentra **fijo** en la iteración es el **alcance**.

Por otro lado, en Scrum lo que nosotros dejamos **fijo** es el tiempo. *Yo voy a hacer entregas tempranas cada 2-3 semanas y te voy a entregar lo que yo alcancé a construir*

El ciclo de vida de un proyecto me define:

- Qué trabajo técnico debería realizarse en cada fase
- Quien debería estar involucrado en cada fase
- Como controlar y aprobar cada fase
- Cómo deben generarse los entregables
- Cómo revisar, verificar y validar el producto

3. Estimaciones de Software

En un proyecto de desarrollo de software, se trata de estimar para **predecir** el valor de **un elemento relacionado con el proyecto o con el producto**, por ejemplo, el tiempo que va a llevar, qué costo va a tener el sistema, cuántas personas necesito para realizar el desarrollo, etc.

En un enfoque tradicional, esta estimación de valores es realizada únicamente por el líder de proyecto. En la metodología tradicional, existe un orden definido y recomendado para realizar las estimaciones:

En la gestión tradicional, se plantea una secuencia de estimaciones que tiene que tener el siguiente orden

1. **Tamaño:** Definir el producto a construir. Esta estimación es una de las más importantes ya que esta influye en otros aspectos como el esfuerzo, el costo, tiempo de calendario, etc.

Algunos ejemplos pueden ser:

- Líneas de código (Así se hacía antes y es muy pobre este método)
- Por casos de uso / Requerimientos (Un poco más sofisticado)

2. **Esfuerzo:** Medido en horas persona lineales (no se tienen en cuenta las hora de ocio)

3. **Calendario:**

- Determinar qué días y que horas trabajar, y cuantas personas van a trabajar.
- Es necesario realizar esta estimación para poder darle una idea al cliente de cuánto tiempo se va a demorar la finalización del proyecto,

4. **Costo:** de forma similar a la estimación de la duración del proyecto, la estimación de costos depende directamente del tamaño y del esfuerzo que se necesite para el desarrollo del software, por eso es lo último que se estima. De la misma manera, la estimación de costos es muy útil para darle una idea al cliente de cuál es el costo aproximado del proyecto.

5. **Recursos criticos:**

Nota: Con respecto a las técnicas de estimación no las voy a nombrar ahora porque no tiene sentido, debido a que más adelante en el resumen se las explica con mayor profundidad

4. Gestión de riesgo:

El riesgo es un **problema** que está esperando para poder suceder y que podría comprometer el éxito del proyecto

Acá hay **2 variables** importantes a considerar:

- **Probabilidad de ocurrencia**
- **Impacto del riesgo:** *¿Que le sucede a mi proyecto si este riesgo ocurre?*

Entonces lo que se hace, es que **entre estas 2 variables** (que en esencia son numeros), **se los multiplica** y se obtiene un valor que se conoce como **exposición al riesgo**

Entonces cuando nosotros hablamos de **gestionar riesgos**, lo que nosotros estamos tratando de hacer es **bajar el número** que tiene estas 2 variables.

Entonces lo que hacemos es **priorizar los riesgos** según su exposición, lo que haremos será, a partir de la lista de los riesgos, **seleccionar el que tenga mayor exposición y gestionarlo**:

- Mitigar: Evitar que el riesgo suceda
- Elaborar plan de contingencia: Acciones que vamos a tomar para disminuir su impacto

¿Cuáles son las actividades de la gestión de riesgo?

1. **Identificación de riesgos:** Se identifican los riesgos que suponen una mayor amenaza al proceso de ingeniería de software, al software a desarrollar o a la organización que lo desarrolla (al recibir los requerimientos).
2. **Análisis de riesgos:** Para cada riesgo identificado se realiza un juicio acerca de la probabilidad y del impacto. No existe una forma certera de realizar esto. Se utilizan intervalos de probabilidad y clasificaciones de gravedad. El criterio dependerá de la experiencia de quien realice el análisis de riesgos.

3. **Planeación del riesgo:** Para cada riesgo analizado, se definen estrategias para manejarlos. Estas estrategias consisten en considerar acciones a tomar para minimizar o evitar el impacto sobre el proyecto como consecuencia de la ocurrencia de la amenaza que representa el riesgo. Otras estrategias consisten en desarrollar planes de contingencia, el cual define un plan para enfrentar una situación controversial.
4. **Monitorización del riesgo y control:** Proceso que permite determinar que las suposiciones acerca de los riesgos de producto, proceso y organización no han cambiado. Esto permite revalorar al riesgo en términos de posibles variaciones de su probabilidad e impacto. Este proceso se aplica en todas las etapas del proyecto



Algunos ejemplos de riesgos pueden ser:

- Fallo en el script de despliegue (Corroborar CI/CD)
- Alguna api puede fallar

5. Asignación de responsabilidades (o recursos):

La asignación de responsabilidades a personas, permite asignar roles a los integrantes del equipo de trabajo. En el entorno de procesos definidos (PUD, por ejemplo), estos roles se ven definidos en el concepto de trabajadores.

Dentro de un mismo proyecto, una persona puede desenvolverse en más de un rol, ya que puede que el presupuesto del desarrollo de software no sea lo suficientemente grande para permitirse tener una persona distinta por rol, o bien, el proyecto no sea lo suficientemente grande o complejo para justificar dicha adición.

Se debe hacer mucho foco en la selección del personal de un proyecto, ya que las capacidades de los integrantes del equipo afectan a la calidad del software y al correcto desarrollo del proyecto en términos de tiempo, por ejemplo, ya que el desarrollo de software es una actividad HUMANO-INTENSIVA

6. Programación de proyectos (calendarización):

En esta etapa se trata de definir en detalle, cada tarea que debe ser cumplida en el desarrollo. Se toma como base lo definido en la estimación del calendario realizada previamente, pero se refina al máximo detalle posible cada actividad.

Cuando se habla de detallar las tareas, se hace referencia a descomponer el proceso de desarrollo en actividades refinadas a nivel de días, identificando quien la realiza, cuando debe realizarse y cuánto esfuerzo debe llevar en forma teórica.

Generalmente la calendarización se realiza a través de un Diagrama de Gantt, realizado a través de alguna herramienta, como Microsoft Project.

7. Métricas de software:

Las métricas son medidas **numéricas / porcentajes** que aportan visibilidad sobre el proyecto. Son útiles para saber si nuestro proyecto está en línea con lo que nosotros planificamos o si se está desviando

El **dominio** de las métricas de software se divide en:

- **Métricas de proceso:** Saber en términos de organización la manera en la que estamos trabajando
 - *Porcentaje de proyecto que terminan a tiempo vs el porcentaje de proyecto que terminan con desvío*

Las métricas de proceso son una **despersonalización** de las métricas del proyecto → Es decir que. **todas las métricas que son del proyecto, también lo son del proceso**

- **Métricas de proyecto:** Son las que me permiten saber **si un proyecto de software que está en ejecución se está cumpliendo de acuerdo a lo planificado o no**

$(Tiempo\ Real / Tiempo\ planificado) * 100 \rightarrow$ Esto me sirve para saber si el tiempo de duración del proyecto está en línea o no con lo planificado

- **Métricas de producto:** Directamente relacionadas con el producto de software que estamos construyendo.

Están muy relacionadas con métricas de **defectos**

Las métricas del proyecto se consolidan para crear métricas de proceso que sean públicas para toda la organización del software.

¿Cuáles son las métricas básicas?

- Tamaño del producto
 - *Producto*
- Esfuerzo
 - *Proyecto*
- Tiempo (Calendario)
 - *Proyecto*
- Defecto
 - *Producto*

Un aspecto clave a la hora de tomar métricas es **mantenerlo simple** → El esfuerzo tiene que ser el menor posible

Ejemplo sencillo: → *Tiempo (calendario)*

Esto nos sirve para entender a qué nos referimos con **despersonalización**.

Supongamos que yo tomo métricas de las desviaciones de los últimos 10 proyectos de software de mi empresa y noto que el **desvío promedio** de los proyectos es de un 30 % con respecto a lo planificado; esto refleja que mi proceso de desarrollo está siendo mal implementado o mal entendido. Es decir que estamos ante una métrica de proceso

Esto es una métrica de proceso debido a que no se analiza en particular un proyecto en particular, sino que se los despersonaliza llevándolos a un nivel más superior.

$$(T.\text{real} - T.\text{planificado}) / T.\text{planificado} * 100 \rightarrow \text{Desvío porcentual del tiempo de planificación}$$

Ejemplo sencillo: → *Esfuerzo*

Esto nos sirve para entender a qué nos referimos con **despersonalización**.

Supongamos que yo tomo métricas de las desviaciones de los últimos 10 proyectos de software de mi empresa y noto que el **desvío promedio** de los proyectos es de un 30 % con respecto a lo planificado; esto refleja que mi proceso de desarrollo está siendo mal implementado o mal entendido. Es decir que estamos ante una métrica de proceso

Esto es una métrica de proceso debido a que no se analiza en particular un proyecto en particular, sino que se los despersonaliza llevándolos a un nivel más superior.

¿Qué es el monitoreo y control?

El monitoreo y control, que **es para lo que usamos las métricas**, 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. De esto se encarga el Líder de proyecto

En el libro de “*The Mythical Man Months*” se habla de que los **procesos se retrasan 1 día a la vez**. Justamente si tenemos una buena planificación y buenas métricas, un desvío no será difícil de corregir si es tomado a tiempo

8. Definición de controles de desarrollo:

En la planificación de los controles del desarrollo, se toma como base las métricas ya definidas, y verifica que todo se está haciendo en concordancia con lo establecido. En esta planificación se definen reuniones periódicas, reportes o informes necesarios, etc.

Unidad 2: Gestión LEAN/Ágil de productos de Software

Filosofía Ágil - Manifiesto Ágil

Agilismo:

Introducción:

Contexto (años 90):

- El desarrollo de software estaba dominado por métodos pesados (llamados “metodologías pesadas” o heavyweight), como CMMI, RUP o el modelo en cascada.
- Estos métodos requerían una planificación exhaustiva al inicio, documentación muy detallada y fases secuenciales, lo que dificulta adaptarse a los cambios de requisitos del cliente.
- Los proyectos fallaban con frecuencia por ser muy largos y poco flexibles.

La reunión de Snowbird (2001):

- 17 expertos en desarrollo de software se reunieron en febrero de 2001 en Snowbird, Utah (EE.UU.).
- Entre ellos estaban Kent Beck (creador de XP), Martin Fowler, Robert C. Martin (Uncle Bob), Jeff Sutherland (co-creador de Scrum), entre otros.

- Buscaban una alternativa a las metodologías tradicionales, algo más ligero, flexible y centrado en las personas.

El resultado:

- De esa reunión nació el "Manifiesto por el Desarrollo Ágil de Software", que definió 4 valores y 12 principios que guían el desarrollo ágil.

Es un compromiso útil entre nada de proceso y demasiado de proceso

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.

Empezamos con algo, lo construimos y lo mostramos para obtener retroalimentación para corregir cosas si es necesario y mejorar.

Manifiesto Ágil:

Bueno básicamente este manifiesto se divide en principios y valores

Con respecto a los **principios**, tenemos esta linda infografía que los explica muy bien:



Nota: El principio 4 quiere decir *técnicos y no técnicos trabajando juntos*

Con respecto a los **valores**, tenemos:

i. **Valorar más a los individuos y sus interacciones que a los procesos y a las herramientas:**

Es preferible un equipo de personas motivadas con un buen funcionamiento y conocimiento, donde la comunicación sea fluida, pero con herramientas pobres para su trabajo, que tener un equipo de individuos desmotivados o poco funcional con las mejores herramientas y procesos. Así lo expresa el manifiesto, debido a que el desarrollo de software es una actividad humano-intensiva

ii. **Valorar más el software funcionando que la documentación exahusativa**

Priorizar el software asegura que se tengan versiones estables, incrementales y mejoradas del software al finalizar cada iteración. Esto permite que a medida que el software crezca en cuanto a funcionalidades, tras cada iteración puede ir mostrándose a los usuarios finales y obtener una retroalimentación de ellos, para que el equipo de desarrollo se asegure de estar trabajando siempre en aquellas funcionalidades de mayor valor para el cliente y aquellas que satisfagan sus expectativas.

Solo se documenta aquello que agregue valor al producto y esté centrado en el cliente (valor agregado). Existe la necesidad de mantener información sobre el producto de software y sobre el proyecto que se realiza para obtener un producto y el enfoque ágil lo que plantea es que se debe generar la documentación cuándo sea necesario. No plantea “no generar documentación”.

iii. **Valorar más la colaboración con el cliente que la negociación contractual**

Implica comprometer al cliente como parte importante del producto que se está construyendo. No se trata de tener a un cliente lejano que no sabe lo que sucede durante el desarrollo, si no tener a uno que esté comprometido con cada entrega, sobre todo al momento de probarlo y dar el feedback.

A veces las negociaciones contractuales imponen ciertas restricciones que impactan de manera negativa en las partes, lo cual se traduce en un mayor distanciamiento en la relación equipo de desarrollo-cliente

iv. **Valorar más la respuesta ante el cambio a que seguir un plan**

Para un modelo de desarrollo que surge de entornos inestables, que tienen como factor inherente el cambio y la evolución rápida y continua, resulta mucho más valiosa la capacidad de respuesta ante el cambio que la de seguimiento y aseguramiento de planes preestablecidos

El manifiesto ágil se sustenta en procesos empíricos y por lo tanto en los pilares del empirismo, es por esto que la filosofía ágil se adapta muy bien a los cambios debido a que incorpora estos ciclos cortos y de inspección y adaptación.

Agilismo - Concepto:

El agilismo **no es** una metodología o un proceso ; es una **ideología** con un conjunto definido de principios que guían el desarrollo del producto

El **objetivo** primordial es **entregar software de forma frecuente en un entorno cambiante**. Estas metodologías se utilizan en **entornos** con gran **variabilidad** de **requerimientos**, involucrando al **cliente** en el proceso de desarrollo para conseguir una rápida retroalimentación.

El desarrollo ágil adopta el **ciclo de vida iterativo-incremental**, donde el concepto del empirismo subyace al proceso, el software no se desarrolla como una sola unidad, sino como una serie de incrementos y cada uno de ellos incluye una nueva funcionalidad del sistema. Esta es una de las diferencias con el PUD o el RUP, que, si bien utilizan un modelo de proceso iterativo - incremental, son procesos definidos.

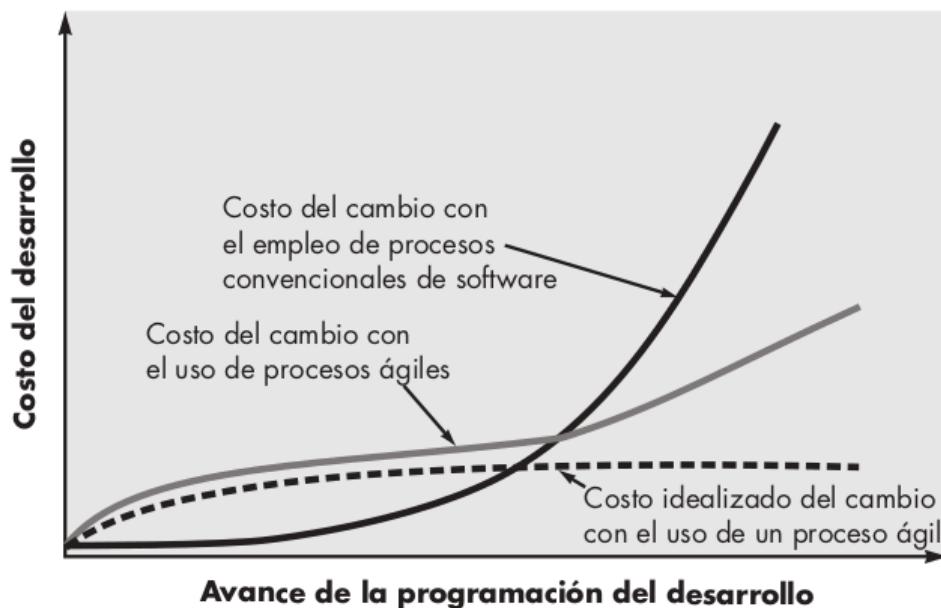
Martin Fowler define al enfoque ágil como **un compromiso entre nada de proceso y demasiado proceso**.

Desde el punto de vista de **Jacobson** **la ubicuidad del cambio es el motor principal de la agilidad**. Los ingenieros de software deben ir rápido si han de adaptarse a los cambios.

Es importante entender que los **métodos ágiles** son:

- **Adaptables** en lugar de predictivos.
- **Orientados a la gente** en lugar de orientados al proceso.

Otra de las características del agilismo, es que **hay un aplanamiento en el costo de la curva de cambio**, lo que permite que el equipo de software haga cambios en una fase tardía de un proyecto de software sin que haya un efecto notable en el costo y en el tiempo



El proceso ágil reduce el costo del cambio porque el software se entrega a incrementos y de esta forma el cambio se controla mejor

Ejemplos de frameworks ágiles:

- FDD
- Crystal
- XP
- Scrum
- ATDD

Valores de los equipos ágiles:

- Planificación continua, multi-level
- Facultados, auto-organizados, equipos completos
- Entregas frecuentes, iterativas y priorizadas
- Prácticas de ingeniería disciplinadas
- Integración continua
- Testing concurrent

Requerimientos ágiles:

Lo más difícil del software es decidir qué software queremos construir, ya que las cuestiones tecnológicas no son el impedimento, sino que los proyectos de software fracasan debido a que no se cumplen con las expectativas del cliente: lo que se construye no es lo que el usuario quería o necesitaba. Esto se viene advirtiendo desde la crisis de software.

Vamos a identificar las siguientes **características**:

- Usan el **valor** para construir el producto correcto. Nosotros como ingenieros en sistemas somos profesionales de soporte, esto quiere decir que ayudamos a otras profesiones a **obtener algo valioso** del sistema que están utilizando

El valor lo asociamos con la utilidad, beneficio o satisfacción que les ofreces a los usuarios finales, por cada funcionalidad completa que les entregas - Pablo Lischinsky

- Usan las **historias y modelos** para mostrar qué construir

Es importante ir construyendo el producto en conjunto con el cliente, los stakeholders tienen una mirada interesante y hay que tenerlos en cuenta

- *Técnicos y no técnicos trabajando juntos*

- Determinar que es *solo lo suficiente*. A lo que vamos con esto, es que los requerimientos los vamos a ir descubriendo de a poco. Osea que hay que empezar con lo mínimo necesario para empezar a girar la rueda (de acá la importancia del MVP)

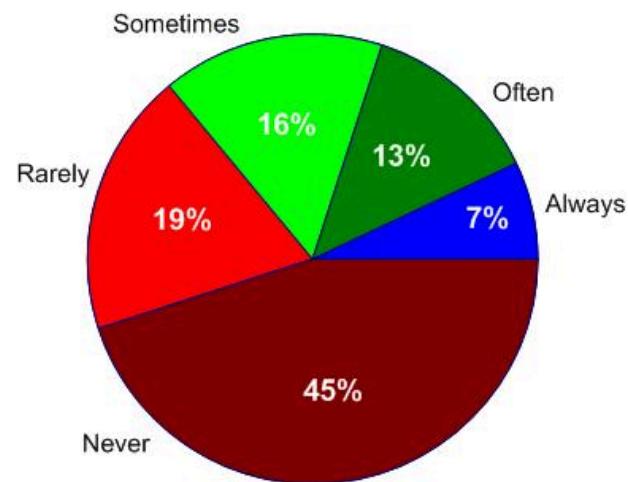
¿Y por qué queríamos cambiar el enfoque tradicional?

BRUF = Big Requirements Up front → Osea cebarte desde el principio y poner muchos requerimientos como enfermito

Esto es típico de modelos como por ejemplo el de **cascada**:

- **Requisitos completos al inicio:** Se espera que todos los requisitos del sistema estén completamente definidos y documentados al inicio del proyecto.
- **Documentación exhaustiva:** Se produce una gran cantidad de documentación de requisitos antes de que se escriba una sola línea de código.

Average percentage of delivered functionality actually used when a serial approach to requirements elicitation and documentation is taken on a "successful" information technology project.



Source: Chaos Report v3, Standish Group.

Copyright 2005-2006 Scott W. Ambler

Fíjate que es un error que le pasa hasta los productos que han sido exitosos, han definido 800 mil requerimientos pero solamente un 7% de ellos son realmente utilizados. Encima hay algunos que ni se los usa.

Just In Time:

En el contexto de los requerimientos ágiles, el concepto de **Just In Time (JIT)** se refiere a la práctica de definir y detallar los requerimientos justo en el momento en que se necesitan para su implementación. En lugar de intentar capturar y documentar todos los detalles al inicio del proyecto, en un enfoque ágil se proporcionan detalles adicionales de los requerimientos a medida que el equipo de desarrollo se aproxima a la fase en la que estos requerimientos serán implementados.

Al detallar los requerimientos solo cuando es necesario, se **evita el desperdicio de tiempo y esfuerzo** en especificar características que podrían cambiar o incluso ser eliminadas antes de que se implementen.

Tipos de Requerimientos:

- **Requerimientos de negocio:** Disminuir un X % de tiempo invertido en procesos manuales relacionados con atención al cliente
- **Requerimientos de usuario:** Realizar consultas en línea del estado de cuenta de los clientes

Para entenderlo de manera más sencilla, el requerimiento de negocio es algo que le otorga valor al negocio justamente, mientras que el requerimiento del usuario es el **cómo voy a hacer para llevar a cabo este valor**

- **Requerimiento funcional:**

- Generar reporte de saldos de cuenta.
- Recibir notificaciones por email

- **Requerimiento no funcional:**
 - Formato del reporte PDF.
 - Cumplir con niveles de seguridad para credenciales de usuarios según la ley bancaria 9999 XX
- **Requerimientos de implementación:** Servidores en la nube

Dominio del problema: Abarca todo lo relacionado con el contexto en el cual surge la necesidad del software. Aquí es donde se definen los requerimientos en términos de lo que el negocio y los usuarios necesitan.

- Req. de negocio
- Req. de usuario

Dominio de la solución: Abarca las decisiones sobre cómo se implementará el software para cumplir con los requerimientos definidos en el dominio del problema.

- Req. del software

¿Cómo estamos situados nosotros en este esquema?

Las user stories son en esencia requerimientos de usuarios, alineado a un requerimiento de negocio

Ahora bien, las user stories **no sirven para especificar requerimientos de software**, esto lo hacemos partir de casos de uso

Nota:

- Los requerimientos de usuario detallan que es lo que quiere hacer el usuario
 - *Quiero registrar mis compras del super*
- Los requerimientos funcionales detallan cómo el sistema realiza esta funcionalidad
 - *El sistema debe permitir al usuario ingresar datos de una compra, incluyendo: fecha, lista de productos, cantidades, precios y total de la compra*

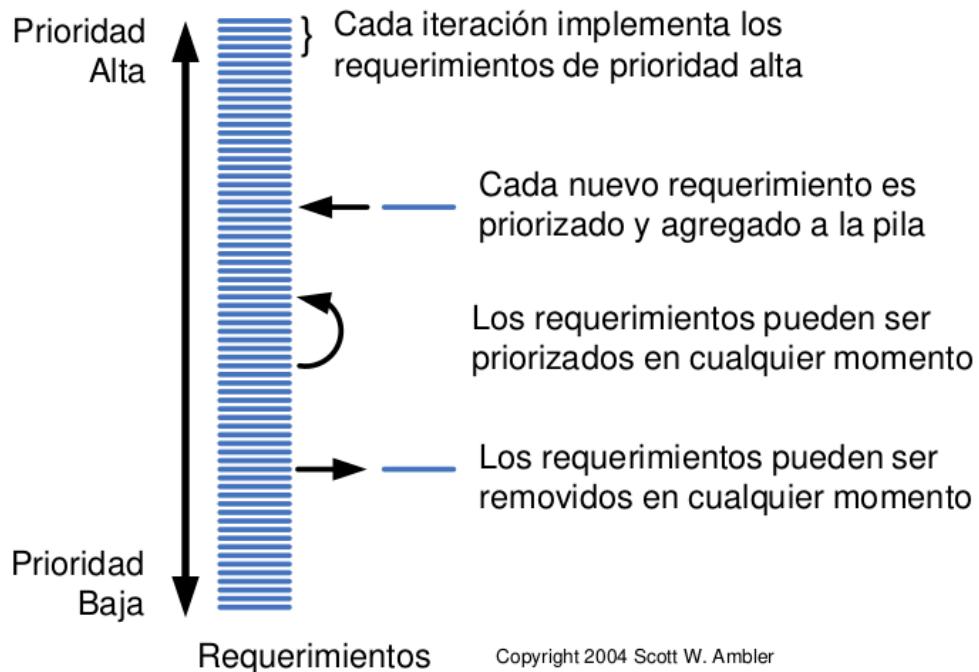
Gestión ágil de requerimientos:

Principalmente lo hacemos a través del **producto backlog**.

Este es un **listado dinámico y priorizado** de todas las funcionalidades, características, mejoras, correcciones, y demás elementos que el producto final debería tener para cumplir con las expectativas del cliente o usuario.

El **product owner** es el encargado de crear y mantener el product backlog. Es la persona que representa los intereses del negocio al cual le estamos desarrollando y producto y se asegura de maximizar el valor de el producto

Cada ítem en el Product Backlog se denomina un **Product Backlog Item (PBI)** y debe estar descrito de manera clara para que el equipo de desarrollo entienda lo que se necesita.



En **contraposición** con el enfoque tradicional, donde teníamos un cliente que no estaba dispuesto a comprometerse y dejaba la priorización de los requerimientos al

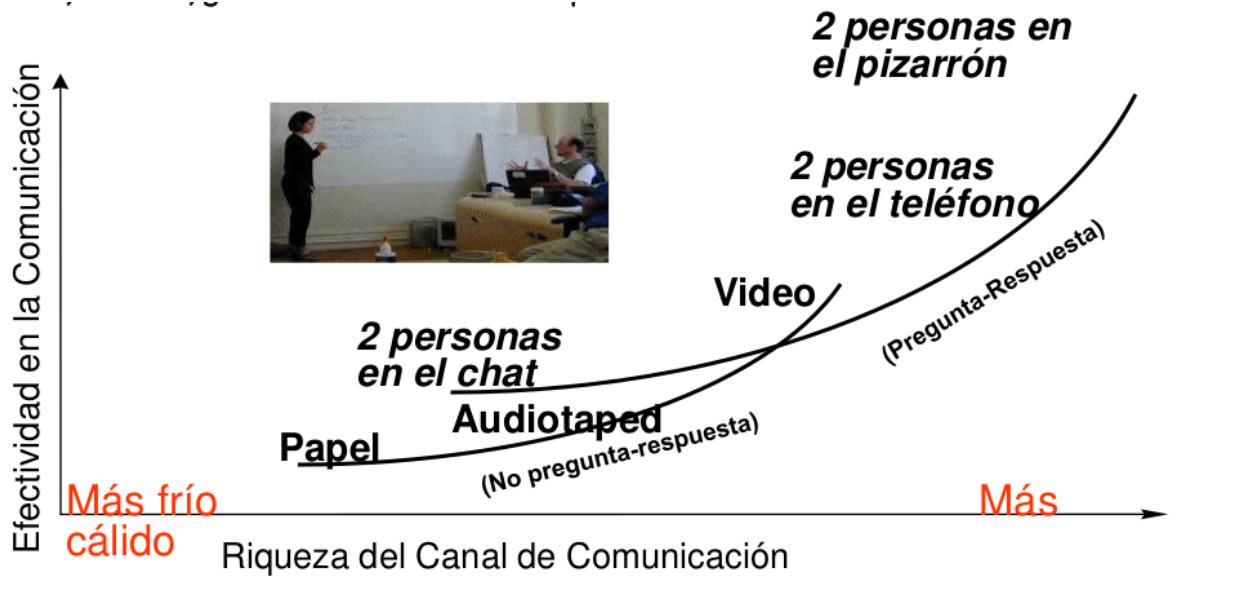
equipo que hace el software, se terminaba construyendo un producto que no dejaba satisfecho al cliente. En ágil es el cliente el encargado de esta priorización, lo que lleva a que se construya el producto deseado.

Con esta situación se permite la **compensación** de no tener documentos de requerimientos formales escritos, detallando todos los puntos.

Los principios ágiles relacionados a la gestión ágil de requerimientos son:

1. La prioridad es satisfacer al cliente (entregando software funcionando) a través de releases tempranos y frecuentes
 2. Recibir cambios de requerimientos, aún en etapas finales
 4. Técnicos y no técnicos trabajando juntos todos el proyecto
 6. El medio de comunicación por excelencia es el cara a cara
 11. Las mejores arquitecturas, diseños y requerimientos emergen de equipos autoorganizado
-

Ahora lo que queremos analizar es la forma más eficiente a partir de la cuál podemos hacer que la información fluya



Lo **cara a cara** es lo más efectivo que se puede hacer, establecemos una conexión más calidad con la persona

Momento de captura de requerimientos:

En el **enfoque tradicional**, los requerimientos son definidos al principio del proyecto. El caso particular del PNUD, que es un proceso definido con un modelo de proceso iterativo e incremental, el flujo de trabajo de requerimientos tiene un gran peso en la fase inicial y de elaboración.

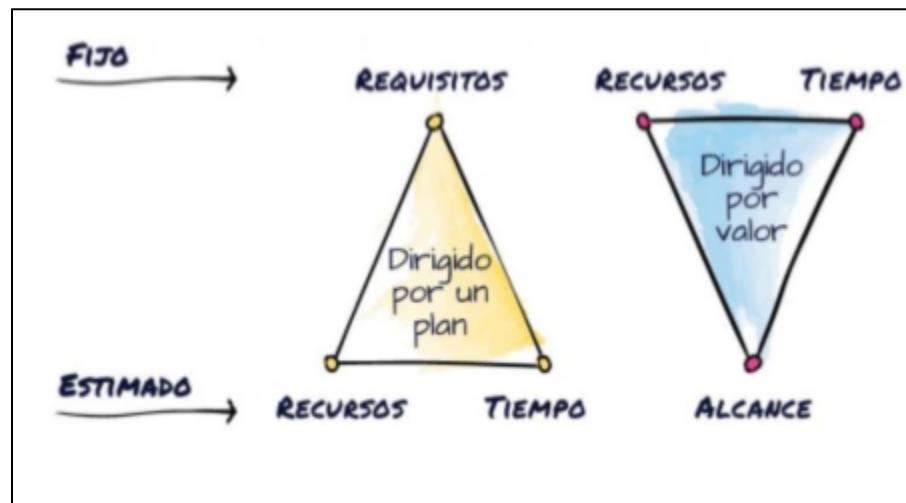
En el **enfoque ágil**, los requerimientos son relevados continuamente durante todo el proyecto. El inicio de la construcción del software se da en un contexto en el cual los requerimientos no están completamente definidos, sin embargo, se tiene una visión

Contenedor y persistencia de los requerimientos:

En el enfoque **tradicional**, la ERS contiene los requerimientos a lo largo del ciclo de vida del producto todo el tiempo. En caso de que surjan cambios, es necesario actualizar la ERS. En definitiva, es el documento de más alto nivel que define las características del sistema.

En el **enfoque ágil**, se utiliza al Product Backlog como un contenedor transitorio de los requerimientos, ya que una vez implementados estos se “persisten” en el producto funcionando. El detalle de valor queda persistente en los casos de prueba que se construyen en el Testing.

Tradicional vs Ágil:



Esto que estás viendo, es lo que se conoce como **triángulo de hierro**.

Fíjate que siempre que trabajamos en un contexto de proceso **tradicional**, suelo dejar fijos los requisito o el alcance y en función de eso defino los recursos y el tiempo

En el **agilismo** lo que hacemos es lo siguiente:

- Dejamos **fijo el tiempo**, con iteraciones de duración fija (Scrum las llama sprint y como máximo pueden durar un mes)
- Dejamos **fijo los recursos**, es decir que tenemos un equipo de trabajo con una determinada capacidad
- Esto nos deja como variable el alcance → **¿Cuánto puedo construir si tengo este equipo y este tiempo?**

A continuación se muestra un cuadro con las diferencias más en detalle:

	Tradicional	Ágil
Prioridad	Cumplir el plan	Entregar valor
Enfoque	Ejecucion ¿Cómo?	Estrategia (Por y para que)
Definicion	Detallados y cerrados. Descubrimientos al inicio	Esbozados y evolutivos (Descubrimiento progresivo)
Equipo	Hay una jerarquía marcada con un líder con roles bien definidos	Equipo multidisciplinario y autoorganizado
Herramientas	Entrevistas, observación y formularios	Principalmente prototipado
Documentacion	Alto nivel de detalle (Matriz de rastreabilidad para los requerimientos)	Historias de usuario, Mapeo de historias
Producto	Definidos en alcance	Identificados progresivamente
Proceso	Estables, adversos al cambio	Incertidumbre, abierto al cambio

User stories:

Concepto:

Addison-Wesley

Una historia de usuario describe una **funcionalidad** que será valiosa para un usuario o comprador de un sistema o software.

Son básicamente **técnicas** que nosotros utilizamos para poder trabajar con los requerimientos ágiles

Se las llama stories porque se supone que usted cuenta una historia. Lo que se escribe en la tarjeta no es importante, lo que habla usted si

Son un **token** para una conversación. → *Como usuario quiero iniciar sesión en google*
→ Esto es un puntapié para que al leer esta user story se desate una **conversación** en donde se toquen temas como

- ¿Qué datos necesito de google?
- ¿Qué país vamos a usar?
- Etc

Con esto logramos reducir un montón la documentación y la burocracia

Son **escritas** por el Product Owner

Las user stories son **multipropósitos**, esto significa que son:

- Una necesidad del usuario
- Una descripción del producto
- Un ítem de planificación
- Token para una conversación
- Mecanismo para diferir una conversación

Las **partes** de una user story son:

- **Tarjeta**: Es la parte visible, donde escribimos la historia de usuario. La tarjeta está compuesta por:
 - Frase verbal: Suele agregarse una frase resumen de los que se trata la user story a modo de título
 - Frase principal (Nombre de rol - Actividad - Valor de negocio)
 - Criterios de aceptación: Son los límites para la user story. Define qué es lo que el product owner quiere para que la use sea aceptada

Supongamos que quiero registrar un usuario:

- *El sistema debe permitir registrarse con un correo único.*
- *Se debe mostrar un mensaje de confirmación al registrarse.*
- *El correo debe ser validado para asegurar que es válido.*
- **Pruebas de aceptación:** Son las pruebas que se entiende que va a hacer el Product Owner y los usuarios cuando se entregue la funcionalidad lista. (Entonces en teoría esto va en la parte de la confirmación pero bueno se nombra aca q yo)

Se especifican entonces si fallan o si pasan

La forma de expresar una user story es la siguiente:

Como <nombre del rol>, yo puedo <actividad> de forma tal que <valor de negocio que recibo>

Por ejemplo:

Como conductor quiero buscar un destino a partir de una calle y altura para poder llegar al lugar deseado sin perderme

- **Conversación:** La conversación se considera la parte más importante de las user stories. Esta no queda guardada en ningún lado, porque según los principios ágiles el mejor medio de comunicación es cara a cara

La conversación reemplaza a la documentación exhaustiva

- **Confirmación:** Comprende la definición de las pruebas de aceptación las cuales son necesarias para que el PO me acepte la user story como válida.

User stories: Porciones Verticales

Para entender esto, primero entendamos que nosotros desde DSI venimos trabajando con el concepto de **arquitectura**.

Lo que tienen las user stories es que hacen un corte vertical a la arquitectura, es decir que tocan temas que van desde la interfaz de usuario, hasta la base de datos, pasando entre medio por la capa lógica de negocios.



Fíjate que en el supuesto caso de que las user stories cortaran a la arquitectura de manera **horizontal**, las mismas serían incapaces de dar valor al cliente.

Ejemplo god:

Imagínate que estás construyendo una aplicación de comercio electrónico:

- **Porción Horizontal:** Sería una tarea que solo se enfoca en una capa del sistema, como "crear la tabla de productos en la base de datos" o "diseñar la interfaz para agregar productos al carrito". Estas tareas son importantes, pero no aportan valor directo al usuario final por sí solas, ya que no son funcionalidades completas.
- **Porción Vertical:** Una user story que describe una porción vertical sería algo como **"como usuario, quiero agregar un producto al carrito para comprarlo más tarde"**. Esta historia implica trabajo en la base de datos (para almacenar el

carrito), en la lógica de negocio (para manejar el proceso de agregar al carrito) y en la interfaz de usuario (para que el usuario pueda realizar la acción). Es una funcionalidad completa que, cuando está terminada, proporciona valor al usuario.

Usuarios representantes (Proxies):

Cuando desarrollamos software, no siempre podemos hablar directamente con los usuarios finales (ej: si son miles, o si son clientes externos). Por eso, en la planificación, ciertas personas actúan como "representantes" (proxies) de esos usuarios. Su rol es transmitir las necesidades, expectativas y contextos reales de quienes usarán el sistema.

Acá nos encontramos con:

- Gerentes de usuario
- Gerentes de desarrollo
- Alguien del grupo de marketing
- Vendedores
- Expertos del dominio
- Clientes
- Capacitadores y personal de soporte

El product owner hace referencia a roles de negocio, **no a roles técnicos**. Osea estaría mal poner:

- Analista en diseño
- Programador C++
- Dev ops no se que pinchila

Criterios de aceptación:

Información concreta que tiene que servirnos a nosotros para saber si lo que implementamos es correcto o no.

Si el product owner nos va a aceptar esta característica implementada o no

- Definen **límites** para una user story (US)

- Ayudan a que el equipo tenga una visión compartida de la US
- Ayudan a desarrolladores y testers a derivar las pruebas
- Ayudan a los desarrolladores a saber cuando parar de agregar funcionalidad en una US

¿Cuales son los criterios de aceptación es bueno?

- Definen una intención, no una solución
- Son independientes de la implementación
- Relativamente de alto nivel, no es necesario que se escriba cada detalle

Pruebas de aceptación:

Son declaraciones de intención de que hay que probar

Estan pruebas de aceptacion nacen de las confirmaciones; de las conversaciones que se dan

Modelo INVEST:

El modelo INVEST es un acrónimo utilizado en el contexto de las historias de usuario para guiar la creación y evaluación de historias bien formadas y efectivas dentro de metodologías ágiles.

- **Independiente:** Una historia de usuario debe ser independiente de las otras historias de usuario en la medida de lo posible

Las **user stories** no deben ser dependientes entre sí porque las dependencias complican la planificación, priorización y estimación del trabajo en un proyecto ágil.

Por ejemplo:

- **Historia A:** "Como usuario, quiero poder crear una cuenta."

- **Historia B:** "Como usuario, quiero poder enviar mensajes después de crear una cuenta."

En este caso, la Historia B depende completamente de que la Historia A esté terminada. Esto significa que la Historia B no puede ser desarrollada o probada de forma independiente. Si por alguna razón hay un retraso en la Historia A, la Historia B también se verá afectada, lo que podría causar problemas en la planificación del sprint.

- **Negociable:** Las historias de usuario deben ser flexibles y negociables, no una especificación rígida. Deben servir como punto de partida para el product owner y el equipo de desarrollo
- **Valiosa:** Cada historia de usuario debe agregar valor al usuario o al negocio. Si una historia no tiene un valor claro, probablemente no debería ser priorizada.
- **Estimable:** Una historia de usuario debe ser lo suficientemente clara y pequeña como para que el equipo pueda estimar el esfuerzo necesario para completarla. Si una historia es demasiado grande o vaga, es difícil de estimar.
- **Pequeña:** Una historia de usuario debe ser lo suficientemente pequeña para ser completada en un solo sprint o dentro de un ciclo iterativo breve
- **Testable:** Una historia de usuario debe poder ser probada para verificar si se ha cumplido correctamente

Por ejemplo: *Como usuario, quiero que la aplicación sea fácil de usar para que tenga una buena experiencia.*

Definition of Ready:

Es una medida de calidad que determina si la User Story está en condiciones de entrar a una iteración de desarrollo.

Para que la US pueda ser implementada, mínimamente debe satisfacer el INVEST Model.

Entre el equipo de desarrollo y el PO pueden definir **características o condiciones extras al INVEST** para que una US se considere lista para entrar a la iteración de desarrollo.

Definition of Done:

Nos define si la historia **está decentemente terminada** para poder presentarla al Product Owner al final de la iteración, para que este pueda decidir si la misma pasa a **producción** o no.

Para que la user sea considerada *done*, debe cumplir con una serie de **ítems de una checklist**:

Definición de Hecho (DONE)	
<input type="checkbox"/>	Diseño revisado
<input type="checkbox"/>	Código Completo
<input type="checkbox"/>	Código refactorizado
<input type="checkbox"/>	Código con formato estándar
<input type="checkbox"/>	Código Comentado
<input type="checkbox"/>	Código en el repositorio
<input type="checkbox"/>	Código Inspeccionado
<input type="checkbox"/>	Documentación de Usuario actualizada
<input type="checkbox"/>	Probado
<input type="checkbox"/>	Prueba de unidad hecha
<input type="checkbox"/>	Prueba de integración hecha
<input type="checkbox"/>	Prueba de sistema hecha
<input type="checkbox"/>	Cero defectos conocidos
<input type="checkbox"/>	Prueba de Aceptación realizada
<input type="checkbox"/>	En los servidores de producción

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. Es una característica más del producto a la cual el equipo le tiene que dedicar tiempo para:

- Se clasifican en: técnicas y funcionales
- Pueden utilizarse para:
 - Inversión básica para familiarizar al equipo con una nueva tecnología o dominio
 - Analizar un comportamiento de una historia compleja y poder así dividirla en piezas manejables
 - Ganar confianza frente a riesgos tecnológicos, investigando o prototipando para ganar confianza
 - Frente a riesgos funcionales, donde no está claro como el sistema debe resolver la interacción con el usuario para alcanzar el beneficio esperado

Spike técnica:

Utilizadas para evaluar diferentes enfoques técnicos y tecnológicos en el dominio de la solución. Están asociados a la implementación de la funcionalidad. Nueva tecnología.

Cualquier situación en la que el equipo necesite una comprensión más fiable antes de comprometerse a una nueva funcionalidad en un tiempo fijo. Dependen de los técnicos (nosotros).

Spike funcional:

Utilizada cuando hay cierta incertidumbre respecto de cómo el usuario interactúa con el sistema, usualmente son

mejor evaluadas con **prototipos** para obtener retroalimentación de los usuarios involucrados.

Dependen de definiciones del negocio, del PO.

Lineamientos para los spikes:

- Estimables, demostrables y aceptables
- La excepción, no la regla:
 - Toda historia tiene incertidumbre y riesgos
 - El objetivo del equipo es aprender a aceptar y resolver cierta incertidumbre en cada iteración
 - Los spikes deben dejarse para incógnitas más críticas y grandes
 - Utilizar spikes como ultima opción
- En general, se recomienda implementar la spike en una iteración separada de las historias resultantes

Épicas:

Una Épica es un conjunto grande de funcionalidades o historias de usuario (User Stories) que están relacionadas entre sí y que, en conjunto, aportan un valor significativo al producto

Estimaciones de Software:

Introducción:

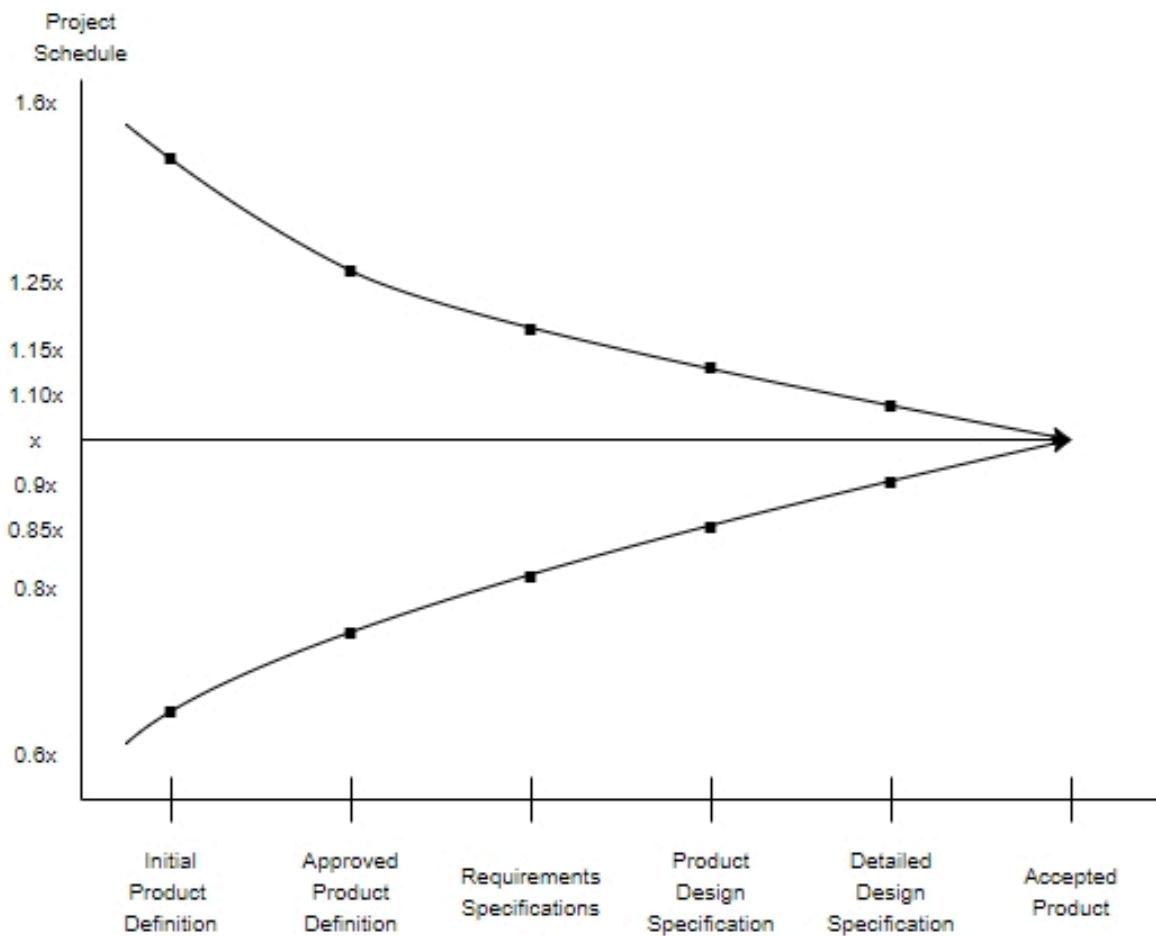
Concepto:

Estimar es básicamente un **proceso probabilístico**, el cual consiste en predecir el valor que va a asumir algún aspecto del proyecto(trabajo, esfuerzo, tiempo, etc) que será necesario realizar evidentemente con un cierto grado de **incertidumbre**.

Algunas **consideraciones** a tener en cuenta son:

- Por definición **una estimación no es precisa**
- Estimar no es planear, así como planear no es estimar
- Las estimaciones son la base de los planes, pero los planes puede diferir de estas estimaciones
- A mayor diferencia entre lo estimado y lo planeado **mayor riesgo**
- Las **estimaciones no son compromisos**

El cono de la incertidumbre:



El cono de la incertidumbre es una forma gráfica de expresar que, **a medida que un proyecto avanza, la precisión de nuestras estimaciones aumenta.**

Al arrancar, no obstante, la **variabilidad** de las estimaciones siempre será alta, puesto que sabemos bastante poco sobre el producto en cuestión. Y lo será incluso si invertimos un gran esfuerzo en realizar las estimaciones. Conforme avanzamos en el trabajo de desarrollo y obtenemos más y más información, aprendiendo en cada Sprint, seremos capaces de acotar dicha variabilidad.

De acuerdo con estos valores, al comienzo de un proyecto la estimación típicamente varía entre un 60% y un 160%. Es decir, un proyecto estimado en 20 semanas puede tardar entre 12 y 32 semanas.

De cualquier forma → ***Es mejor estimar que no estimar***

¿De dónde vienen los errores de estimación?:

- **Actividades omitidas:** Solemos estimar solamente el tiempo de programación (30 - 35 % del total) y no consideramos otras actividades.

Por ejemplo, supongamos que queremos estimar el tiempo que dura realizar el softwar

- 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)
- **El proceso mismo de estimación:** Es decir, que de por si el proceso de estimación siempre va a tener un sesgo obviamente
- **No tenemos claro el proceso a utilizar:** Por ejemplo si tenés un nivel de madurez muy bajo de acuerdo al CMMI, tus estimaciones no van a ser del todo correctas
- **Falta información:** Entonces cuando estimamos, estamos asumiendo ciertas cosas pero que en realidad no las sabemos.

Métodos utilizados para la estimación tradicional:

- **Basados en la experiencia:**
 - Datos históricos
 - Juicio experto

- Puro
 - Delphi
 - Analogía
- **Demás métodos:**
- Basados exclusivamente en los recursos
 - Método basado exclusivamente en el mercado
 - Basados en los componentes del producto o en el proceso de desarrollo
 - Métodos algorítmicos

Datos históricos:

La principal razón para utilizar datos históricos es que básicamente mejora la precisión de las estimaciones que vos hagas

Consiste en recolectar datos básicos de otros proyectos para ir generando una base de conocimientos que sean de utilidad para futuras estimaciones, con esto se busca una alternativa en la cual el conocimiento de cada individuo se transfiere a la organización y no estamos atados a una única persona. Se deben utilizar para no tener problemas entre desarrolladores y clientes

Fijate que esto va en contra de lo que dice el agilismo → La experiencia no es extrapolable

Otra cosa a considerar es que depende mucho de donde vos sacas los datos históricos ya que estoy evidentemente influye más o menos en la precisión de tus estimaciones:

- Datos históricos de organizaciones relacionadas a la tuya
- Datos históricos de tu misma organización
- Datos históricos de proyectos anteriores

Juicio experto → Puro:

- Un experto (o gurú) estudia las **especificaciones** y hace su estimación
- Se basa fundamentalmente en los **conocimientos del experto**
- Si desaparece el experto, la empresa deja de estimar
- Es el enfoque de estimaciones **más utilizado en la práctica**
- Acerca del 75% de organizaciones de software la utilizan

¿Cómo estructurar el juicio del experto?

- Es importante tener tareas de **granularidad** aceptable
- Utilizar el **método de optimista, pesimista y habitual** cuya fórmula es igual a **(o + 4h + p) / 6**
- Importante utilizar **checklist** y un criterio definido para asegurar cobertura

Juicio experto → Delphi:

Es similar al anterior, pero este juicio se realiza de manera **grupal**, y esto se debe a que las estimaciones en grupo suelen ser mejor que las individuales

Entonces, algunas **características** del método son:

- Se dan especificaciones a un grupo de expertos
- Se les reúne para que discutan tanto el producto como la estimación
- Remiten sus estimaciones individuales al coordinador
- Cada estimador recibe información sobre su estimación y las estimaciones ajenas aunque estas últimas de manera anónima
- Se reúnen de nuevo para discutir las estimaciones

- Cada uno revisa su propia estimación y la envía al coordinador
- Se repite el proceso hasta que la estimación converge de forma razonable

Este método lo hacemos en **iteraciones**. Esto hasta que entre todos nos pongamos de acuerdo más o menos en alguna estimación.

Analogía:

Meles:

Es como un ajuste de los datos históricos, básicamente agarras la base de datos está que generas y se realiza una **abstracción** de los que son parecidos a mi proyecto, por ejemplo

- Agarró los que usan python como backend
- Los que tenían 3 desarrolladores front end
- Etc

Básicamente la idea general es que **vos podes crear una estimación para un nuevo proyecto que vas a hacer comparándolo con un proyecto pasado y similar**

A continuación pongo un FRAGMENTO del ejemplo del libro que es clave:

Table 11-1 Detailed Size Comparison Between AccSellerator 1.0 and Triad 1.0

Subsystem	Actual Size of AccSellerator 1.0	Estimated Size of Triad 1.0	Multiplication Factor
Database	10 tables	14 tables	1.4
User interface	14 Web pages	19 Web pages	1.4
Graphs and reports	10 graphs + 8 reports	14 graphs + 16 reports	1.7
Foundation classes	15 classes	15 classes	1.0
Business rules	???	???	1.5

Este **factor multiplicativo** lo multiplicas con las LOC del proyecto viejo y te da un estimativo de las LOC del proyecto nuevo

Table 11-2 Computing Size of Triad 1.0 Based on Comparison to AccSellerator 1.0

Subsystem	Code Size of AccSellerator 1.0	Multiplication Factor	Estimated Code Size of Triad 1.0
Database	5,000	1.4	7,000
User interface	14,000	1.4	19,600
Graphs and reports	9,000	1.7	15,300
Foundation classes	4,500	1.0	4,500
Business rules	11,000	1.5	16,500
TOTAL	43,500	-	62,900

También se puede **estimar el effort** de la siguiente manera

Table 11-3 Final Computation of Effort for Triad 1.0

Term	Value
Size of Triad 1.0	62,900 LOC
Size of AccSellerator 1.0	÷ 43,500 LOC
Size ratio	= 1.45
Effort for AccSellerator 1.0	× 30 staff months
Estimated effort for Triad 1.0	= 44 staff months

Nota:

Puede parecer un poco complicado entender la medida de *44 staff months*, pero es facil.

Primero hay que partir de un valor base que es **1 staff month**, o lo mismo que la cantidad de horas que trabaja un empleado al mes, entonces con esa relacion podemos calcular lo que sea

1 staff months = 160 horas

44 staff months = $160 \times 44 = 7040$ horas que necesito para terminar mi proyecto

Basados exclusivamente en recursos:

Consiste en analizar el personal que se tiene, el tiempo, el presupuesto y en base a ese punto de partida se estima que es lo que se puede hacer.

En la realización: “El trabajo se expande hasta consumir todos los recursos disponibles, independientemente si se alcanzan o no los objetivos.”.

Basados exclusivamente en el mercado:

Acá se pone el foco en el cliente y se realizan las estimaciones en función de su capacidad de pago. En este método no se evalúa al producto que se está intentando vender, sino a quien se lo vende. A veces se cobra menos para que nos elijan antes que a la competencia.

Basados en los componentes del producto:

Este es un enfoque en el cual el proyecto se descompone en sus partes constituyentes o módulos funcionales, y se estima el esfuerzo y tiempo requerido para cada uno de ellos de forma individual. Luego, estas estimaciones se agregan para obtener una visión completa del esfuerzo total necesario para el proyecto.

Acá es posible trabajar con 2 enfoques:

- **Bottom - up:** Se descompone el producto en unidades lo más pequeñas posibles. Se estima cada unidad y luego se hace una estimación total
- **Top - down:** Se descompone el producto en grandes bloques y luego se estima cada uno de los componentes

Métodos algorítmicos:

Se ejecutan algoritmos que tienen como parámetros de entrada medidas cuantitativas de tamaño, complejidad y conocimiento de dominio.

Cada variable tendrá asociada un **factor multiplicativo** y en función de esto se obtiene como resultado un valor de esfuerzo. Es importante destacar que, a pesar de ser un cálculo matemático, puede no ser certero igual.

Un ejemplo de modelo algorítmico es el **modelo COCOMO** (Constructive - Cost - Model) que significa modelo constructivo de costos. Este modelo es útil para estimar:

- Esfuerzo del proyecto (eh horas trabajo)
- Duración del proyecto
- Costo del proyecto

Para calcular esto se usa principalmente **datos** como el tamaño de software medido en KLOC (miles de líneas de código)

La **fórmula** es la siguiente:

$$\text{Esfuerzo (en personas-mes)} = a \times (\text{Tamaño})^b$$

- **Tamaño:** miles de líneas de código (KLOC).
- **a y b:** constantes que dependen del tipo de proyecto.
- **Resultado:** el esfuerzo estimado en **meses-hombre**.

Ejemplo básico de COCOMO:

Supongamos que vas a desarrollar una aplicación de 20 KLOC (20,000 líneas de código) y tu proyecto es semi-acoplado:

$$Esfuerzo = 3.0 \cdot (20)^{1.12} \approx 90 \text{ PM}$$

Esto significa que se requerirían 90 personas-mes. Si tienes 5 personas trabajando, el tiempo sería:

$$Duración \approx \frac{90}{5} = 18 \text{ meses}$$

No Estimate (Estimaciones en ambientes ágiles):

Diferencias entre estimaciones ágiles y tradicionales:

- Con respecto al **triángulo de hierro**:

En Agile se tiene fijo el tiempo (lo que dura la iteración) y los recursos afectados para trabajar, dejando variable el alcance.

Por lo tanto, al estimar en estos ambientes se responde a cuánto se puede comprometer el equipo, qué cantidad de las características de producto se puede construir en la iteración. Se entiende, además, que en estos ambientes el producto no está completamente definido, sino que se tiene una visión de este.

En los **ambientes tradicionales**, el alcance es lo que está fijo y el producto se encuentra definido. A partir de esto, se deriva el resto de las variables. Por esta razón en los procesos definidos se estima primero el tamaño, ya que es la base para estimar el resto.

- Con respecto a la **precisión**:

Las estimaciones en los ambientes tradicionales buscan ser más precisas, debido a que son las bases para luego planificar. Dado que la precisión es realmente cara, la precisión en los ambientes ágiles y en correspondencia con la filosofía Lean, es considerada un desperdicio.

- **Principal diferencia:**

Las estimaciones ágiles son relativas, a diferencia de las estimaciones en ambientes tradicionales que son absolutas.

En Agile, las características del producto son estimadas utilizando una medida de tamaño relativa dado que las

Las personas no somos buenas realizando estimaciones absolutas, sino que es más fácil y rápido comparar medidas. Por otra parte, la práctica de estimar mediante comparaciones permite obtener una mejor dinámica grupal y pensamiento de equipo por sobre el individuo.

- **Quién estima:**

En las metodologías de estimación tradicional, las estimaciones las realiza el **Líder del proyecto** o en ocasiones las realiza un **experto**, por lo que su estimación no es representativa de la cantidad de trabajo que requiere una persona del equipo para terminar una funcionalidad del producto. Esta es una de las razones principales de los desvíos en los proyectos de software tradicional.

En los ambientes ágiles y en concordancia con los valores y principios explicitados en el manifiesto ágil, es el equipo el que realiza la estimación. Lo óptimo es que cada uno estime su propio trabajo y de esta manera, es posible disminuir la probabilidad de desviaciones.

- **Confusión con la planificación:**

En los ambientes **tradicionales** existe un gran error que es asumir las estimaciones como compromisos, como si fueran planificaciones. Esto implica que las estimaciones se transforman en promesas.

Mientras que en **agile**, asumen que las estimaciones están inmersas en el campo de las probabilidades y por lo tanto no implican un compromiso. Una falla en la estimación no implica extender el tiempo, ya que en ágiles trabajamos con el tiempo fijo, por lo tanto, lo que se ajusta es el alcance.

- **El momento para estimar:**

En el enfoque tradicional se estima al comienzo del proyecto, luego de la especificación de requerimientos y luego del diseño (debido que ahí es cuando puedo saber cuál es el tamaño y de ahí derivó todas las demás estimaciones). Realmente se estima cada vez

que se modifica el alcance, ya que es la variable que se mantiene fija y de la cual se deriva el resto.

Las estimaciones a lo largo del proyecto se van haciendo cada vez más certeras. Al comienzo, pueden diferir hasta un 400%, esto se debe a:

- Al comienzo del proyecto contamos con muy poca información
- Las estimaciones son demasiado optimistas

En ágil se estima al comienzo de cada sprint

Concepto:

Las estimaciones ágiles, a diferencia de las estimaciones absolutas de la gestión tradicional, tienden a incorporar un método de estimación relativa basado en **obtener la estimación a partir de la comparación**.

En las estimaciones ágiles, **sólo se estima el tamaño de una user**, con una serie de **parámetros** (Complejidad, Esfuerzo, Incertidumbre)

Fíjate que **aspectos como el tiempo no se estiman** ya que en el enfoque ágil lo que se mantiene fijo es el tiempo (trabajamos en timeboxes)

Esto se debe a que:

- Las personas no saben estimar en términos absolutos
- **Somos buenos comparando cosas**
 - *¿Cuántas veces más complejo es esto?*
 - *¿Cuántas veces más difícil es esto?*
- Comparar es generalmente más rápido
- **Se obtiene una mejor dinámica grupal y pensamiento de equipo más que individual**
- Se emplea mejor el tiempo de análisis que las stories

Tamaño vs Esfuerzo

- “La palabra tamaño refiere a cuán grande o pequeño es algo”
- El tamaño es una medida de la cantidad de trabajo necesaria para producir una feature/story
- Nos da una idea de cuán compleja y grande es una user feature

El tamaño no es esfuerzo. Vos podés tener dos user story del mismo tamaño, pero el esfuerzo requerido por dos personas va a diferir según sus capacidades, habilidades, disciplinas, etc.

Esfuerzo vs Calendario:

Por el lado del esfuerzo, habíamos visto que este se correspondía con horas de trabajo lineales, mientras que el tamaño no tiene nada que ver con esto (lo vimos en el anterior ítem)

El esfuerzo no debe ser confundido con el calendario, si bien mi deadline podría ser hasta el viernes el esfuerzo que me requiere (es decir horas de trabajo lineales) podrían verse afectadas por una multitud de factores como (horas de ocio, baño, el jefe me cambia algo, etc)

Otra aclaración es que este esfuerzo depende específicamente de la persona que lo va a llevar a cabo, su capacitación, etc.

Ejemplo Meles: Vos vas a una empresa y preguntas, cual es el esfuerzo que se requiere para realizar tal caso de uso o tal requerimiento y te responden:

- Se requieren 2 días y medio

Bueno esto **está mal**, eso es calendario no esfuerzo.

Porque ponele que realmente necesitas 2 días y medio de trabajo, aun así vos en la empresa no estás todo el tiempo trabajando pueden pasar impedimentos o puede venir el jefe y te manda a hacer otra tarea, etc

Story Points:

Es una unidad de estimación que especifica la **complejidad, esfuerzo e incertidumbre**, propio del equipo respecto a una user en términos relativos **respecto a otra user**.

Story point la idea del **peso** de cada story y decide cuán grande o compleja es

La complejidad de una feature/story tiene a incrementarse exponencialmente

- **Complejidad:** que tan compleja es implementar esa US. Relacionada a la cantidad de partes y relaciones entre las mismas posee la US.
- **Esfuerzo:** Es importante acá que estime la persona que va a realizar el trabajo, debido a que el esfuerzo es distinto entre dos personas. Son las horas lineales que se requieren para terminar la user story
- **Incertidumbre:** Es cuánta información nos falta, se relaciona con el grado de entendimiento de la user.

Velocidad:

Esta es una de las métricas **más importantes** que se utiliza en agilismo sobre todo porque **mide el producto**.

- Recordemos uno de los principios ágiles que decía que la mejor métrica de progreso era que el producto se encuentre funcionando

Velocidad es una medida (métrica) del progreso de un equipo.

Para que la user story **entre en el cálculo** de la velocidad debe cumplir con la Definition of Done y el Product Owner la tiene que haber aceptado.

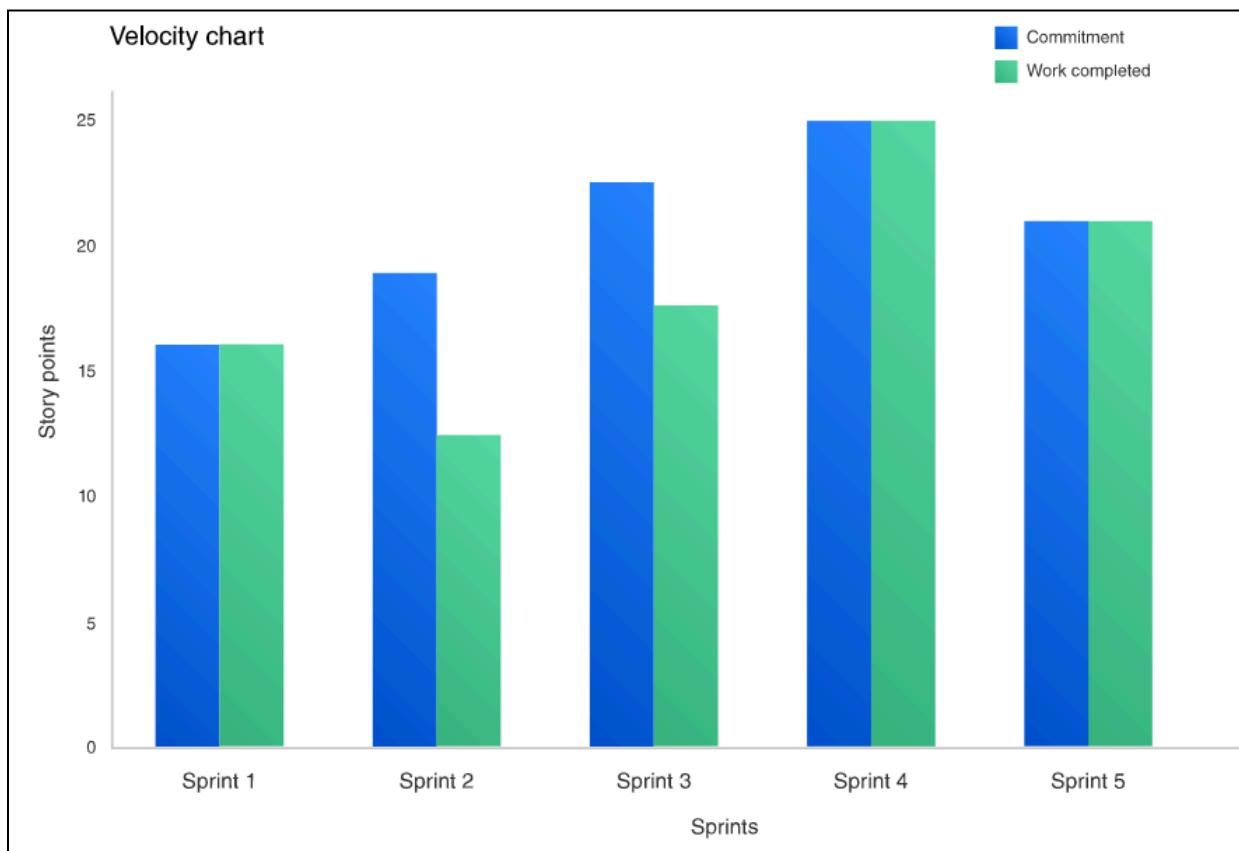
Se cuentan los story points de las user stories que están completas, no parcialmente completas

No se estima, **se calcula** al final del sprint

¿Y por qué es importante la velocidad?

- La velocidad es útil porque corrige los errores de estimación
- Me sirve para ir viendo si yo logro lo que pide el otro principio de ágil → Desarrollo sostenible

Si vos tenes un desarrollo sostenible vas a tener una grafica asi mira



¿Cómo calculamos la duración de un proyecto?

1. Estimamos como equipo todas las user points
2. Calculamos la métrica de velocidad
- 3. Cálculo de la duración del proyecto:**
 - a. Sumamos todos los story points estimados para las historias de usuario del proyecto.

- b. Divide el total de story points entre la velocidad del equipo para obtener el número de sprints necesarios para completar el proyecto.

Por ejemplo, si el proyecto tiene 120 story points y la velocidad del equipo es de 30 story points por sprint, se necesitan aproximadamente 4 sprints ($120 / 30 = 4$).

4. Por último multiplicamos la cantidad de sprints (4 en este caso) por la duración de cada uno (supongamos 1 mes)

$$4 * 1 \text{ mes} = 4 \text{ meses}$$

Poker estimation:

Es una técnica de estimación en ambientes ágiles publicada por Mike Cohn. Resulta de la conjunción de diferentes métodos como el de juicio experto, analogía y desagregación, basado en que *4 ojos ven más que 2*.

En contraposición al método de juicio experto tradicional, asume que las personas que desarrollan el producto deben ser aquellas que estimen. En otras palabras, el equipo estima su propio trabajo y los miembros opinan sobre lo mismo, donde se mezcla la experiencia y el conocimiento de cada uno con la posibilidad de compartir con el resto del grupo.

Un detalle es que el Product Owner participa en la planificación, pero no realiza estimaciones, suele ser el **moderador**, pero no es necesario que sea así siempre

Prerrequisitos:

- Lista de features/stories a ser estimadas
- Cada estimador tiene un mazo de cartas

Escalas:

- Tamaño por números: De 1 a 10
- Tallas de remeras: S, L, M
- Serie 2ⁿ: 1, 2, 4, 8, 16
- Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21

Optamos por elegir la **escala de Fibonacci** porque refleja un crecimiento exponencial que tiene la complejidad del software

Pasos:

1. Determinar la **base story** (la canónica) que será utilizada para comparar con las otras historias. Digamos, Story Z.

Está user suele ser la que tenga story point de 1, es decir baja complejidad, esfuerzo e incertidumbre

- i. La story a ser estimada se lee a todo el equipo
 - ii. Los estimadores discuten la story, haciendo preguntas al product owner cuando lo necesiten
 - iii. Cada estimador selecciona una carta y pone la carta boca abajo en la mesa.
 - iv. Cuando todos pusieron las cartas, las mismas se exponen al mismo tiempo.
 - v. Si todos los estimadores seleccionan el mismo valor, ese es el estimado. Sino, los estimadores discuten sus resultados, poniendo especial atención en los más altos y los más bajos. Despues de la charla, GOTO to 1.3
2. Se toma la próxima story, se discute con el producto owner
 3. Cada estimador asigna a la story un valor por comparación contra la base story
¿Cuan grande/pequeña, compleja, riesgosa es esta story comparada con la canónica?

Nunca elegir transacciones como canónicas

Suelen ser registros de entidades de negocio muy pequeñas

Goto 1.3

¿Cómo decodificar las estimaciones?

- **0:** Muy probable que no tengas idea de tu producto o funcionalidad en ese punto
- **1 / 2,1:** Funcionalidad pequeña (usualmente cosmética)
- **2 - 3:** Funcionalidad pequeña a mediana. Es lo que queremos
- **5:** Funcionalidad media. Es lo que queremos
- **8:** Funcionalidad grande, de todas formas lo podemos hacer, pero hay que preguntarse si no se pueden partir o dividir en algo más pequeño. No es lo mejor, pero todavía está bien
- **13:** ¿Alguien puede explicar por qué no lo podemos dividir?
- **20:** ¿Cuál es la razón de negocio que justifica semejante story y más fuerte aún, por qué no se puede dividir?
- **40:** No hay forma de hacer esto en un sprint
- **100:** Hay algo que está muy mal. Mejor ni arranquemos

Gestión de Productos de Software:

Aclaración:

Los conceptos que vamos a ver ahora, provienen de los que se conoce como **Lean Startup** (no es lo mismo que LEAN ya que este tema lo vemos en profundidad más adelante)

Lean startup es una adaptación de los principios y la filosofía de LEAN diseñada para **ayudar a emprendedores y empresas a desarrollar productos y servicios de manera más eficiente y efectiva**, especialmente en **entornos de alta incertidumbre**.

Fue introducida por **Eric Ries** en su libro "*The Lean Startup*" publicado en 2011. El enfoque principal es **acortar los ciclos de desarrollo de productos**, validar hipótesis de negocio rápidamente y aprender de manera continua y sistemática.



Concepto:

Un producto de Software es un artefacto que se construye para satisfacer una necesidad específica en base a ciertos requerimientos.

Cuando creamos un producto debemos enfocarnos en aquello que es de valor para el cliente, sabiendo que el 7% de las características del software son siempre usadas. Esto implica, que la mayor parte del valor agregado está concentrado en ese 7%. Alrededor del 80% de las características del software que se desarrolla no se utilizan. El desafío está en encontrar ese 7%.

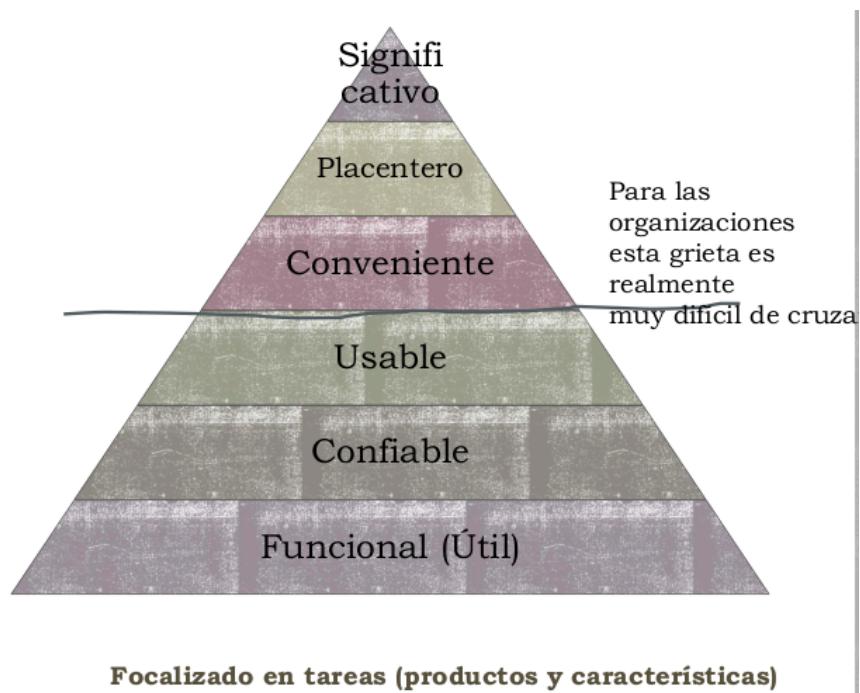
¿Y cómo evolucionan los productos?

Cuando vamos a desarrollar un software la mayoría de las veces se empieza a construir un producto sobre la base de lo **funcional**, que haga lo que tiene que hacer (las tareas que debe realizar), luego si se quiere avanzar más, el software debe ser **confiable** donde los usuarios no corran peligro usando el producto y los resultados que este producto me da yo pueda depender de ellos sin ningún problema.

Luego, la usabilidad tiene que ver con aspectos de la disciplina UX/UI, donde el usuario podrá lograr sus objetivos productivamente usando este producto, estando a gusto con su uso. Tiene que ser productivo, no es lo mismo que la utilidad.

Por ejemplo, si tardo 12 horas en realizar una operación donde el anterior tarda 4 horas no es productivo para mí y no mejora la calidad de vida al usuario.

Existe una línea en la pirámide que separa la usabilidad de la conveniencia. La mayoría de los productos no se ubican por encima de esta línea, debido a que resulta complejo que un software sea conveniente para el desempeño de tareas de ciertas personas (es decir, que el producto nos haga sentir que lo necesitamos porque nos produce un cambio, una facilidad para cumplir con un objetivo), y que estas no sean “esclavos” del software, limitándose a cumplir con las características de funcional, confiable y/o usable. Placentero y significativo para todo el mundo, algo que nos cambie la forma de usar algo o de vivir.



Los mínimos:

Lo que vamos a ver ahora son como los distintos elementos que se necesitan para poder construir un **producto** independientemente de su propósito (ya sea que quiera lograr algo significativo o quiera ganar guita)

Propuesta de Valor Única (UVP):

Bueno todo empieza con este concepto. Una declaración clara y concisa que explica por qué un producto o servicio es diferente y mejor que el de la competencia. Es el factor que hace que los clientes elijan tu producto sobre otros disponibles en el mercado

Nosotros acá vamos a plantear una hipótesis que explique por qué nuestro producto / servicio será único

Producto Mínimo Viable (MVP):

Eric Ries: Es una versión del producto que permite a un equipo recopilar la cantidad máxima de aprendizaje validado sobre clientes con el menor esfuerzo.

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

- Dirigido a un subconjunto de clientes potenciales
- Utilizado para obtener aprendizaje validado
- Más cercano a los prototipos que a una versión real funcionando de un producto
- Tiene el valor suficiente para que las personas estén dispuestas a usarlo o comprarlo inicialmente
- Demuestra suficiente beneficio futuro para retener a los primeros usuarios
- Proporciona un ciclo de retroalimentación para guiar el desarrollo futuro

Justamente una de las cosas que nosotros queremos lograr con nuestro MVP es validar la hipótesis antes establecida acerca de por qué nuestro producto / servicio es único

¿Qué hacemos para preparar un MVP?

1. Encontrar un nicho de mercado
2. **Roadmap:** Definición de muy alto nivel de qué características del producto se podrían tener a cierta altura del año. Esto sirve para los inversores
3. Investigar la competencia (Se hacen suposiciones)
4. Pre - vender el MVP
5. Testear las distintas suposiciones
6. Enfocarse solo en las funcionalidades principales

Característica Minima Viable (MVF):

El MVF incluye solo lo necesario para que la característica cumpla su propósito básico. Esto significa que no contiene extras o funciones adicionales que no sean fundamentales para la operación de la funcionalidad principal.

Es como una **versión en miniatura** del MVP

Característica a pequeña escala que se puede construir e implementar rápidamente, utilizando **recursos mínimos**, para una población objetivo para probar la utilidad y adopción de la característica.

Al igual que un MVP (Minimum Viable Product), la idea detrás del MVF es lanzar rápidamente una versión funcional de la característica para obtener **retroalimentación** de los usuarios reales. Esto permite validar si la nueva funcionalidad es útil, si satisface las necesidades de los usuarios y si vale la pena invertir más en su desarrollo.

Característica Minima Comercial (MMF):

Es una característica a pequeña escala que se puede construir e implementar rápidamente utilizando recursos mínimos, para una población objetivo para probar la utilidad y la adopción de una característica

El enfoque del MMF está en la **comercialización**. A diferencia del MVF, que puede ser más básico y enfocado en la validación, el MMF debe estar lo suficientemente completo como para que los usuarios lo perciban como una mejora valiosa del producto y, por lo tanto, justifique su promoción o venta.

- Esta es la pieza **más pequeña** de funcionalidad que puede ser liberada a los clientes.
- Tiene un **valor** tanto para los usuarios como para los clientes
- Es **parte de un MMR o un MMP**

Producto Mínimo Comercializable (MMP):

El MVP **evoluciona** al MMP luego de haber aprendido lo suficiente y haber validado la hipótesis

El MMP es una versión del producto que es lo suficientemente completa para ser lanzada al mercado y satisfacer a los primeros usuarios. Incluye todas las funcionalidades necesarias para que el producto sea comercializable y competitivo, aunque no necesariamente tenga todas las características finales planeadas.

Fíjate que este producto mínimo comercializable (MMP) está compuesto justamente por varias MMF

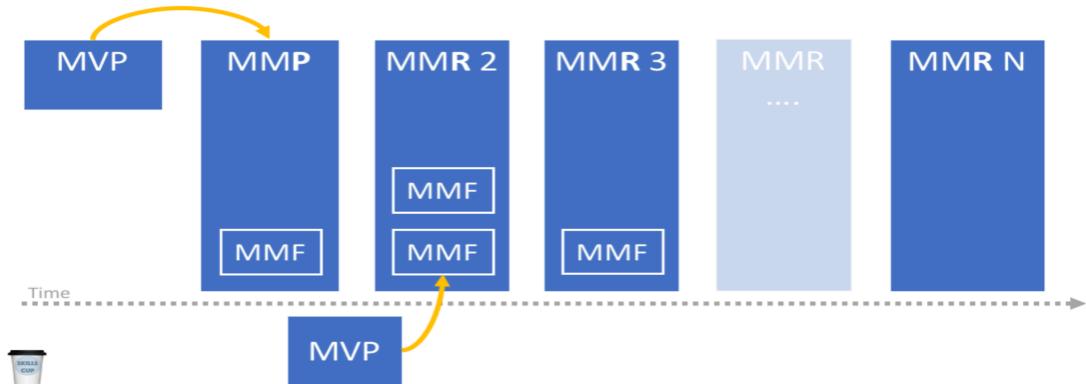
- Primer release de un MMR dirigido a **primeros usuarios - early adopters**
- Focalizado en características clave que satisfagan a ese grupo clave

Las Características Mínima de Release (MMR):

- ¿Que tiene que tener como características mínimas un release de un producto para poder ser liberado?
- **Release** de un producto que tiene el conjunto de característica lo más pequeño posible

- Un incremento más pequeño que ofrece un valor nuevo a los usuarios y satisface sus necesidades actuales
- **MMP = MMR1**

¿Cómo se relacionan todos estos conceptos?



¿Qué errores comunes podemos cometer con estos conceptos?

- Enfatizar la parte **mínima** del MVP con exclusión de la parte **viable**. El producto entregado no es de calidad suficiente para proporcionar una evaluación precisa de si los clientes utilizarán o no el producto
- Entregar lo que consideran un MVP, y luego no hacer más cambios a ese producto, independientemente de los comentarios que reciban al respecto.
- Confundir un **MVP** que se enfoca en el **aprendizaje**, con una MMF o con un MMP que ambos se enfocan en **ganar**

Valor vs Desperdicio:

¿Cuáles de nuestros esfuerzos crean valor y cuáles son desperdicio?

Lean Thinking define la **creación de valor** como proveer beneficios a los clientes, cualquier otra cosa será desperdicio

La productividad de un Startup no puede medirse en términos de cuánto se construye cada día, por el contrario, se debe medir en términos de construir lo correcto cada día y lo correcto es aquello que agrega valor.

Si no construimos valor, entonces estamos invirtiendo esfuerzo en desperdicio.

Design Thinking:

Mnemotecnia → EDIPO (EDIIPP)

Design Thinking es una metodología de innovación centrada en las personas que busca resolver problemas complejos de forma creativa. Se basa en un **proceso iterativo**

Ponerse en la piel del usuario que va a consumir el producto

Es básicamente solucionar problemas de una manera **creativa e innovadora**

Para llevar a cabo su proceso, utiliza un esquema de **pensamiento divergente y convergente**. Con el primero se busca tener una visión amplia del espectro de soluciones que responden al problema inicial. El segundo consiste en reducir a la mejor idea el conjunto de posibles soluciones, con el objetivo de brindar aquella que es más adecuada para las necesidades del cliente.

El proceso cuenta con **5 pasos y es iterativo**:

1. **Empatizar:** Implica comprender las necesidades del cliente mediante la utilización del razonamiento intuitivo
2. **Definir:** Determinar lo valioso para el cliente a partir de lo recopilado durante la primera etapa, lo cual permitirá guiar el trabajo.
3. **Idear:** Articular muchas soluciones posibles. Esto se puede lograr en los **equipos multifuncionales**, donde cada miembro aporta una mirada diferente para generar muchas ideas para luego filtrar las más viables.

4. **Prototipar:** Consiste en materializar las ideas. Las ideas seleccionadas se transforman en prototipos que permitan visualizar la solución propuesta.
5. **Probar:** Es una de las etapas más importantes del proceso, ya que permite validar los prototipos con los usuarios implicados en la solución, obteniendo retroalimentación y permitiendo iterar.

Ejemplo:

Empresa que presta servicio de alquiler de herramientas como taladros para que las personas puedan colgar sus cuadros en la pared.

Empatiza con la gente porque no se los vende para que lo usen una vez y nunca más, sino que se los alquilan por día para que realmente cumplan sus expectativas y necesidades

Lean UX:

Es un **marco o enfoque de trabajo colaborativo** que se enfoca en diseñar productos digitales de forma iterativa, validando hipótesis y aprendiendo rápidamente del usuario.

Se centra en **3 pilares fundamentales**:

1. Design Thinking (Esto ya lo explicamos más arriba)
2. El desarrollo de software con metodologías ágiles

Esto es importante ya que Lean UX hereda los valores que se trabajaban en el agilismo, los cuales eran:

- Individuos e interacciones por sobre procesos y herramientas
- Software funcionando por sobre documentación exhausitiva
- Relación con el cliente en lugar de una negociación contractual
- Adaptarse a cambios en lugar de seguir un plan

3. El último pilar es básicamente lo que se plantea en el Libro de Lean Startup de Eric Ries, que es básicamente este proceso de **build, measure, learn**

- **Build:**
 - Se crea un MVP para poder validar nuestra hipótesis y lo lanzamos al mercado
 - No nos interesa que funcione o no es solo para aprendizaje y feedback real
- **Measure:**
 - Se recogen datos reales de los usuarios para poder generar métricas
 - ¿Cuánta gente usa esta funcionalidad?
 - ¿Cuánto tiempo la gente pasa dentro de la app?
- **Lean:**
 - Esta es la parte en donde se analizan los resultados de los datos y podemos determinar si la hipótesis ha sido validada o no
 - El aprendizaje nos sirve para la próxima iteración

Un MVP varía en complejidad desde pruebas de humo (smoke tests) **extremadamente simples** (poco más que un anuncio) hasta prototipos tempranos



El dilema de la audacia cero

El "dilema de la audacia cero" es un concepto que se refiere a la paradoja que enfrentan las organizaciones o individuos cuando, en un intento de evitar cualquier tipo de riesgo o fracaso, se vuelven tan cautelosos que terminan estancándose o siendo incapaces de innovar o progresar.

Si vos empezas a **posponer la experimentación** con el MVP, van a surgir algunos resultados desafortunados como:

- La cantidad de trabajo desperdiciado puede aumentar
- Se perderán los comentarios esenciales
- El riesgo de que su startup construya algo que nadie quiere puede aumentar

Compensaciones:

- ¿Preferiría atraer capital de riesgo y potencialmente derrocharlo?
- ¿O preferiría atraer capital de riesgo y utilizarlo sabiamente?

Use un MVP para experimentar (inicialmente, en silencio) con los primeros usuarios del mercado

Verifique su concepto probando TODOS sus elementos, comenzando por los más riesgosos

Framework Scrum 2020:

Concepto:

Scrum es un marco ligero que ayuda a las personas, equipos y organizaciones a **generar valor** a través de soluciones adaptables para problemas complejos.

Scrum, requiere de un **Scrum Master** para fomentar un entorno donde:

1. Un propietario del producto (Product Owner) ordena el trabajo de un problema complejo en un Product Backlog.

2. El equipo de Scrum convierte una selección del trabajo en un incremento de valor durante un Sprint.
3. El equipo de Scrum y sus partes interesadas inspeccionan los resultados y realizan los ajustes necesarios para el próximo Sprint.
4. Repetir

El marco de Scrum es deliberadamente **incompleto**, 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.

Ahora bien, SCRUM utiliza procesos empíricos por ende **se basa en los 3 pilares empíricos (T.I.A):**

- **Transparencia:** En el centro del scrum se encuentra la transparencia, un principio fundamental que se centra en la comunicación abierta y sin obstáculos.

La transparencia es la base de la confianza y la colaboración, ya que promueve un intercambio de información claro y sincero entre todas las partes interesadas del proyecto.

Transparencia en el proceso y en el producto

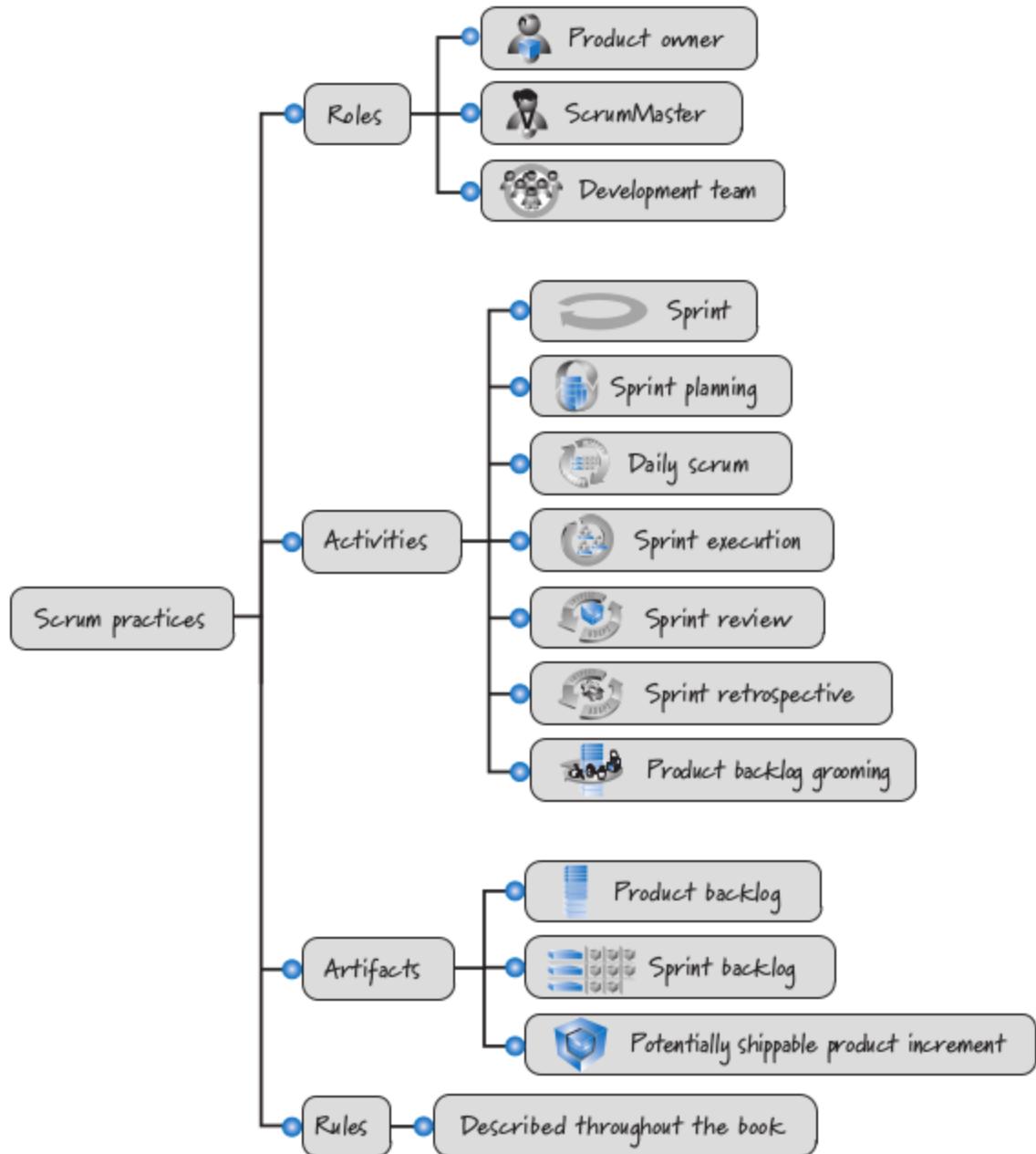
- **Inspección:** Los equipos deben **identificar las desviaciones mediante evaluaciones periódicas**, lo que fomenta la mejora y mantiene la trayectoria hacia el éxito del proyecto.
- **Adaptación:** Una vez que el equipo ha inspeccionado el producto y los procesos, adapta sus estrategias en función de los conocimientos adquiridos. A medida que los equipos descubren nueva información y entienden mejor la dinámica de sus proyectos, pueden corregir el rumbo de una forma ágil.

¿En qué valores se basa SCRUM?

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

Compromiso, Enfoque, Apertura, Respeto y Coraje

Scrum - Framework:



Roles:

1. Scrum Master:

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

Sirve al equipo:

- Es responsable de su **efectividad**
- **Promueve la mejora continua**
- **Capacita a los miembros del equipo en autogestión y multifuncionalidad**
- Ayuda al equipo a **centrarse en la creación de incrementos de alto valor**
- **Asegura que los sprints o incrementos sean llevados a cabo, dentro de un time box determinado y que sean productivos**

Sirve al Product Owner:

- Ayudar a encontrar técnicas para una **definición eficaz de los objetivos del producto** y la gestión de los **retrasos en el producto**
- Ayuda a establecer la **planificación de productos para un entorno complejo**
- Facilita la **colaboración de las partes interesadas** según sea solicitado

2. Product Owner:

El Propietario del Producto es responsable de **maximizar el valor del producto resultante del trabajo del equipo de Scrum**.

Encargado de la **gestión eficaz del producto backlog**.

- Definición del producto
- Creación de elementos de trabajo
- Asegurarse que el trabajo pendiente sea transparente, visible y comprendido

El Propietario del Producto puede hacer el trabajo anterior o puede delegar la responsabilidad a otros. En cualquier caso, el propietario del producto sigue siendo responsable.

Para que los Propietarios de Productos tengan éxito, toda la organización debe respetar sus decisiones.

3. Scrum Team:

La unidad fundamental de Scrum es un pequeño equipo de personas, un equipo Scrum. El equipo Scrum consta de un Scrum Master, un propietario de producto (Product Owner) y desarrolladores. Dentro de un equipo de Scrum, no hay subequipos ni jerarquías. Es una **unidad cohesionada** de profesionales enfocada en un objetivo a la vez, el objetivo del Producto.

Fíjate que en el scrum team a las personas no se las etiqueta ni hay roles super definidos como en el enfoque ágil; acá tenemos básicamente a los **developers** que esto envuelve a analistas, desarrolladores, testers, aseguradores de calidad, etc

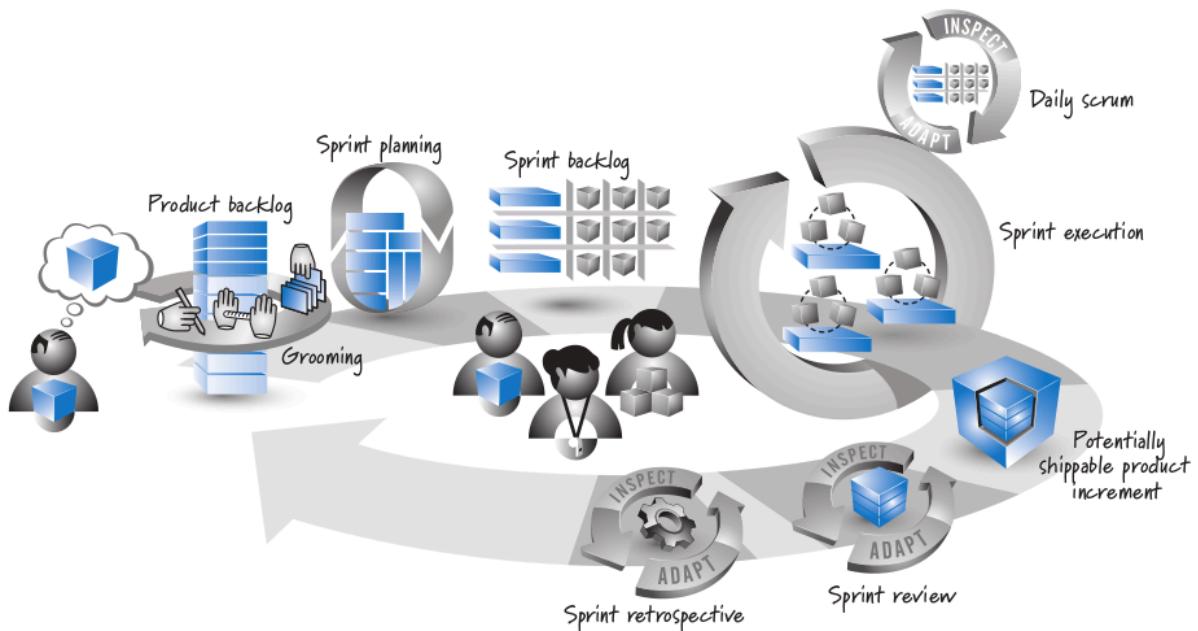
Los equipos de Scrum son **muy funcionales**, lo que significa que los miembros tienen todas las habilidades necesarias para crear valor en cada Sprint.

También son **autogestionados**, lo que significa que internamente deciden quién hace qué, cuándo y cómo.

El equipo de Scrum es lo suficientemente pequeño como para permanecer ágil y lo suficientemente grande como para completar un trabajo significativo dentro de un Sprint, por lo general **10 o menos personas**.

Actividades:

Vamos a explicar esto, por medio del desarrollo de la siguiente imagen ilustrativa:



n

El sprint:

Los sprints son el latido del corazón de Scrum, donde las ideas se convierten en valor.

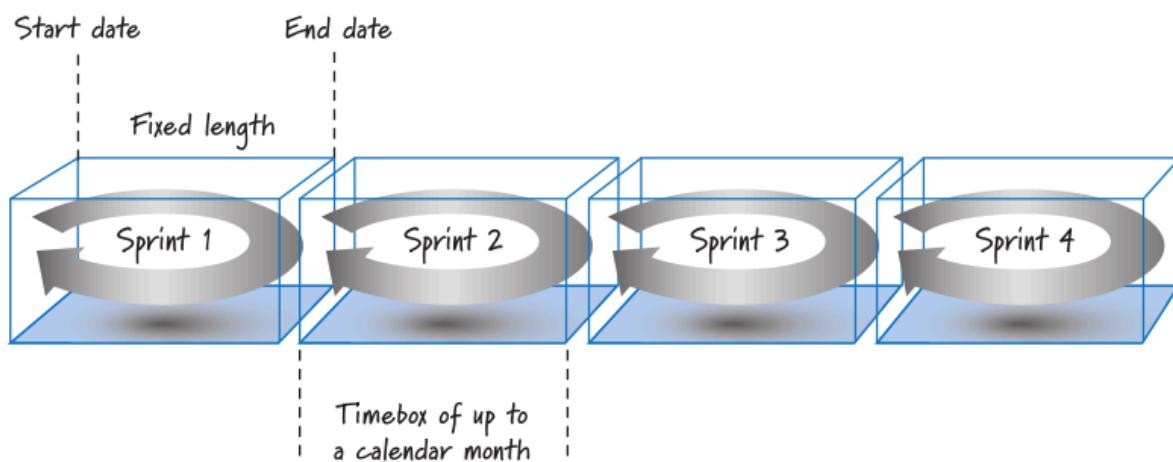
Son eventos de **longitud fija** de un mes o menos para crear consistencia. Un nuevo Sprint comienza inmediatamente después de la conclusión del Sprint anterior.

Fíjate que durante el sprint:

- No se hacen cambios que pongan en peligro el objetivo del sprint → Triángulo de hierro
- La calidad no disminuye
- El alcance se puede clarificar y negociar con el Product Owner a medida que vamos aprendiendo más

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

Es importante entender que los sprint se ubican en un **timebox** de manera que siempre tengan una fecha de inicio y fin fija. Además suelen tener un tiempo de duración similar



Planificación de Sprint:

El Sprint Planning inicia el Sprint estableciendo el trabajo que se realizará para el mismo. Este plan resultante es creado por el trabajo colaborativo de todo el equipo de Scrum

Se abordan los siguientes temas:

1. ¿Por qué el sprint es valioso?

El Propietario del Producto (Product Owner) propone cómo el producto podría aumentar su valor y utilidad en el Sprint actual..

Se define un **objetivo del sprint**, que explica justamente por qué el sprint es valioso

2. ¿Qué se puede hacer con este Sprint?

A través del debate con el propietario del producto (Product Owner), los desarrolladores seleccionan los elementos del Product Backlog para incluir en el Sprint actual. El

equipo de Scrum puede refinar estos elementos durante este proceso, lo que aumenta la comprensión y confianza.

3. ¿Cómo se realizará el trabajo?

Para cada elemento de trabajo pendiente de producto (Product Backlog item) seleccionado, los desarrolladores planifican el trabajo necesario para crear un incremento que cumpla con la definición de hecho.

En este punto, se realiza un proceso de descomposición de los elementos del product backlog, de esta manera se los define en productos más pequeños que se puedan realizar en unos pocos días

El objetivo del sprint, los elementos seleccionados a trabajar en el sprint, más el plan para entregarlos se conocen conjuntamente como el Sprint Backlog

El sprint planning tiene una duración máxima de 8 horas para un Sprint de un mes

Daily Scrum:

El propósito del Daily Scrum es inspeccionar el progreso hacia el Objetivo Sprint y adaptar el Sprint Backlog según sea necesario.

El Daily Scrum es un evento de 15 minutos (máximo) para los desarrolladores del equipo de Scrum. Para reducir la complejidad, se lleva a cabo al mismo tiempo y lugar todos los días laborables del Sprint.

También se lo suele llamar daily stand-up porque una práctica común es que todos se levanten durante la reunión para que sea más breve

Un buen acercamiento a esta práctica consiste en el Scrum Master como facilitador, ya que les habla a cada miembro del equipo y juntos responde 3 preguntas en beneficio del resto del team:

- ¿Qué es lo que he realizado desde el último daily scrum?
- ¿En qué estoy pensando trabajar en el próximo daily scrum?
- ¿Cuáles son los obstáculos que me impiden lograr un mayor progreso?

Por último, un dato extra es que Scrum ha utilizado los términos **pigs** y **chickens** para poder distinguir a quienes deberían participar durante la daily scrum y los que simplemente deberían observar

- Con respecto a esto hay opiniones divididas, sin embargo el autor del libro señala que en realidad es más útil que todo sean **pigs** (osea que hagan algo)

Sprint Review:

El propósito de la revisión del Sprint es **inspeccionar** el resultado del Sprint y determinar **futuras adaptaciones**. El equipo de Scrum presenta los resultados de su trabajo a las partes interesadas clave y se discute el progreso hacia el **objetivo del producto**

Se analizan los **incrementos**

Se recibe **feedback** proporcionado por los stakeholders (clientes, patrocinadores, etc)

Acá es donde se calcula la **velocity** del sprint

La revisión de Sprint es el penúltimo evento del Sprint y se utiliza en un **plazo máximo de cuatro horas** para un Sprint de un mes.

En este momento, es donde **las user stories que cumplen con la DoD** son presentadas al product owner justamente para inspección y adaptación

Sprint Retrospectives:

Mientras que el Sprint Review es un tiempo de inspección y adaptación acerca del **producto**, el Sprint Retrospective es un momento de inspección y adaptación del **proceso**

El equipo de Scrum inspecciona cómo fue el último Sprint con **respecto a individuos, interacciones, procesos, herramientas y su definición de Hecho**. Los elementos inspeccionados a menudo varían según el dominio del trabajo. Las suposiciones que los desviaron se identifican y se exploran sus orígenes.

El equipo de Scrum analiza qué fue bien durante el Sprint, qué problemas encontró y cómo esos problemas fueron (o no fueron) resueltos.

La retrospectiva Sprint concluye el Sprint. Se utiliza un intervalo de tiempo de hasta un **máximo de tres horas** para un Sprint de un mes. Para sprints más cortos, el evento suele ser más corto.

Importante destacar que a lo largo de la sprint retrospective se tratan **3 cuestiones importantes:**

- ¿Qué es lo que salió **bien**?
- ¿Qué es lo que salió **mal**?
- ¿Qué es lo que pudo haberse **mejorado**?

¿Y cuál es la metodología para realizar una retrospectiva?

1. Se prepara el escenario
2. Se reúnen los datos → *Se hace una vista hacia el pasado, es como ver por el espejo retrovisor*
3. Se generan ideas
4. Se decide qué hacer
5. Se cierra la retrospectiva

Artefactos de Scrum:

Los artefactos de Scrum representan **trabajo o valor**. Están **diseñados para maximizar la transparencia** de la información clave.

Cada artefacto contiene un **compromiso** para garantizar que proporciona información que mejora la transparencia y el enfoque con el que se puede medir el progreso:

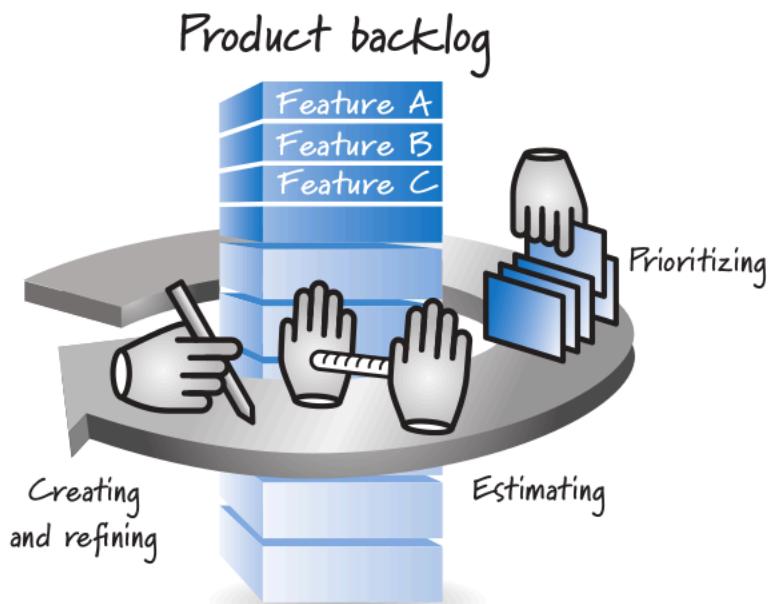
Product Backlog:

El trabajo pendiente del producto es una lista emergente y ordenada de lo que se necesita para mejorar el producto. Es la única fuente de trabajo emprendida por el equipo Scrum.

La actividad de crear, refinar los ítems del product backlog, estimarlos y priorizarlos se conoce como **grooming**

Scrum de por si no dicta algún tipo de medida específica para medir los ítems del product backlog, sin embargo en la práctica nos vamos a valer de **medidas relativas** para poder comparar los items y asi poderlos priorizar como pueden ser los story points o días ideales

Cada elemento de trabajo del product backlog debe acercar al equipo de trabajo cada vez más hacia el objetivo de producto



Compromiso: Objetivo del Producto

El objetivo del producto (Product Goal) describe un estado futuro del producto que puede servir como objetivo para el equipo Scrum contra el cual planificar. El objetivo del producto se encuentra en el trabajo pendiente del producto (Product Backlog). El resto del trabajo pendiente del producto surge para definir "qué" cumplirá el objetivo del producto.

El objetivo del producto es el objetivo a largo plazo para el equipo Scrum. Deben cumplir (o abandonar) un objetivo antes de asumir el siguiente.

Sprint Backlog:

El Trabajo pendiente de Sprint se compone del objetivo sprint (por qué), el conjunto de elementos de trabajo pendiente de producto seleccionados para el Sprint (qué), así como un plan accionable para entregar el incremento (cómo).

Es una imagen muy visible y en tiempo real del trabajo que los desarrolladores planean realizar durante el Sprint para lograr el Objetivo Sprint.

Compromiso: Sprint Goal

El Sprint Goal es el único objetivo para el Sprint.

El objetivo de Sprint se crea durante el evento Sprint Planning y, a continuación, se agrega al Trabajo pendiente de Sprint. A medida que los desarrolladores trabajan durante el Sprint, tienen en cuenta el objetivo de Sprint. Si el trabajo resulta ser diferente de lo que esperaban, colaboran con el propietario del producto para negociar el alcance del Trabajo pendiente de Sprint dentro del Sprint sin afectar al objetivo de Sprint.

Incremento:

Un Incremento es un **paso de hormigón** hacia el Objetivo del Producto.

Se pueden crear **varios incrementos dentro de un Sprint**. La suma de los incrementos se presenta en la Revisión Sprint apoyando así el empirismo. Sin embargo, un incremento puede ser entregado a las partes interesadas antes del final del Sprint. La revisión de Sprint nunca debe considerarse una puerta para liberar valor.

El trabajo no se puede considerar parte de un incremento a menos que cumpla con la Definición de Hecho.

Compromiso: Definición de hecho

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

En el momento en que un elemento de trabajo pendiente de producto cumple con la definición de hecho, **se crea un incremento**.

La definición de Hecho **crea transparencia** al proporcionar a todos una comprensión compartida de qué trabajo se completó como parte del Incremento.

Todo lo que tenga de estado **done** son características del producto que están habilitadas para poder ser presentadas al product owner y que él las acepta o no

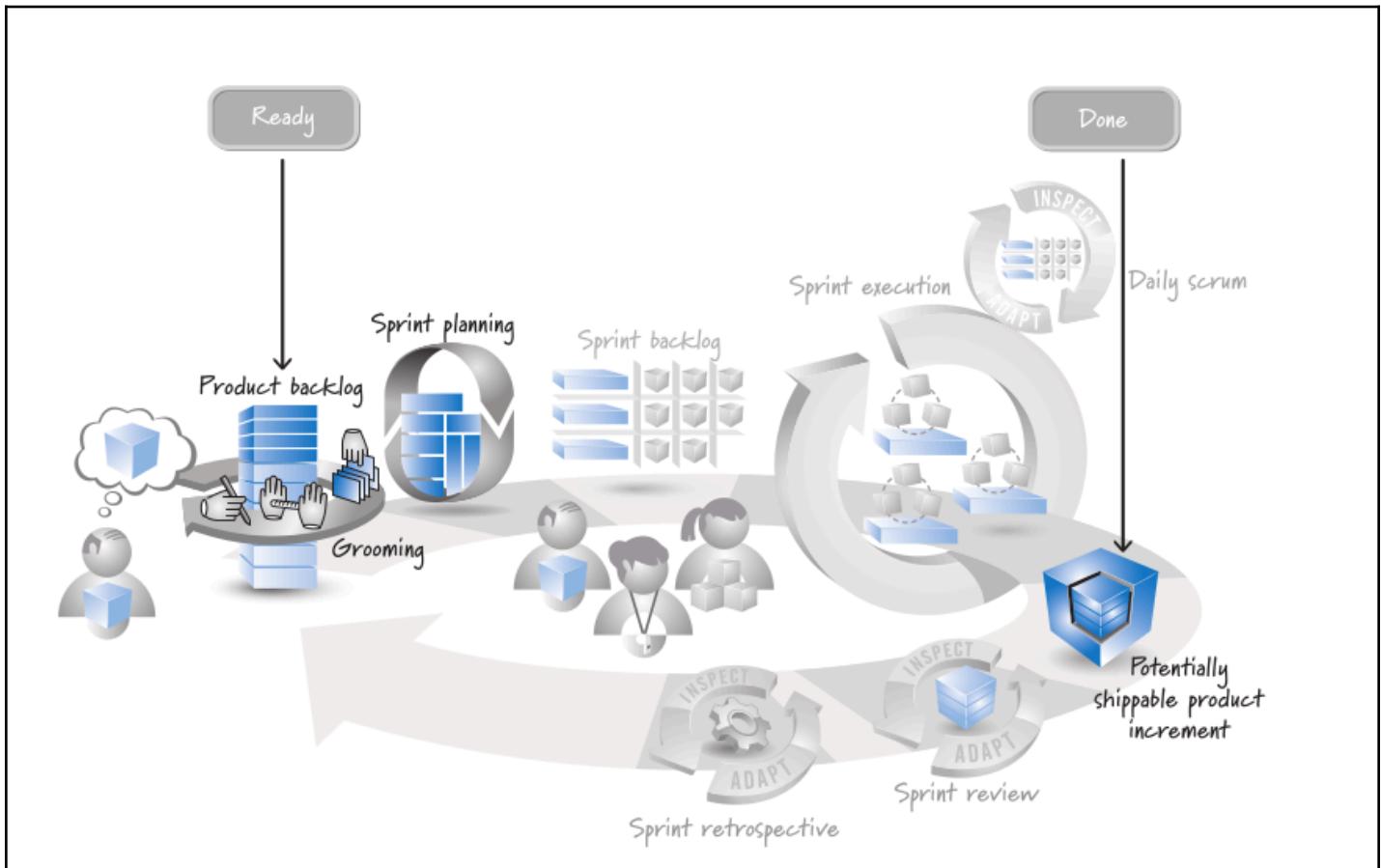
Las user stories que están en estado de done, pueden entrar en el sprint review

Luego de esta aceptación, **irá o no a producción**

Si un elemento de trabajo pendiente de producto no cumple con la definición de hecho, no se puede liberar, ni siquiera presentar en la revisión de Sprint. En su lugar, vuelve al Trabajo pendiente del producto para su consideración futura.

Definición de Hecho (DONE)	
<input type="checkbox"/>	Diseño revisado
<input type="checkbox"/>	Código Completo
<input type="checkbox"/>	Código refactorizado
<input type="checkbox"/>	Código con formato estándar
<input type="checkbox"/>	Código Comentado
<input type="checkbox"/>	Código en el repositorio
<input type="checkbox"/>	Código Inspeccionado
<input type="checkbox"/>	Documentación de Usuario actualizada
<input type="checkbox"/>	Probado
<input type="checkbox"/>	Prueba de unidad hecha
<input type="checkbox"/>	Prueba de integración hecha
<input type="checkbox"/>	Prueba de sistema hecha
<input type="checkbox"/>	Cero defectos conocidos
<input type="checkbox"/>	Prueba de Aceptación realizada
<input type="checkbox"/>	En los servidores de producción

Aclaración: Para que no te pierdas, ésta es la ubicación de el definition of ready y el definition of done en Scrum:



Herramientas de Scrum:

Uno de los beneficios de trabajar en cortos timeboxes con pequeños equipos es que no necesitas gráficos complejos ni reportes para comunicar el progreso.

A continuación veremos los distintos **métodos para comunicar el progreso:**

Taskboard:

A continuación se muestra una configuración básica de un tablero de Scrum

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

Story: Son las historias que han salido del product backlog y han entrado al sprint backlog.

Es básicamente un **artefacto visual**, que me permite visualizar el flujo de trabajo de un sprint backlog

Ahora bien, a menudo resulta útil **desagregar** el trabajo de una user story en varios actividades, con lo cual, a cada una se le asignan **horas de trabajo ideales**

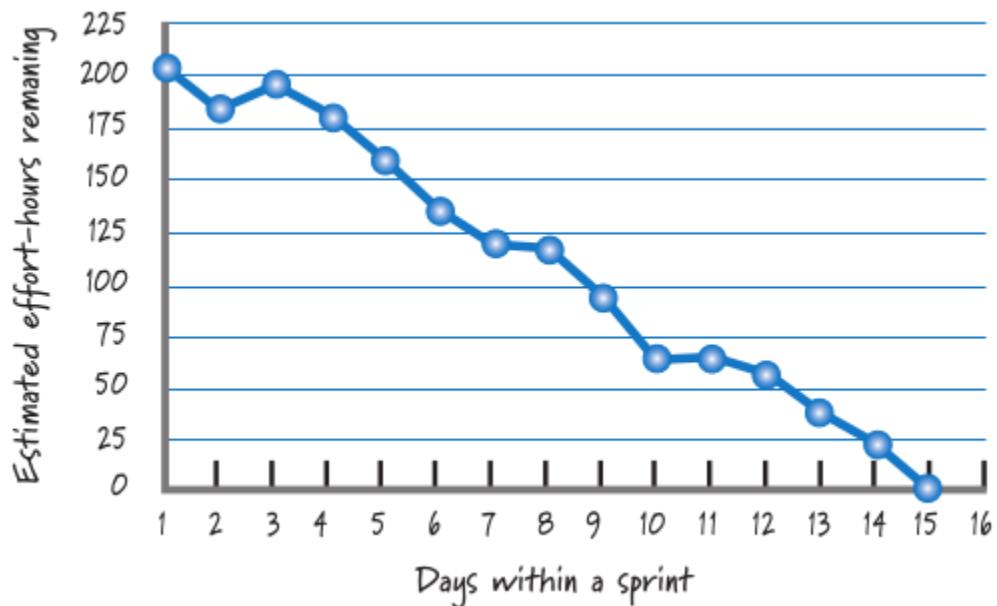
Con respecto a la granularidad

Cuando vos tenes un tablero que tiene pocas tareas, tareas gordas con mucho contenido es porque estás utilizando una **granularidad mala**.

Lo que nosotros queremos es tener el taskboard lleno de actividades, para poder lograr una **granularidad fina** de las users

NOTAR que tanto Ágil como Lean apuntan a una **gestión binaria**: Esto quiere decir que las cosas están terminadas o no, no hay punto medio

Sprint Burndown Charts:

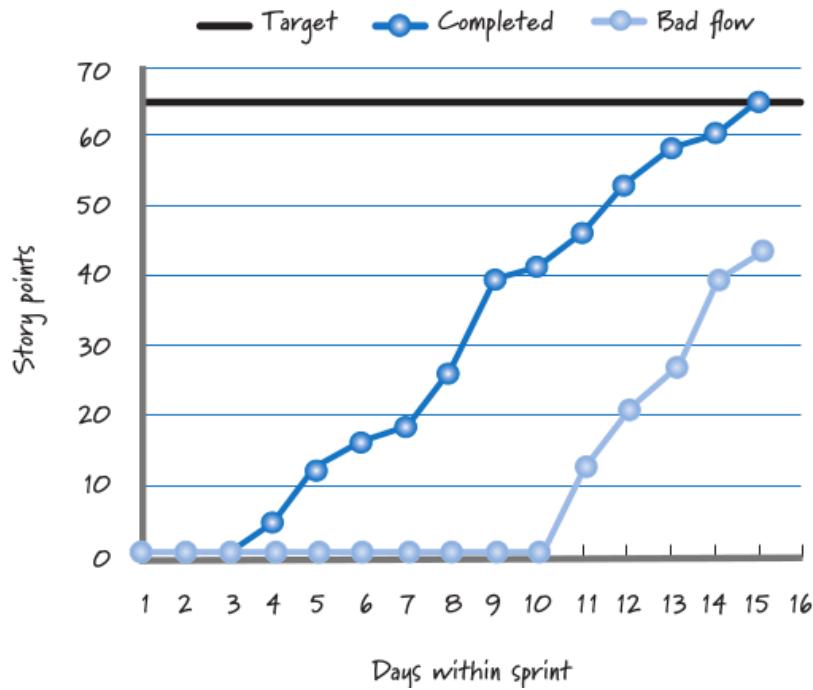


Hemos de fijar que en el eje de las y ubicamos los **story points** y en el eje de las x ubicamos los días que dura el sprint

Empezamos con un compromiso y de ahí en más, el gráfico va descendiendo a medida que se vayan completando las actividades

Fíjate que el gráfico de burndown, se va a ir actualizando en las **daily sprints**, que es donde las personas anuncian el progreso y lo que han completado o no

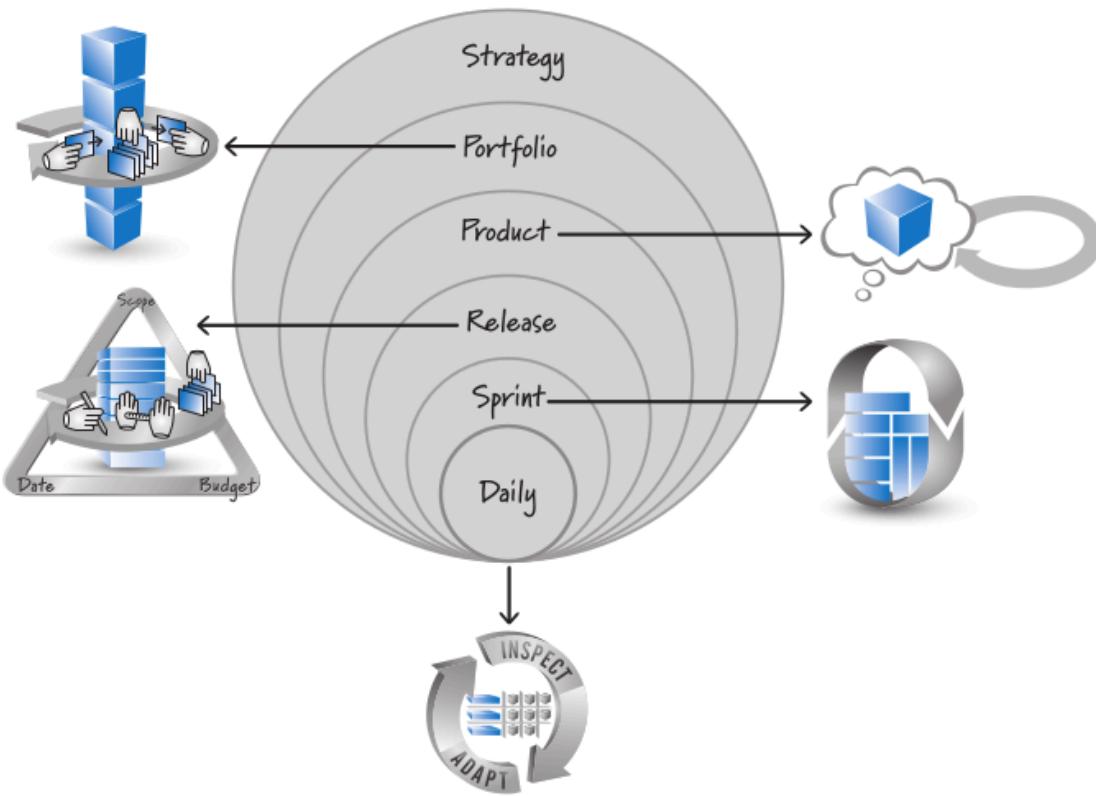
Sprint Burnup Chart:



Este gráfico es al revés, vamos **sumando story points** a medida que las stories se van completando

Múltiples niveles de planificación:

Lo que vamos a ver ahora son básicamente distintos **niveles de granularidad**, con los que nosotros trabajamos en SCRUM, y con que artefacto se representa estos niveles



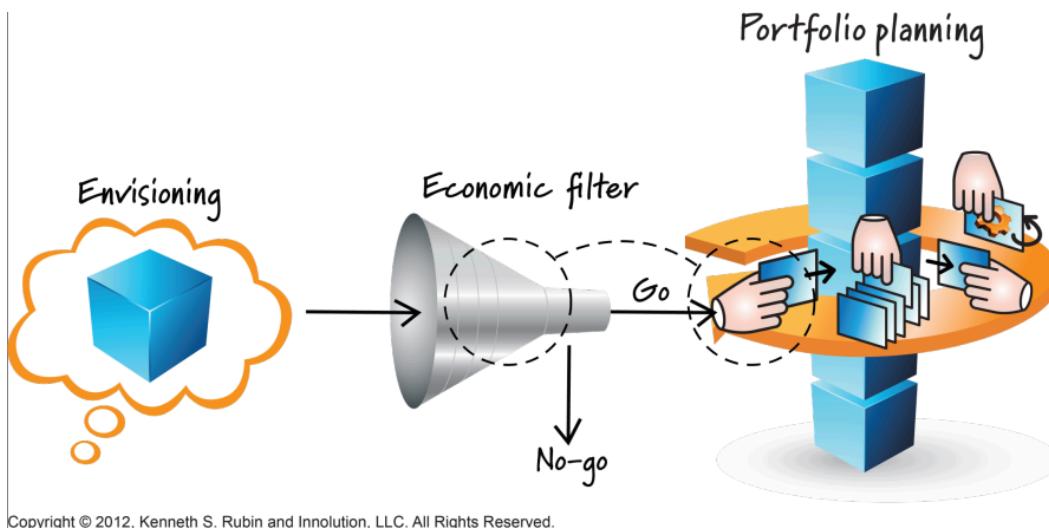
1. Estrategia:

- Visión general del negocio.
- Objetivos estratégicos de largo plazo.
- Ejemplo: “Creación de productos digitales que permitan el crecimiento económico de las compañías.”

2. Planificación del Portfolio:

Es una actividad que sirve para determinar qué productos son los que nosotros vamos a trabajar, en qué orden y por cuánto tiempo.

En realidad para poder realizar un portfolio planning se tiene que pasar por un tipo de **filtro económico** el cual me dice es es redituable a nivel económico



3. Planificación del producto:

El envisioning comienza con la creación de una visión, seguida de la creación de product backlog de alto nivel y frecuentemente un roadmap de producto.

El envisioning tiene como **inputs**:

- Horizonte de planeamiento:
 - *Planificamos hasta un límite de 3 años*
- Fecha a completar
- Recursos / Presupuesto
- Umbral de confianza
 - *Tenemos alta confianza en esta planificación*

El envisioning tiene como **participantes**:

- Stakeholders internos
- Equipo de Scrum
- Otros

El envisioning tiene como **output**:

- Vision del producto
 - Product Backlog
 - Product roadmap
 - Demás artefactos
-

Nota: Con respecto a estos dos primeros niveles de planificación, nosotros en la cátedra no vamos a hacer foco en ellos, sino en lo que viene ahora. *Meles*

Muy bien, entonces de acá se desprenden los siguientes elementos

- **Vision:**

Viene a ser una **declaración del propósito** del producto el cual va a generar **valor** para los distintos interesados

- **Product Backlog de alto nivel:**

Una vez establecida la visión, el próximo paso es generar una versión inicial de un product backlog.

Es muy importante notar que el **PB** nunca está terminado y tampoco necesitamos esto para poder comenzar a trabajar. Solamente tiene que tener la suficiente cantidad de características de producto para que podamos empezar a poner manos a la obra

El PB no es el la ERS, porque acordate que una vez que vos sacas características del PB para trabajar en un Sprint, las mismas desaparecen del mismo

A partir de este PB, yo voy a establecer una **planificación de releases**.

Es muy importante notar que un **release** va a estar formado por un conjunto de sprints

Como **inputs** del release planning tenemos:

- Vision del producto
- Product backlog

- Product Roadmap
- Velocity

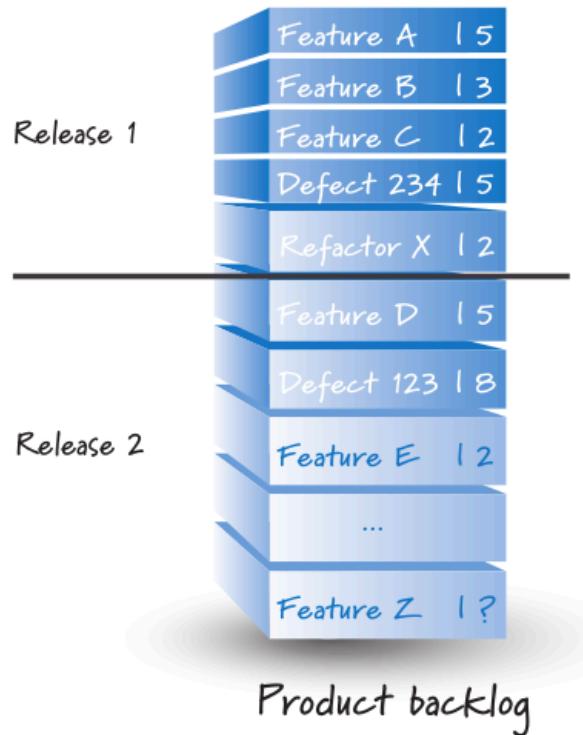
Como **participantes** del release planning tenemos:

- Stakeholders Internos
- Equipo de Scrum

Al momento de definir un release, es importante tener en cuenta el alcance, la fecha, el presupuesto, que serían restricciones con las que tenemos que ir jugando

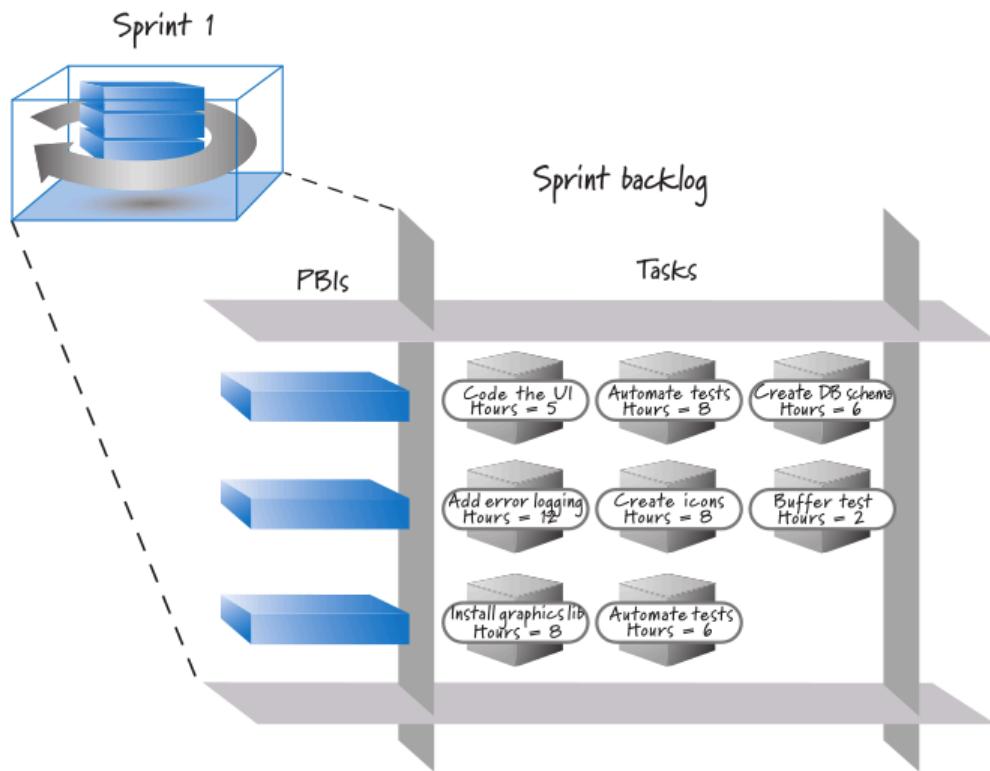
Entonces **se genera un release plan:**

- Ranago de features (características) o Rango de Sprints
- MRFs → Minimum Release Features
- Sprint Map



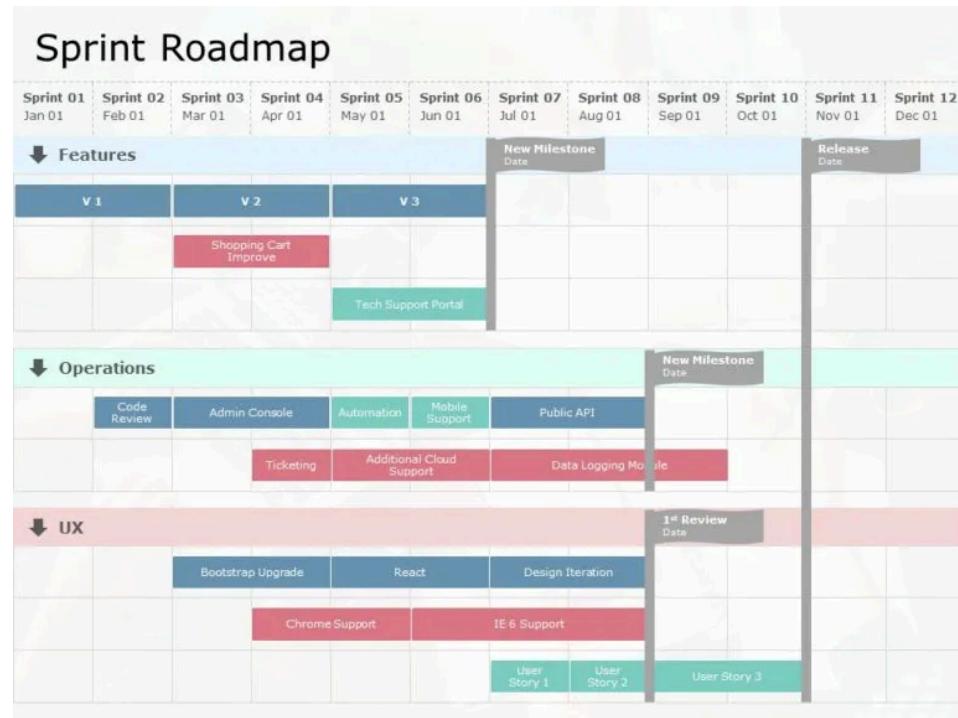
Ahora bien, si nos situamos a un lugar más fino de planificación, nos encontramos con el Sprint Planning y el Daily planning

Con el **sprint planning**, vamos a determinar cual de todas los elementos que se encuentran en el product backlog, vamos a trabajar en el **sprint x**

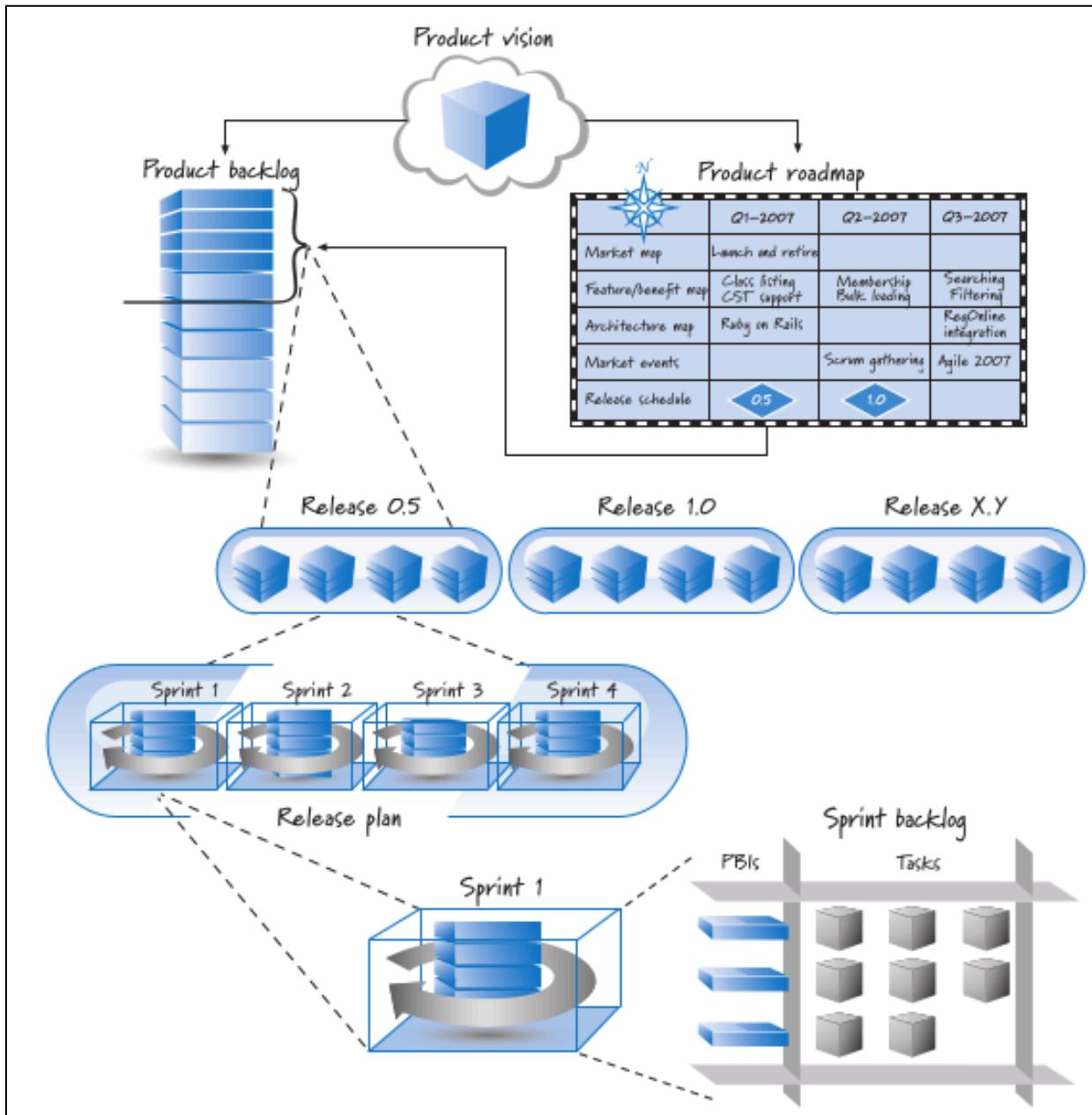


Con la **daily planning**: Este es el nivel de planificación más detallado. Durante el daily scrum, los miembros del equipo describen de manera colectiva y de manera muy visible, el plan del día es decir la big - picture del día.

- **Roadmap:** Es la **hoja de ruta**. Un roadmap de producto **comunica la naturaleza incremental** de cómo se construirá y se entregará el producto con el tiempo, junto con los factores importantes que impulsan cada lanzamiento individual.
-

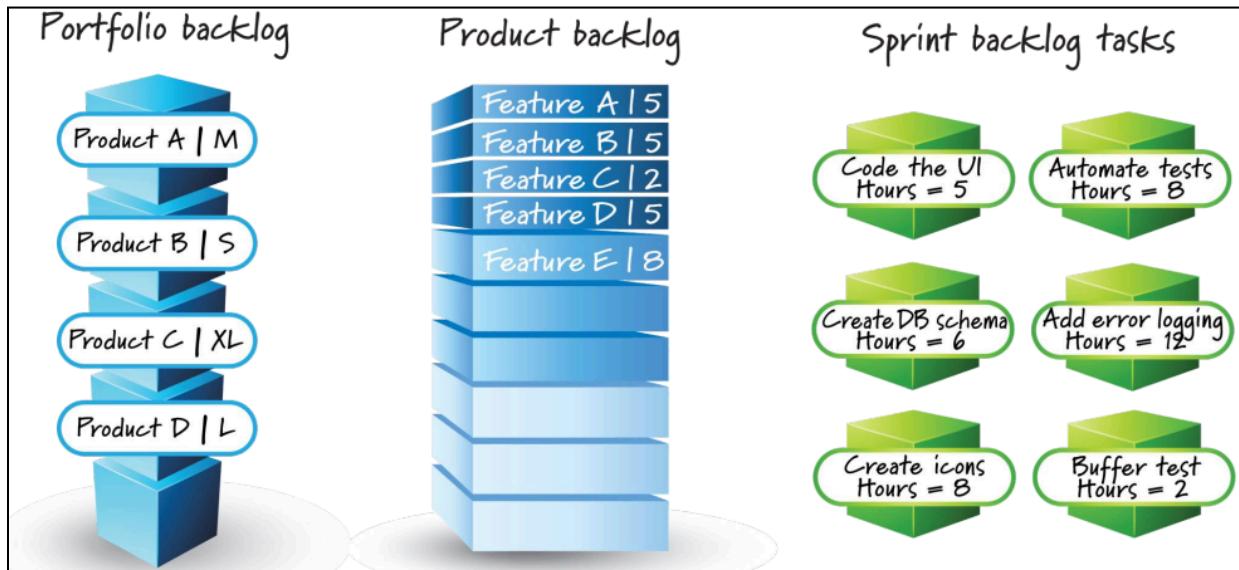


¿Cómo se ven todos estos elementos relacionados?



¿Qué y cuándo estimar?

De acuerdo al **nivel de granularidad** con el que estamos trabajando, podemos optar por distintas alternativas a la hora de estimar



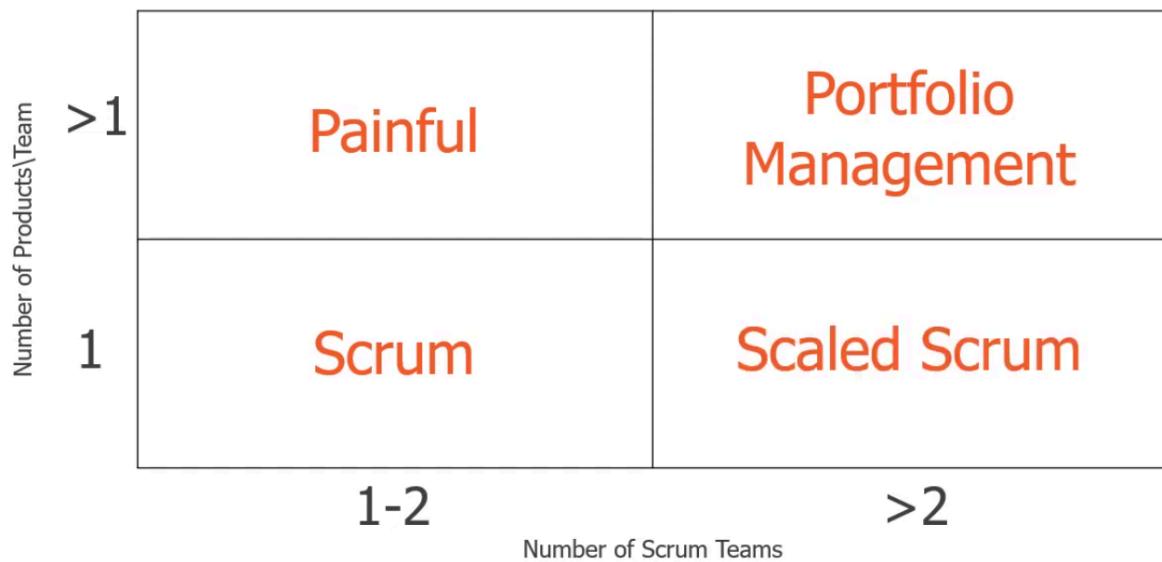
- **Portafolio:** Se utilizan talles de remeras
- **Product Backlog:** Se utilizan story points
- **Sprint backlog task:** Horas de trabajo

Frameworks para escalar SCRUM:

Introducción:

Richard Hundhausen | Professional Scrum Trainer

What is Scaled Scrum



Básicamente, si tenemos más de un equipo de SCRUM trabajando sobre el mismo producto, las cosas se empiezan a complejizar un montón:

- Muchísimas dependencias
- Mucho desperdicio
- Ineficiencias

Dado que Scrum impone un límite en la cantidad máxima de integrantes del Equipo de Desarrollo, surge la necesidad de escalar este Framework para poder gestionar productos de mayor envergadura, ya que las prácticas ágiles se utilizan en ambientes complejos para reducir tal complejidad.

Existen diferentes frameworks para escalar como:

- Nexus
- Scale Scrum
- Less
- Safe

Framework Nexus:

Un Nexus es un grupo de aproximadamente **tres a nueve Scrum Teams** que trabajan juntos para entregar un único producto; es una conexión entre personas y cosas.

Un Nexus tiene un solo Product Owner que gestiona un único Product Backlog con el cual trabajan los Scrum Teams.

Nexus extiende mínimamente el marco de trabajo Scrum solo donde es absolutamente necesario, para facilitar que **múltiples equipos trabajen un único Product Backlog para construir un Integrated Increment** que cubra un propósito

¿Y para qué me sirve Nexus?

El marco de trabajo Nexus ayuda a los equipos a solucionar desafíos comunes de escalado, como **reducir (hacer transparentes) las dependencias entre los equipos, preservando la autogestión y la transparencia de los equipos, y asegurando su responsabilidad.**

A menudo las **dependencias** están relacionadas con:

- Estructura del producto
- Comunicación de los equipos

Roles:

Pequeña aclaración:

Con respecto a los roles, tenemos los mismos tres que antes y ahora se **escala** (se agrega un cuarto) a un NIT. Mira, el Nexus está estructurado de la siguiente manera:

- **Hay múltiples equipos Scrum** (de 3 a 9), cada uno con su propio **Scrum Master** y sus miembros de desarrollo.
- **Hay un solo Product Owner (PO)** que gestiona el **Product Backlog** compartido entre todos los equipos.
- **El Nexus Integration Team (NIT)** Provee una capa de integración al trabajo de los equipos scrum

El NIT NO es un equipo separado. Es más bien un grupo de personas seleccionadas de los equipos existentes para enfocarse en la integración del producto.

La principal función que tiene el NIT es asegurarse que se produzca un incremento integrado de manera que cumpla con una Definition of Done común a todos los equipos

La integración incluye abordar las restricciones técnicas y no técnicas del equipo multifuncional que pueden impedir la capacidad de un Nexus para entregar constantemente un incremento integrado

Se asegura que todos lleguen al mismo consenso de definition of done y definition of ready

El Nexus Integration Team está formado por:

- **Product Owner:** Es responsable de maximizar el valor del producto y el trabajo realizado e integrado por los Scrum Teams en un Nexus, así mismo también es responsable de la gestión eficaz del Product Backlog.
- **Scrum Master:** Responsable de asegurar que el marco de trabajo Nexus se entienda y se promulgue como se describe en la Guía de Nexus. Puede ser un Scrum Master en uno o más de los scrums teams en el Nexus.
- **Miembros de los demás equipos SCRUM**

Ejemplo de NIT:

Contexto:

- El **Equipo X** no realiza pruebas de rendimiento en sus entregas.
- El **Equipo Y** no documenta los endpoints del API.
- El **Equipo Z** usa un formato diferente para los logs.

Entonces los 3 equipos tienen una definition of ready diferente y por lo tanto el producto no puede entrar en la iteración para ser desarrollado

Intervención del NIT:

- Organiza un taller con representantes de todos los equipos para alinear una única definición de **definición de ready**.
- El estándar incluye requisitos mínimos como pruebas funcionales, integración de

código, documentación y cumplimiento de estándares de seguridad.

Eventos:

Nexus agrega o extiende los eventos definidos por Scrum. La duración de los eventos Nexus se guía por la duración de los eventos correspondientes en la Guía de Scrum. Tienen definido un bloque de tiempo adicional a sus correspondientes eventos de Scrum.

- **Nexus Sprint:** Los scrums teams producen un solo Integrated Increment (Incremento de integración), es lo mismo que scrum.
- **Refinamiento del product backlog:** El Refinamiento Entre Equipos del trabajo del Product Backlog reduce o elimina las dependencias entre equipos dentro de un Nexus.

El Product Backlog debe descomponerse para que las dependencias sean transparentes, se identifiquen entre equipos y se eliminan o se minimicen. Esto sirve a un propósito dual:

- Ayuda a los Scrum Teams a prever qué equipo entregará qué elementos del Product Backlog.
- Identificar dependencias entre estos equipos.

Ejemplos: En lugar de que un equipo trabaje solo en backend y otro solo en frontend, ambos equipos podrían manejar funcionalidades completas (frontend + backend) de una sección específica del producto, como "Gestión de usuarios" o "Panel de estadísticas".

Al tener equipos "cross-funcionales" y responsables de módulos completos, se reducen las dependencias técnicas.

- **Nexus Sprint Planning:** El propósito de la Nexus Sprint Planning es coordinar las actividades de todos los Scrum Teams dentro de un Nexus para un solo

Sprint. Los representantes apropiados de cada Scrum Team y el Product Owner se reúnen para planificar el Sprint.

Resultados:

- Objetivo del nexus sprint: Este es el objetivo común para todos los equipos
 - Objetivo del sprint: Si bien cada equipo comparte el nexus sprint goal, puede tener a su vez sus propios sprints goal
 - Nexus Sprint Backlog: Backlog del sprint comun a todos los equipos
 - Sprint Backlog: Cada equipo tiene su propio sprint backlog
- **Nexus Daily Scrum:**
- Se discuten **problemas de integración y se inspecciona el progreso hacia el objetivo de sprint del nexus.**
 - **Solo asisten representantes** de cada Scrum team.
 - **La Daily Scrum de cada Scrum Team complementa la Nexus Daily Scrum creando planes para el día**, centrados principalmente en abordar los problemas de integración planteados durante la Nexus Daily Scrum.
 - Los Scrum teams no solo se reúnen o coordinan en el Nexus daily scrum, sino que durante el día, puede existir comunicación entre equipos con mayor detalle sobre adaptar o planificar el resto del trabajo del sprint.
- **Nexus Sprint Review:** La Nexus Sprint Review se lleva a cabo al final del Sprint para brindar retroalimentación al Integrated Increment terminado que el Nexus ha construido durante el Sprint y determinar adaptaciones futuras.
- La nexus sprint review reemplaza a las sprint reviews de cada scrum team.**
- **Nexus Sprint Retrospective:** El propósito de la Nexus Sprint Retrospective es planificar formas de mejorar la calidad y la eficacia en todo el Nexus.

El Nexus inspecciona cómo fue el último Sprint con respecto a individuos, equipos, interacciones, procesos, herramientas y su Definición de Terminado. Además de las mejoras de cada equipo, las Retrospectivas de los Scrum Teams complementan la Nexus Sprint Retrospective mediante el uso de inteligencia de abajo hacia arriba para centrarse en los problemas que afectan al Nexus en su conjunto.

Artefactos:

Representan trabajo o valor y están diseñados para maximizar la transparencia. El Nexus Integration Team trabaja con los Scrum Teams dentro de un Nexus para garantizar que se logre la transparencia en todos los artefactos y que el estado del Integrated Increment se entienda.

Cada artefacto contiene un compromiso, y estos existen para reforzar el empirismo y el valor de Scrum para el Nexus y sus interesados.

- **Product Backlog:** Hay uno solo que contiene la lista de todo lo que el Nexus y sus Scrum Teams necesitan para mejorar el producto. El Product Backlog debe entenderse con tal nivel que permita detectar y minimizar las dependencias. Es responsabilidad del Product Owner.

El compromiso es el Objetivo del Producto, que describe el estado futuro del producto y sirve como un objetivo a largo plazo para el Nexus.

- **Nexus Sprint Backlog:** Está compuesto del Objetivo del Sprint de Nexus y elementos del Product Backlog de cada Scrum Team del Nexus. Se usa para resaltar las dependencias y el flujo de trabajo durante el Sprint, y se actualiza durante el mismo a medida que se obtiene más conocimiento. Este debe tener un nivel de detalle tal que permita que Nexus pueda inspeccionar su progreso en la Nexus Daily Scrum.

Su compromiso es el Objetivo de Sprint de Nexus, que es un único objetivo para Nexus. Es la suma de todo el trabajo y los Objetivos de Sprint de los Scrum Teams dentro del Nexus. Se crea en la Nexus Sprint Planning y se agrega al Sprint Backlog de Nexus. Los Scrum Teams lo tienen en cuenta a medida que

trabajan. En la Nexus Sprint Review se debe de mostrar la funcionalidad de valor y útil que está terminada para alcanzar el Objetivo del Sprint.

- **Integrated Increment:** Es la suma actual de todo el trabajo integrado completado por un Nexus en relación con el Objetivo del Producto. Se inspecciona en la Nexus Sprint Review pero puede entregarse antes de la misma. Debe cumplir con la definición de terminado.

El compromiso de este artefacto es la Definición de Terminado, que define el estado del trabajo integrado cuando cumple con la calidad y las medidas requeridas para el producto. El incremento se termina sólo cuando está integrado, es de valor y utilizable. Es responsabilidad del Nexus Integration team una definición de terminado que se pueda aplicar en cada sprint, y todos los Scrum Team deben definir y adherirse a esta definición. Cada Scrum Team se autogestiona para llegar a este estado, pudiendo aplicar una definición de terminado más estricta pero nunca usar criterios menos rigurosos de lo acordado para el Integrated Increment.

Framework Less:

LeSS es un marco de trabajo que permite escalar Scrum y se aplica a productos de entre dos a ocho equipos. LeSS propone un sólo Product Owner, un Scrum Master que puede servir de uno a tres equipos, gestionando un único Product Backlog. El Sprint es común para todos los equipos. Si se trabaja con más de ocho equipos, se utiliza LeSS Huge.

LeSS es un marco que trata de aplicar los conceptos y principios de SCRUM a gran escala de la manera más sencilla posible.

¿Cuáles son algunos de los principios de Less?

- **Scrum a gran escala es Scrum:** No se trata de un Scrum nuevo y mejorado. Más bien, LeSS trata de descubrir cómo aplicar los principios, reglas, elementos y propósito de Scrum en un contexto a gran escala, de la forma más sencilla posible.
- **Transparencia:** Basada en tareas “realizadas” tangibles, ciclos cortos, trabajo en equipo, definiciones comunes y eliminación del miedo en el lugar de trabajo.

- Mas con menos:
 - No queremos **más roles** porque más roles implican menos responsabilidad para los equipos.
 - No queremos más artefactos porque **más artefactos** generan una mayor distancia entre los equipos y los clientes.
 - No queremos **más procesos** porque eso conlleva menos aprendizaje y responsabilidad del equipo sobre los procesos.
- Enfoque en el producto integral: Un Product Backlog, un Product Owner, un producto entregable, un Sprint, independientemente de si se trata de 3 o 33 equipos. Los clientes buscan funcionalidad valiosa en un producto cohesivo, no componentes técnicos separados.
- Centrado en el cliente
- Mejorar hasta lograr la perfección
- Teoría de cola: Comprender cómo se comportan los sistemas con colas en el dominio de I+D y aplicar esos conocimientos para gestionar los tamaños de colas, los límites de trabajo en curso, la multitarea, los paquetes de trabajo y la variabilidad.

Roles:

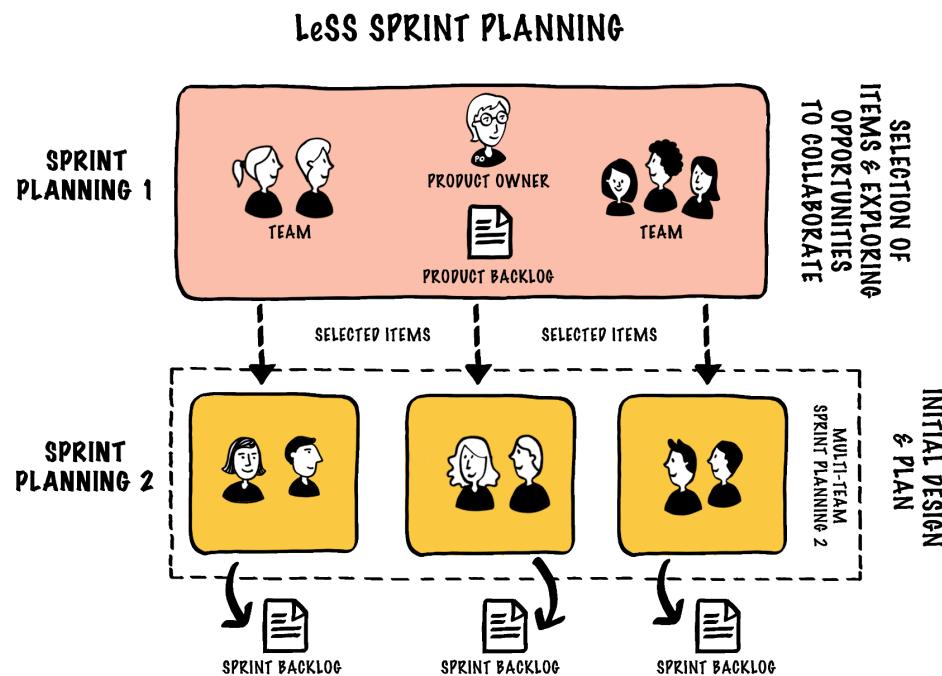
Un *Product Owner* , de dos a ocho *equipos* , un *Scrum Master* para uno a tres equipos. Fundamentalmente, estos equipos son *feature teams* : verdaderos equipos full-stack multifuncionales e inter componentes que trabajan juntos en un entorno de código compartido, donde cada uno se encarga *de todo* para crear *elementos terminados*.

Eventos:

- Less Sprint: Hay un sprint a nivel producto, no un Sprint diferente para cada equipo. Cada equipo comienza y termina el Sprint al mismo tiempo.

- **Sprint Planning One:** es atendida por el Product Owner y los equipos o representantes de estos. Juntos seleccionan tentativamente a los ítems que cada equipo trabajará en ese Sprint. Los equipos identifican oportunidades para trabajar juntos y se aclaran las preguntas finales.
- **Sprint Planning Two:** Básicamente en esta parte cada equipo trabaja para determinar el cómo se va a llevar a cabo las tareas que se les han asignado a los equipos.

Fíjate que va a surgir un **sprint backlog** por cada sprint que realice cada equipo



- **Daily Scrum:** cada equipo tiene su propia Daily Scrum y no hay diferencia con las Daily Scrum para un solo equipo. Tiene una duración de 15 minutos. Y los miembros del equipo deben responder a tres preguntas.
 - ¿Qué hice ayer?
 - ¿En qué voy a trabajar hoy?
 - ¿Qué queda por hacer?

- **Sprint Review:** El propósito de la revisión del Sprint es inspeccionar el resultado del Sprint y determinar futuras adaptaciones. El equipo de Scrum presenta los resultados de su trabajo a las partes interesadas clave y se discute el progreso hacia el objetivo del producto

Se recibe **feedback** proporcionado por los stakeholders (clientes, patrocinadores, etc)

- **Sprint Retrospective:** Al final del Sprint, cada equipo individualmente realiza su propia Sprint Retrospective.
- **Overall Retrospective:** Básicamente es lo mismo que antes pero se hace en conjunto con todos los equipos. Se responden preguntas del tipo:
 - ¿Qué tan bien trabajan juntos los equipos?
 - ¿Están los equipos aprendiendo juntos?
 - ¿Hay algo que hizo un equipo que debería compartirse?

- **Product Backlog Refinement:** En el refinamiento del product backlog (PBR), básicamente se realizan 3 actividades: (1) dividir los elementos grandes, (2) aclarar los elementos hasta que estén listos para su implementación sin más preguntas de "qué", y (3) estimar el tamaño, el valor, los riesgos, etc.

En LeSS, PBR es un **evento** formal esperado (reunión de taller) en lugar de solo una "actividad" no especificada, y se realiza en:

PBR multi equipo:

El PBR multiequipo es posiblemente el **evento más importante** en LeSS para **impulsar sprints efectivos** con equipos bien alineados, coordinados y adaptables

El PBR multiequipo se da cuando **varios equipos están** (literalmente) **en la misma sala al mismo tiempo** realizando el PBR.

Generalmente **incluye a todos** los equipos del grupo de productos (o, en el caso de LeSS Huge, a todos los equipos dentro de un Área de Requerimiento) y, por lo tanto, a **todos los miembros del equipo**.

En el PBR multiequipo se realizan las 3 actividades anteriormente dichas

Artefactos:

- **Product Backlog:** Todos los equipos construyen un solo producto en base a los ítems de un solo Product Backlog. Estos ítems no están pre asignados a los equipos. La definición de producto debe ser tan amplia y centrada en el usuario final/cliente como sea práctico. Con el tiempo, la definición de producto podrá expandirse. El Product Backlog de LeSS es el mismo que en un entorno Scrum de un solo equipo.
- **Sprint Backlog:** Cada equipo tiene su propio Sprint Backlog. Es la lista de trabajo que el equipo debe realizar para poder completar los ítems del Product Backlog. Por lo tanto, el Sprint Backlog es por equipo y no hay diferencia entre un LeSS Sprint Backlog y un Sprint Backlog.
- **Incremento del producto potencialmente enviable:** La salida de cada Sprint es un incremento de producto potencialmente entregable. Es la forma de integrar el trabajo de todos los equipos al finalizar el Sprint. Potencialmente enviable es una declaración sobre la calidad del software y no sobre el valor o comerciabilidad de este. Si el producto se puede enviar realmente dependerá del Definition of Done.

Existe un **DoD** que es común hacia todos los equipos:

- Cada equipo puede tener su propia DoD más detallada, pero siempre expandiendo de la común.
- El objetivo es mejorar la DoD para que resulte en su envío de producto en cada Sprint (o incluso con más frecuencia).

Framework LeSS - Huge:

LeSS Huge es una versión extendida del marco **LeSS (Large-Scale Scrum)** diseñada para organizaciones grandes con **más de 8 equipos** trabajando en un mismo producto.

Mientras que LeSS estándar puede manejar hasta 8 equipos (~50 personas), LeSS Huge permite escalar a cientos o incluso miles de personas sin perder la esencia ágil de Scrum.

Acá la **diferencia sustancial** es que ahora el product backlog se encuentra dividido en **áreas de requisitos** lo que facilita gestión del mismo, entonces como un solo PO no puede gestionar todo el producto se asigna un **Product Owner de Área** por cada área de requisitos

Por otro lado, si bien seguimos trabajando con un solo **producto backlog**, el mismo se descompone en **vistas** según la área en la cual se lo trabaje. Cada vista representa una descomposición más **detallada** del mismo para el área de requisitos que corresponda

Lean - Kanban:

Lean:

Concepto:

LEAN es una **filosofía** y un conjunto de prácticas de gestión y producción que se originó en el sistema de producción de Toyota, conocida como Lean Manufacturing, pero ha sido adaptada a muchos otros campos, incluidos el desarrollo de software y la gestión empresarial.

El **enfoque principal** de Lean es maximizar el valor entregado al cliente mediante la eliminación de desperdicios en los procesos. La idea es hacer más con menos: menos tiempo, menos recursos y menos esfuerzo.

Principios de la filosofía

1. Eliminar desperdicios:

- Evitar que las cosas se pongan viejas antes de terminarlas o evita el re-trabajo

- Eliminar desperdicios significa reducir el tiempo **removiendo lo que no agrega valor**
 - Desperdicio es cualquier cosa que interfiere con darle al cliente lo que él valora en tiempo y lugar donde le provea más valor
 - En manufactura: Es el inventario
 - En software: Es el trabajo parcialmente hecho y las características extra
 - El 20 % del software que entregamos contiene 80 % de valor
- Se relaciona con los principios:
 - 3: Entregar el software frecuentemente
 - 7: El software funcionando es la principal medida de desempeño
 - 10: La simplicidad → Porque todas las funcionalidades parcialmente hechas son desperdicio.

Desperdicios en software	Explicación
Trabajo a medias	<p>Este desperdicio se refiere a cualquier trabajo que está en progreso pero no está completado. El trabajo incompleto o no entregado representa una inversión de tiempo y recursos que aún no ha generado ningún valor.</p> <p><u>Ejemplos:</u></p> <ul style="list-style-type: none"> - Funcionalidades que están parcialmente desarrolladas pero no se han integrado ni probado. - Código que ha sido escrito pero no ha sido desplegado en producción. - Documentación o diseño que no ha sido finalizado.
Características extra	<p>Se refiere a agregar funcionalidades que no son realmente necesarias para el usuario final o que no aportan un valor significativo. Estas características pueden parecer útiles, pero si el cliente no las utiliza, representan un esfuerzo desperdiciado.</p> <p>Esto también tiene que ver con Up Front Specification porque cuando se intenta</p>

	<p>satisfacer esos requerimientos en etapas posteriores de implementación, la información está obsoleta, incompleta o errónea, debido a que cuando se especificó el requerimiento había incertidumbre o falta de información.</p> <p><u>Ejemplos:</u></p> <ul style="list-style-type: none"> - Funciones adicionales en un software que el cliente nunca pidió o usará. Características que podrían haber sido incluidas más tarde, pero que se desarrollaron sin una necesidad clara.
Proceso extra	<p>Se refiere a pasos adicionales o innecesarios en el proceso de desarrollo que no aportan valor directo al producto final. Estos pueden estar relacionados con burocracia, aprobaciones redundantes o pasos innecesarios en el desarrollo o pruebas.</p> <p>Los procesos definidos sobrecargaron mucho a las personas de tareas que no aportan demasiado valor, y como resistencia surgieron los procesos empíricos, que buscan eliminar todos aquellos pasos en los procesos que no aportan valor al desarrollo del producto.</p> <p><u>Ejemplos:</u></p> <ul style="list-style-type: none"> - Producir documentación detallada que nadie utiliza o consulta. - Revisiones de código excesivamente largas y formales, cuando podrían ser más ágiles. - Reuniones innecesarias que no contribuyen directamente a avanzar en el desarrollo.
Defectos	<p>Los errores o fallos en el software son un desperdicio obvio, ya que requieren reprocesamiento para corregirlos. Estos errores pueden ser causados por malentendidos en los requisitos, mala calidad en el desarrollo o pruebas insuficientes.</p> <p><u>Ejemplos:</u></p> <ul style="list-style-type: none"> - Bugs en el código que requieren tiempo adicional para corregir. - Malas implementaciones que no cumplen con los requisitos del cliente. - Fallos en producción que causan interrupciones en el servicio.
Esperas	Este desperdicio ocurre cuando el flujo de trabajo se detiene debido a cuellos de

	<p>botella o retrasos en algún paso del proceso. El tiempo que los equipos pasan esperando bloquea el avance y aumenta el tiempo total de entrega.</p> <p><u>Ejemplos:</u></p> <ul style="list-style-type: none"> - Desarrolladores esperando aprobaciones o decisiones de la gerencia. - Esperar a que los equipos de pruebas verifiquen el código. - Bloqueos mientras se espera la entrega de recursos (como servidores o entornos de prueba).
Movimiento	<p>En el contexto de software, el "movimiento" se refiere al esfuerzo innecesario que deben realizar los desarrolladores, testers o cualquier miembro del equipo para acceder a la información, herramientas o recursos que necesitan para hacer su trabajo.</p> <p><u>Ejemplos:</u></p> <ul style="list-style-type: none"> - Los desarrolladores tienen que buscar en múltiples fuentes para encontrar información relevante (requisitos, especificaciones, etc.). - Moverse entre diferentes entornos de trabajo o herramientas sin integración entre ellas.
Transporte	<p>En software, este desperdicio se refiere a la transferencia de información o entregables entre diferentes equipos o departamentos, lo cual puede provocar malentendidos o demoras. A veces se relaciona con la falta de integración en el equipo.</p> <p><u>Ejemplos:</u></p> <ul style="list-style-type: none"> - Información crítica que se transfiere entre varios equipos. Causando pérdida de contexto o tiempo - Desarrolladores que deben esperar hasta que otro equipo entregue algo (como bases de datos o entornos de prueba).
Talento no utilizado	<p>Este desperdicio ocurre cuando no se aprovechan las habilidades, conocimientos o creatividad del equipo. Los miembros del equipo tienen más capacidad para contribuir, pero no se les da la oportunidad, ya sea debido a una estructura jerárquica rígida o a falta de comunicación y participación.</p> <p><u>Ejemplos:</u></p>

	<ul style="list-style-type: none"> - Desarrolladores que se limitan a seguir órdenes sin participar en la toma de decisiones o sugerir mejoras. - Falta de oportunidades para que el equipo proponga ideas innovadoras o mejoras en los procesos. - Subutilización de las habilidades de los empleados, como asignar tareas simples a personas altamente capacitadas.
--	--

Los **desperdicios** también pueden ser entendidos no sólo en términos de software, sino que en términos de **manufactura**:

- Inventario
- Sobreproducción
- Proceso extra
- Transporte
- Esperas
- Movimiento
- Defectos
- Talento no utilizado

2. Amplificar el aprendizaje:

- Crear y mantener una cultura de mejoramiento continuo y solución de problemas
- Un proceso focalizado en crear conocimiento esperará que el diseño evolucione durante la codificación y no perderá tiempo definiéndolo en forma completa, prematuramente
- Esta cultura de mejoramiento continuo., Debe permitir a los individuos intercambiar sus **experiencias** y conocimientos para contribuir al aprendizaje colectivo
- Relacionado con:
 - Con el principio de **transparencia** del empirismo (Ágil)
 - 6: La mejor forma de transmitir información es mediante el *cara a cara*
 - Lean UX, justamente con esto se genera conocimiento

3. Integridad conceptual:

- Encastar todas las partes del producto o servicio, que tenga **coherencia** y **consistencia** (tiene que ver con los requerimientos no funcionales).
 - **Coherencia:** Significa que las partes de tu producto siguen las mismas normas, principios o reglas. Por ejemplo que toda mi app siga los mismos principios de diseño
 - *En mi app mobile todos los botones tienen el mismo estilo*
 - **Consistencia:** El producto se comporta de manera predecible en distintos contextos
 - *Si el flujo de registro en la versión mobile tiene 3 pasos en la versión web no debería tener 5*
- El **objetivo es construir con calidad desde el principio**, no probar después.
→ La calidad debe ser **embebida**
- No se deben realizar inspecciones luego de que los defectos ocurran sino **antes**, justamente para evitar que aparezcan
- **Las Definition of Done / Ready** son un claro ejemplo de cómo Ágil adopta este principio para lograr consistencia
- Consistencia y coherencia se pueden lograr también utilizando prácticas como **Continuos Integration o TDD**
- Ahora que lo pienso esto tambien se podria relacionar con el principio de **desarrollo sostenible**, ya que justamente describe a un conjunto de principios por los que desarrollamos software

4. Diferir compromisos:

- Esto apunta a postergar la toma de decisiones lo suficientemente como para poder tener la mayor información necesaria para tomar esa decisión, pero a la vez, antes de que ese momento sea muy tarde (diferir la decisión irreversible).
 - *Tomar las decisiones en el momento justo*
- Esto se puede reflejar en Agile con el Product Backlog, donde nunca se tiene el 100% de los requerimientos, sino que se va completando a medida que se tiene más información sobre lo que se necesita implementar.
- Se relaciona con el concepto de Just In Time de los requerimientos ágiles

5. Dar poder al equipo.

- Otorga libertad de acciones y poder de decisión al equipo. Para ello, el equipo debe ser **multifuncional y autogestionado**, y sus miembros deben estar capacitados y motivados.
- Esto implica no subordinar por parte de los superiores, debido a que esto anula las capacidades intelectuales, entrenar líderes, delegar decisiones y fomentar buena ética laboral.
- Este aspecto termina siendo un talón de Aquiles en las filosofías Agile y Lean, ya que en la práctica pocas veces se logran reunir todas estas características.
- Se relaciona con los principios:
 - 5: Individuos motivados
 - 11: Individuos autoorganizados

6. Ver el todo:

- En Lean se busca tener una visión holística, que permita asociar y comprender el todo, el producto, el valor agregado que hay detrás, el servicio que brinda el complemento de los productos, más allá de los objetivos particulares por áreas o gerencias, porque la idea es que los

objetivos particulares de las partes estén alineados con los objetivos globales.

7. Entregar los antes posible

- La idea es entregarle al cliente software funcionando lo más rápido posible y que este producto sea útil para él. Se habla de entregar software antes de que las necesidades de negocio cambien, porque el ambiente donde se desempeña el mismo cambia.
- Para esto Lean propone acotar ciclos de desarrollo (principio 1 de Agile: satisfacer al cliente con entregas tempranas) y entregar rápidamente incrementos pequeños de valor, lo que permite salir pronto al mercado con un producto mínimo que sea valioso (principio ágil de entregas tempranas y frecuentes de software de valor para el cliente).
- Se relaciona con el principio 3 (Entregas frecuentes del producto)

¿Pero entonces cuál es la relación entre Lean y Ágil?

Lean es previo al Manifiesto Ágil. Algunos principios de Agile heredan de los fundamentos de Lean. Ambos están orientados al cliente, a proveer el máximo valor posible.

- Para esto, entienden que los individuos de los equipos deben estar motivados y autoorganizados, de forma tal que la sinergia de este facilite el desarrollo del proyecto. Esta sinergia, se interpreta como un valor agregado para el cliente.
- La flexibilidad para adaptarse a los cambios y ofrecer valor al cliente, es también un denominador común de ambos enfoques.
- Otro aspecto que tienen en común es generar productos de calidad y de mejora continua.

Kanban:

"Kanban" (看板) significa "tarjeta" o "letrero visual

Concepto:

Kanban no es un framework o proceso, es un método para gestionar y mejorar **servicios profesionales** (especialmente en entornos de conocimiento como desarrollo de software, marketing, recursos humanos, etc).

Esto lo logra a través del principio de **empieza por donde estés**, es decir, no plantea introducir cambios revolucionarios, sino que introduce mejoras graduales a un proceso de desarrollo de software **ya existente** en una organización.

Kanban toma algunos varios de los conceptos más importantes de Lean como por ejemplo:

- La cultura de mejoramiento continua
- Generar valor y eliminar los desperdicios

Kanban trabaja con **esquema pull no push**:

- En un sistema **pull**, las tareas o trabajos se "jalan" hacia la siguiente etapa solo cuando hay capacidad disponible.
- En un sistema **push**, el trabajo se "empuja" hacia adelante sin necesariamente considerar si la siguiente etapa tiene espacio o recursos para hacerlo.

Nota: Fijate que decimos que Kanban no es una metodología o marco de trabajo como scrum porque kanban no especifica roles, ceremonias ni artefactos es solamente un enfoque para la mejora de procesos

¿Cuáles son los valores de Kanban?

- Transparencia
- Equilibrio
- Colaboracion

- Foco en el cliente
- Flujo
- Liderazgo
- Entendimiento
- Acuerdo
- Respeto

Prácticas generales de Kanban:

1. Visualizar

- El método Kanban utiliza un **mecanismo de señalización** para hacer visible el trabajo que es requerido por el cliente.

Para ello, divide el trabajo en **piezas de trabajo** (que pueden ser U.S., Features, bugs, temas, épicas, cambios, etc.) y las **escribe** en **tarjetas** señalizadoras (kanban) que serán ubicadas en tableros kanban.

- Existen 2 tipos de columnas:
 - **Producción:** Piezas sobre las que se está trabajando
 - **Acumulación:** Piezas que están listas para pasar a la siguiente etapa (sistema de arrastre)
- Habilita la **cooperación**, ya que todo el mundo tiene la misma imagen

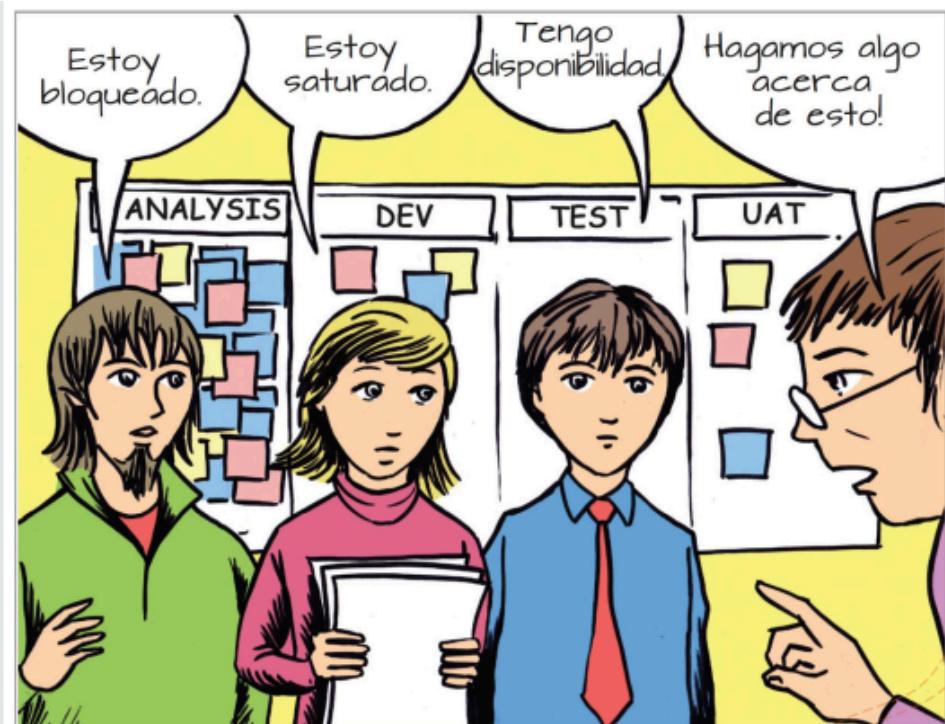
2. Limitar el WIP (Work In Progress)

Limitar el WIP (Work In Progress) en Kanban es una de las prácticas más poderosas y esenciales del método. Sirve para controlar la cantidad de trabajo que se encuentra en proceso al mismo tiempo.

Limitar el WIP sirve principalmente para:

- Evitar la sobrecarga del equipo
- Mejorar el flujo de trabajo

Limitar el WIP promueve **conversación y mejora**



- Las políticas para limitar el WiP crean un sistema de arrastre (pull): el trabajo es “arrastrado” al sistema cuando otro de los trabajos es completado y queda capacidad disponible, en lugar de “empujar” estos trabajos al paso siguiente cuando hay nuevo trabajo demandado.

3. Gestionar el flujo de trabajo:

- El objetivo es poder terminar el trabajo de la forma más fluida y predecible posible, mientras se mantiene un **ritmo sostenido**.
- Se analizan **variables** que afectan al workflow:
 - Cuellos de botella
 - Recursos ociosos
 - Tiempo de entrega
 - Tiempo de espera

4. Hacer las políticas explícitas

- Las políticas en kanban son básicamente una serie de **reglas** que me permiten gestionar el flujo de trabajo
- Las políticas de proceso deben ser **escasas, simples, estar bien definidas, visibles, deben aplicarse siempre, y tienen que ser fácilmente modificables**. Es una buena práctica poder cambiar fácilmente las políticas, ya que si producen efectos contraproducentes para nuestro proceso o también se considera que no pueden aplicarse las mismas, deben poder cambiarse.
- También es importante **visualizar las políticas**; por ejemplo, colocando resúmenes entre las columnas donde se describe lo que debe estar hecho antes de que una tarjeta se mueva de una columna a la siguiente. También se suele colocar el WIP de cada columna.
- Las políticas de calidad o DoD deben estar definidas, publicadas y promovidas para lograr no sólo que se cumplan, si no buscar mejorar continuamente.

5. Establecer ciclos de retroalimentación

- Son una parte esencial para cualquier proceso controlado que nos ayuda a realizar cambios evolutivos. Se debe definir con qué frecuencia se realizan las **reuniones**, donde depende en qué contexto se presentan ya que es importante para el resultado.
- Si se realizan revisiones demasiadas frecuentes pueden obligar a cambiar cosas que no se vieron con los cambios anteriores, pero si no son demasiadas frecuentes, existe un bajo rendimiento durante mucho tiempo.

6. Mejorar y evolucionar:

- Kanban propone una cultura de mejora continua, donde no introduce cambios significativos en los procesos de desarrollo, sino que identifica ese proceso, lo visualiza (hacer explícito para todos los miembros) y propone mejoras sobre él, las cuales se van aplicando de forma gradual a lo largo del tiempo.
-

Para cerrar con los principios es importante entender que para que Kanban funcione, lo que nosotros tenemos que hacer es mapear el proceso a un conjunto de colas de trabajo

Entonces supongamos que nosotros tenemos una **solicitud de pedido**. Bueno este va a ir pasando por cada una de las colas que representan las partes del proceso, y cada una de estas partes del proceso tendrá que hacer un **aporte** a este producto o servicio final a ser entregado

¿Cómo implementamos kanban?

1. Empezar con lo que se tiene: entender el **proceso de desarrollo** actual que se está utilizando.
2. **Dividir trabajo en piezas**
3. **Definir tipos de trabajo**, ya que cada uno tiene una forma distinta de tratamiento, asignando capacidad en función de la demanda

- **Requerimientos:**

- Casos de uso
- Historias de usuario
- Porciones de casos de uso
- Características

- **Defectos:**

- Defectos en producción

- Defectos

- **Desarrollo:**

- Mantenimiento
- Refactorización
- Actualización de infraestructura

- **Solicitudes**

- Solicitud de cambio
- Sugerencia de mejora

4. **Visualizar el flujo de trabajo** diseñando un tablero representando el flujo de trabajo a través de columnas.

- Utilizar nombres en las columnas para ilustrar dónde está cada ítem en el flujo de trabajo
- Distribuir el trabajo en las columnas; el trabajo fluye de izquierda a derecha en las columnas

Cola de Product o	Análisis		Desarrollo		Listo para Build	En Testing		En Producc ión
	En progr eso	Hecho	En progreso	Hecho		En Progreso	Listo para Despliegue	
Casos de Uso 60 %								
Mantenimiento 30 %								
Defectos 10 %								

Nota:

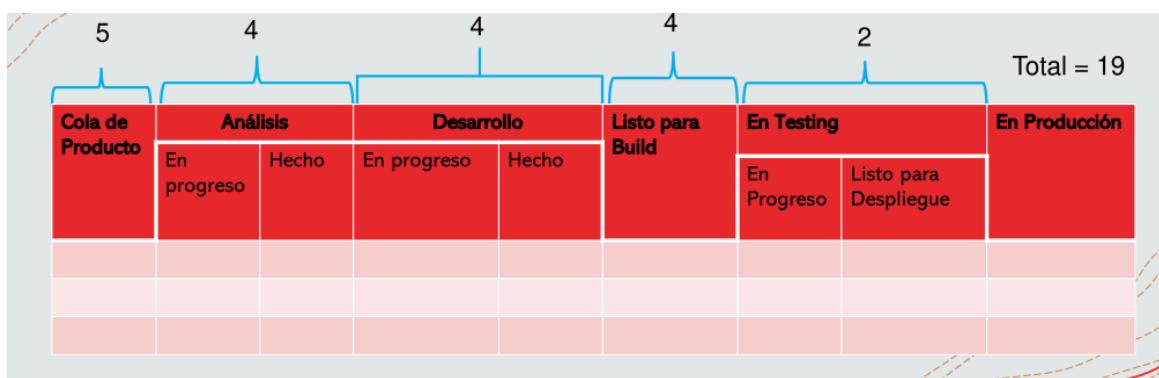
Quizás te estés preguntando qué es lo que se realiza en la columna de **análisis**, y la verdad es que se hacen varias actividades:

- Espacio para **entender** el trabajo a realizar. El equipo le puede preguntar al product owner si tiene dudas con respecto a los requisitos
- Se puede dividir el trabajo en varias tareas grandes
- Se estiman variables como el esfuerzo y se identifican riesgos

Otra cosa es que en la parte de **build** es donde el código ya pasa a ser un **artefacto** por así decirlo

5. Hacer explícitas las políticas

- Todas las políticas deben ser acordadas entre todas las partes involucradas, incluyendo a los clientes, interesados y trabajadores responsables del trabajo que está en el tablero.
- Esto implica acordar los WiP para cada columna, asignar color a las tarjetas kanban, capacidad de trabajo destinada, fechas de entrega, etc.
- Básicamente estamos definiendo **clases de servicio**



6. Identificar cuellos de botella y resolverlos (asignando más recursos o ajustando el WiP donde corresponda).

Ejemplo god:

Supongamos que queremos definir una clase de servicio que se llama **espresso**, entonces le definimos las siguientes políticas:

- Color de tarjeta: Blanco
- Wip: 1
- Los demás trabajos se ponen en espera
- Se puede exceder el límite de WIP para procesar este trabajo (Esto quiere decir que si tenemos un maximo de 20, pero justo llega una pieza de trabajo expresso, entonces vamos a poder exceder este límite)
- La capacidad no se reserva
- Se ser necesario se hace una entrega especial, para ponerla en proporción

Otra política que podríamos utilizar para clase de servicio, podría ser **fecha fija**:

- Color de tarjeta: Rosa
- Deben adherirse al WIP definido
- Fecha de entrega en la parte superior
- Permanecen en cola hasta que sea conveniente que ingresen
- Si se retrasa y la fecha de entrega está en riesgo, puede moverse a la clase de servicio de tipo **expresso**
- Son entregados en entregas programadas cuidando la fecha de entregas

El último ejemplo que podríamos tener es el de la clase de servicios con tipo de política **estandar**:

- Color de la tarjeta: Amarillo

- Deben adherirse al WIP definido
- Son priorizados y puestos en la cola con un mecanismo definido basado en el valor de negocio
- Utilizan la técnica FIFO, si no hay expressos o con fecha fija
- Pueden analizarse por tamaño, en orden de magnitud
- Son entregadas en entregas programadas

Eventos/Cadencias de Kanban:

Eventos	Ejemplo de frecuencia	Propósito
Team Meeting	Diaría	Observar y seguir el estado y flujo del trabajo (no de los trabajadores) ¿Como podemos entregar los elementos de trabajo más rápido en el sistema? ¿Hay capacidad disponible? ¿Qué debemos tomar a continuación?
Team Retrospective	Quincenal o mensual	Reflexionar sobre cómo el equipo gestiona su trabajo y cómo pueden mejorar
Replenishment Meeting	Semanalmente a demanda	Seleccionar los elementos del product backlog que están listos para entrar al flujo de trabajo

Métricas utilizadas en distintos enfoques:

Enfoque tradicional:

Considero útil recordar que en este contexto se entiende por **métrica** a un valor numérico que nos aporta visibilidad sobre algún aspecto en particular, ya sea producto, proyecto, proceso, etc.

Acá nos valemos del concepto de **dominio de las métricas**:

- **Métricas de proceso:**

Es muy importante entender que las métricas de proyecto se consolidan para que luego vos puedas crear métricas de proceso las cuales al ser públicas me permiten compartir la experiencia a lo largo de la organización y generar visibilidad tal y como lo plantean los procesos definidos que lo que quieren es lograr esta **repetición y previsibilidad**

Son públicas y se sobre ellas debe haber un proceso de **despersonalización de los datos** (a menudo sobre métricas de proyecto), para obtener un número aislado y tener una métrica sobre el proceso de la organización en general, como un todo. También generar un promedio

Entonces para que las metricas de proceso pueden ser publicas necesitan cumplir con la característica llamada **no atribución**, es decir que no pueden estar vinculadas a un producto o un proyecto en particular

Son responsabilidad del **Ingeniero de Procesos**, quien realiza las mediciones y luego publica los datos para que todos los empleados de la organización puedan acceder a ellos.

Ejemplos:

- Desviación organizacional de estimaciones es del 80 %
- Defectos por severidad en productos de la organización
- Errores previos a releases por proyecto
- Defectos detectados por usuarios
- Esfuerzo realizado

- **Métricas de proyecto:**

Están enfocadas a los recursos que se dedican al proyecto, como costos, esfuerzos, estimaciones y tiempo.

Son responsabilidad del **Líder de Proyecto** y permiten al equipo adaptar el desarrollo de los proyectos y de las actividades técnicas. Estas métricas son **privadas** de ese proyecto y solo son visibles para los involucrados en el mismo.

Se utilizan para mejorar la planificación del desarrollo, generando ajustes que eviten retrasos, reduzcan riesgos potenciales y por lo tanto, problemas.

Además, se utilizan para evaluar la calidad de los productos en todo momento y en caso de ser necesario, modificar el enfoque para mejorar la calidad, minimizando defectos, retrabajo y por ende el costo total del proyecto.

Las métricas de proyecto se consolidan con el fin de crear métricas de procesos que sean públicas para la organización de software como un todo.

Algunos ejemplos pueden ser:

- Eficiencia de estimación de proyecto
 - Costos estimados versus costos reales
 - Esfuerzo / Tiempo por tarea del Ingeniero de Software
 - Errores no cubiertos por hora de revisión
 - Fechas de entregas reales versus programadas
 - Cantidad de cambios y sus características
-
- **Métrica de producto:**

Están enfocadas en lo que se construye, son responsabilidad del equipo de desarrollo y de Testing y son particulares de ese producto

Se utilizan con propósitos técnicos y tienen como objetivo generar indicadores en tiempo real de la eficacia del análisis, el diseño, la estructura del código, la efectividad de los casos de prueba y calidad del software a construir.

Se deben controlar los artefactos resultantes del proceso de desarrollo (componentes y modelos) para garantizar:

- Que cumplen con los requerimientos del cliente
- Que cumplen con los requerimientos de calidad
- Que estén libres de errores
- Que se realizaron bajo los procedimientos de calidad

Ejemplos:

- Cantidad de líneas de código del producto
 - Defectos por severidad de un producto
 - Promedio de métodos por clase
 - Cantidad de métodos de cada clase
-

¿Cuáles son las métricas básicas?

- Tamaño del producto
 - *Producto*
- Esfuerzo
 - *Proyecto*
- Tiempo (Calendario)
 - *Proyecto*
- Defecto
 - *Producto*

Con respecto a métricas de **tamaño**:

- Lineas de código (LOC)
- Cantidad de requerimientos funcionales, casos de uso o historias de usuarios
- Etc

Con respecto a métricas de **esfuerzo**:

- Horas/Hombre por tareas
- Horas/Hombre destinadas a retrabajo

Con respecto a métrica de **tiempo**:

- Tiempo de entrega vs Tiempo planificado
- Lead time

Con respecto a la métrica de **defectos**:

- Cantidad de defectos por funcionalidad
- Cantidad de defectos que se generan por sprint

Enfoque Ágil:

Antes que nada algunas **aclaraciones**:

- La medición es una salida, no una actividad
- Medir lo que sea necesario y nada más

Dos principios agiles que guian la elección de métricas:

1. Nuestra mayor prioridad es satisfacer al cliente por medio de entregas tempranas y continuas de software valioso
7. El software trabajando es la principal medida de progreso

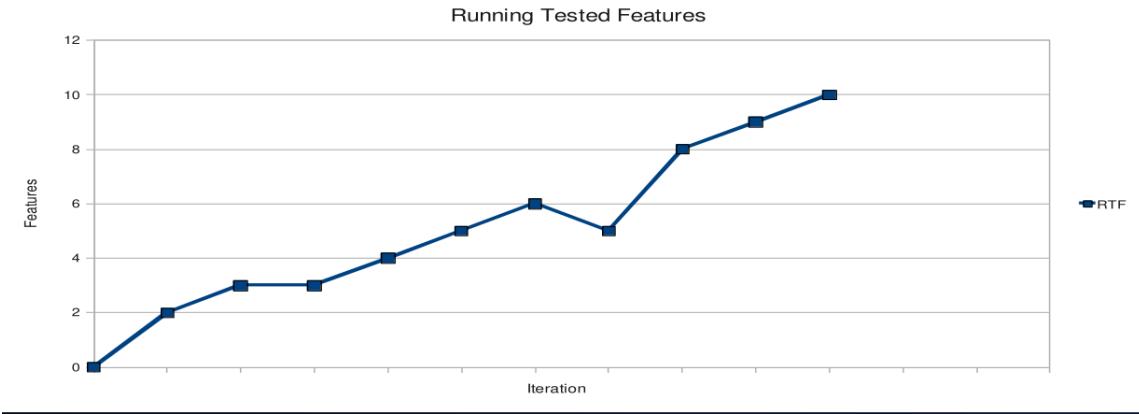
1. Running Tested Features (RTF):

Básicamente esta métrica es **muy poco utilizada**

Mide la cantidad de **features testeadas** que están **funcionando**, es decir, cuantas piezas de producto (historias de usuarios, casos de usos, requerimientos) se terminaron y están en ejecución (es decir que **están en producción**)

El problema de esta métrica es que no tiene en cuenta la complejidad de las piezas de producto que se implementan

Por ejemplo, si se toma como piezas de producto a historias de usuario, se mide cuántas historias de usuarios están funcionando, pero no se sabe de cuantos puntos de historias tiene cada una de ellas. Es una medición absoluta.



2. Capacidad del equipo en un Sprint:

Esta es **una de las tres** métricas que se utilizan en Scrum. Particularmente la capacidad es la **única que se puede estimar**

Es utilizada para poder ver cuánto compromiso de trabajo el equipo puede asumir dentro de un sprint. Entonces con esta capacidad se contrastan las stories del product backlog y así se determinan cuántas se pueden pasar al sprint backlog

Justamente la capacidad del equipo es una de las principales actividades que realizamos en el **sprint planning**

Cuando nosotros calculamos **capacidad** lo podemos hacer de dos formas:

- Horas de trabajo ideales
- Puntos de historia

¿Cómo se calcula la capacidad del Equipo en un Sprint?

Persona	Días disponibles (sin tiempo personal)	Días para otras actividades Scrum	Horas por día	Horas de Esfuerzo disponibles
Jorge	10	2	4-7	32-56
Betty	8	2	5-6	30-36
Simón	8	2	4-6	24-36
Pedro	9	2	2-3	14-21
Raúl	10	2	5-6	40-48
Total				140-197

3. Velocidad (Velocity)

La velocidad no es estima, **se calcula**

Es una medida del progreso de un equipo. Se calcula sumando el número de story points (asignados a cada user story) que el equipo completa durante una iteración

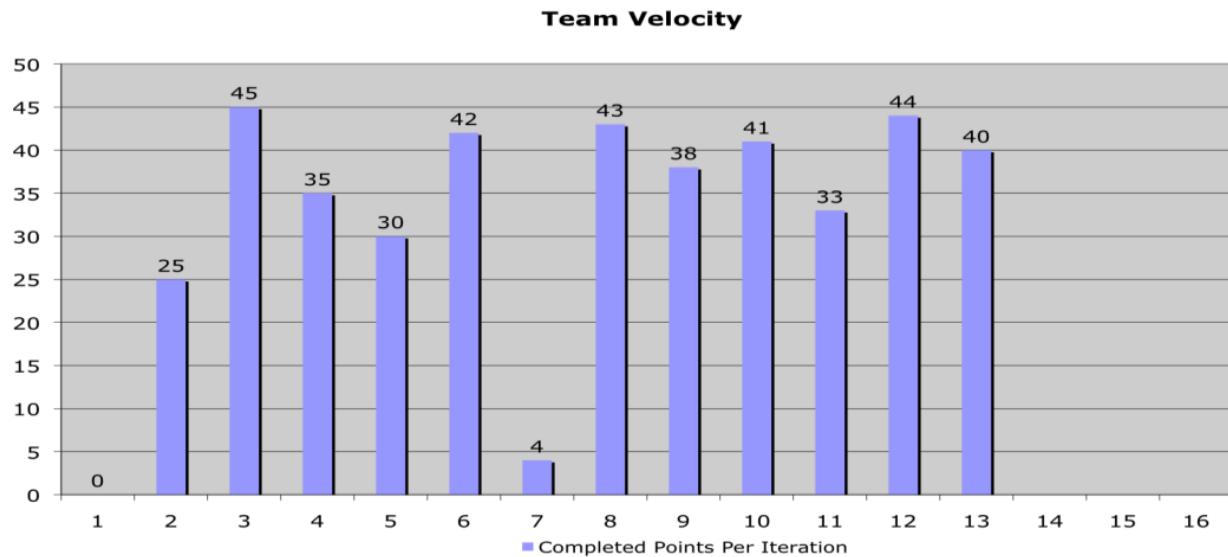
Se cuentan los story points de las user stories que se encuentran **completadas**, no las que se encuentran parcialmente completadas

La velocidad es útil porque **corrige** errores de estimación

Se supone que cuando el equipo aprende a trabajar en conjunto, el gráfico de la velocidad debería **estabilizarse**

La velocidad va a ser calculada en la **sprint review**

$$\text{Velocidad del Sprint} = \sum (\text{Puntos de historia de ítems "Done"})$$



¿Y para qué me sirve esto?

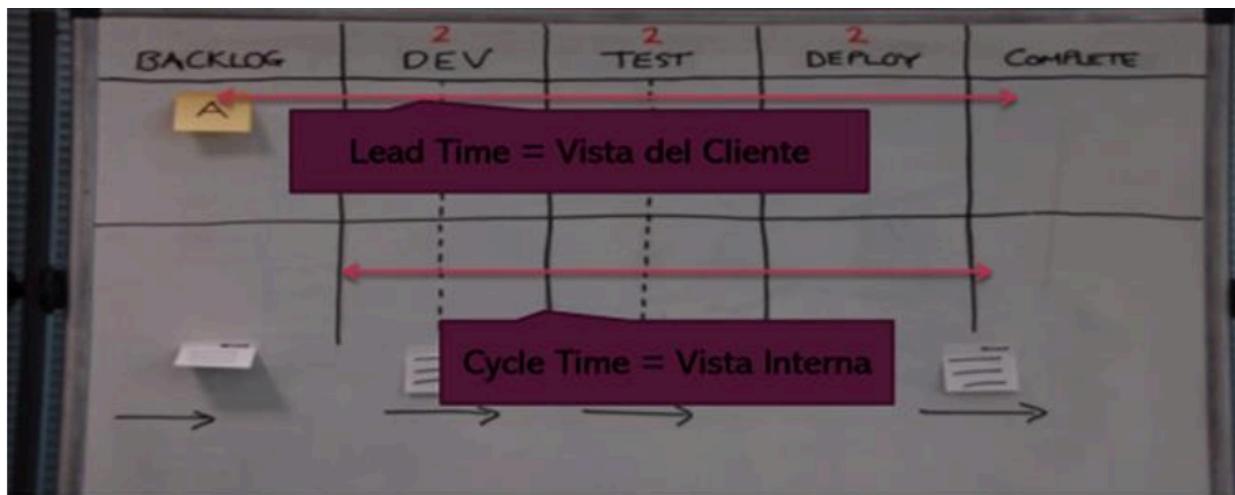
Supón que un equipo trabaja en 4 sprints consecutivos y termina los siguientes puntos de historia:

- **Sprint 1:** 18 story points
- **Sprint 2:** 22 story points
- **Sprint 3:** 20 story points
- **Sprint 4:** 14 story points

$$\text{Velocidad promedio: } \frac{18 + 22 + 20 + 14}{4} = 18,5$$

Esto es clave para ayudarte a planificar, porque para el próximo sprint no deberías comprometerte a realizar más de 18 story points

Métricas en Kanban:



Cycle Time (Tiempo de ciclo):

Es la métrica que registra el tiempo que sucede entre el inicio y el final del proceso, para un ítem de trabajo dado. Se suele medir en días de trabajo o esfuerzo

Medición más mecánica de la capacidad del proceso

¿Qué mide? → Ritmo de terminación al cliente

Lead Time (Tiempo de entrega):

Es la métrica que registra el tiempo que sucede entre el momento en el cual se está pidiendo un ítem de trabajo (entra al backlog) y el momento de su entrega (el final del proceso).

Se suele medir en días de trabajo.

¿Qué mide? → Ritmo de entrega al cliente

Touch Time (Tiempo de tocado):

El tiempo en el cual un ítem de trabajo fue realmente trabajado (o "tocado") por el equipo dentro de el cycle time.

- Esto quiere decir que pudo haber estado 25 días en ciclo pero 2 de esos 25 días, estuvo en estado *pendiente de trabajo*, es decir nadie la tocó

Cuántos días hábiles pasó este ítem en columnas de "trabajo en curso", en oposición con columnas de cola / buffer y estado

Acordate que dentro de cada columna tenías dos tipos de colas (las de ejecución y las de acumulación), entonces lo que hace touch time sería enfocarse en la columna de ejecución

$$\text{Touch Time} \leq \text{Cycle Time} \leq \text{Lead Time}$$

Eficiencia del ciclo proceso:

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

Métricas orientadas a servicio:

- Expectativa de nivel de servicio que los clientes esperan
- Capacidad del nivel de servicio al que el sistema pueden entregar
- Acuerdo de nivel de servicio que es acordado con el cliente
- Umbral de la adecuación del servicio el nivel por debajo del cual este es inaceptable para el cliente

Unidad 3: Gestión del Software como Producto

Software Configuration Management (SCM)

Introducción

Acordate que al principio habíamos visto que la ingeniería de software contaba con 3 disciplinas; bueno, el software configuration management sería la **disciplina de soporte** → Es una actividad paraguas, transversal a todo el proyecto, relevante para el producto a lo largo de su ciclo de vida.

La SCM nos permite:

- Identificar características técnicas y funcionales de ítems de configuración
- Documentar estas características técnicas y funcionales
- Controlar cambios de dichas características
- Registrar y reportar estos cambios
- Verificar trazabilidad de los requerimientos: Cuando hablamos de trazabilidad de requerimientos hacemos referencia a la capacidad de **rastrear** este requerimiento a lo largo de todo el ciclo de vida del proyecto.

Tiene sus **orígenes** a mediados de 1950 's, como CM (por Configuration Management) originalmente utilizado para desarrollo de hardware y control de producción, fue utilizado en el desarrollo de software.

¿Por qué deberíamos gestionar la configuración del software?

Su **propósito** es establecer y mantener la **integridad** de los productos de software a lo largo de su ciclo de vida

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.

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

- Satisface las necesidades de los usuarios
- Permite cumplir con las expectativas:
 - De costos
 - De performance
- Permite que el software sea completamente rastreado durante su ciclo de vida

¿Qué problemas surgen en el manejo de componentes?

- Pérdida de un componente → *Donde está este componente!!*
- Pérdida de cambios → *Esta no es la última versión del componente, otra vez se perdió dioss !!!*
- Sincronía fuente - objeto - ejecutable
- Regresion de fallas
- Doble mantenimiento → *Gastamos una banda de guita manteniendo dos versiones del mismo componente de mierda*
- Superposición de cambios
- Cambios no validos

Algunos conceptos de SCM

- Ítem de configuración
- Repositorio
- Linea base
- Ramas
- Configuración de software

Ítem de Configuración de Software (SCI):

Se llama ítem de configuración (IC) a todos y cada uno de los **artefactos** que forman parte del producto o del proyecto, que pueden sufrir cambios o necesitan ser compartidos entre los miembros del equipo y sobre los cuales necesitamos conocer su estado y evolución.

Correlación → Principio de transparencia

Algunos **ejemplos** de SCI puede ser:

- Planes:
 - SCM
 - Iteracion
 - Integracion
 - Desarrollo
- Riesgos
- Código fuente
- Graficos e iconos
- Prototipo de interfaz
- Propuesta de cambio

¿Y qué se incluye en un plan de SCM?

- Reglas de nombrado de los ítems de configuración
- Herramientas a utilizar para SCM
- Roles e integrantes del comité

- Procedimiento formal de cambios
- Plantillas de formularios
- Como se lleva a cabo la auditoría

Nota: En SCM las reglas de nombrado son normas predefinidas que indican cómo se deben nombrar los elementos versionados dentro del sistema de control de versiones o repositorio.

Algunos ejemplos son:

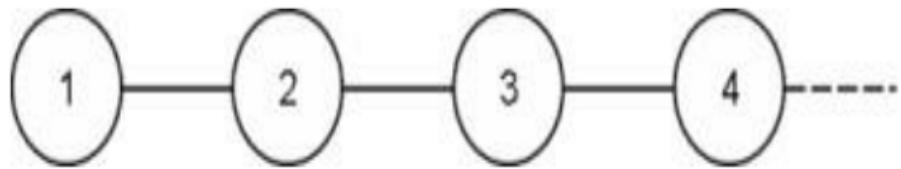
- Archivos fuente (`login.component.ts`)
- Ramas (`feature/login-ui`)
- Tags o etiquetas de versión (`v1.2.3`)
- Builds (`build_20250801_001`)
- Releases (`release-customer-portal-v2`)
- Directorios (`/docs/arquitectura/v1/`)

Versión:

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.



Evolución lineal de un ítem de configuración

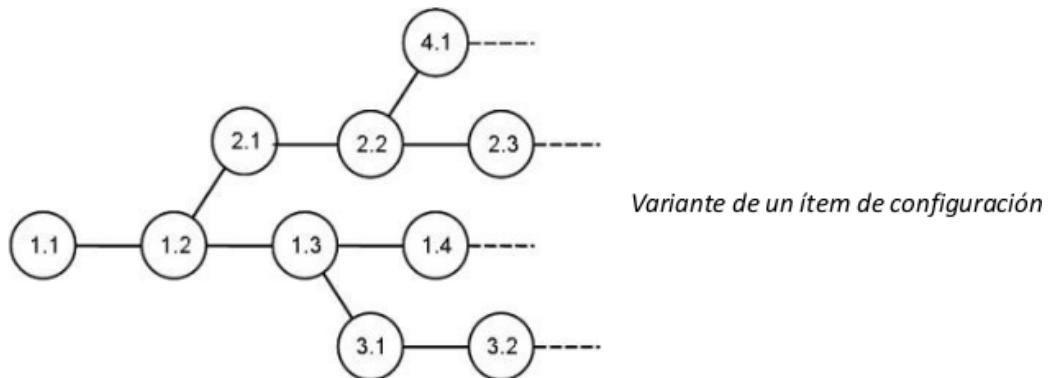
Variante:

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 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.



Variante de un ítem de configuración

Ejemplo para mayor entendimiento:

Escenario: Desarrollo de una aplicación móvil

1. Ítem de configuración

El archivo `LoginModule.java` es un ítem de configuración. Es un módulo que gestiona la autenticación de usuarios dentro de la app.

2. Versión:

- `LoginModule.java v1.0`: Módulo básico que permite loguear usuarios con usuario/contraseña.
- `LoginModule.java v1.1`: Se le agrega validación de email.
- `LoginModule.java v1.2`: Ahora incluye autenticación con Google.

3. Variante:

Supongamos que la app debe funcionar para dos clientes diferentes

- Variante A: `LoginModule.java v1.2` para la app estándar con autenticación Google.
- Variante B: `LoginModule.java v1.2 - Corporate` adaptada para un cliente corporativo que además de los requerimientos anteriores, se agrega una autenticación biométrica

Repositorio:

Es un **contenedor** de ítems de configuración, se encarga de mantener la historia de cada ítem con sus **atributos y relaciones**.

- Usado para hacer **evaluaciones de impacto** de los cambios propuestos
- Puede ser una o varias bases de datos

Con respecto al **funcionamiento** de un repositorio, es bastante sencillo. Básicamente, tenemos el repositorio fuente, a partir del cual se puede realizar 2 acciones:

- **Extracción (Check out):** Es el proceso mediante el cual un desarrollador toma una copia del código fuente desde el repositorio hacia su entorno local de trabajo.

Esto sería análogo al **git pull**

- **Devolución (Check in):** Una vez que el desarrollador ha realizado cambios en el código, devuelve esos cambios al repositorio.

Esto sería análogo al **git push**

Entonces habría una actividad intermedia entre que uno construye los datos en su entorno de trabajo y los envía nuevamente al repositorio, lo cual sería básicamente hacer un **git commit**

Repositorios centralizados:

- Un servidor contiene todos los ítems de configuración con sus respectivas versiones
- Los administradores tienen mayor control sobre el repositorio
- Los desarrolladores no tienen una copia de todos los archivos en su entorno local, solo los archivos con los que ellos trabajan
- Si falla el servidor *estamos al horno*

Repositorios descentralizados:

- Cada cliente tiene una copia exactamente igual del repositorio completo
- Si un servidor falla es solo cuestión de *copiar y pegar*
- Posibilita otros workflows, cosa que no está disponible en el modelo centralizado

Ramas

Es un conjunto de ítems de configuración con sus correspondientes versiones, que permiten **bifurcar el desarrollo** de un software, por varios motivos:

- Experimentacion
- Resolución de errores en el desarrollo
- Desarrollar un mismo software para diferentes plataformas
- Nueva propuesta

Integración de ramas:

- La operación se conoce como **merge**
- Lleva los cambios a la rama principal
- Pueden surgir conflictos (resolverlos con diff)
- Todas las ramas deberían eventualmente integrarse a la principal o ser descartadas

Línea Base

La línea base es un conjunto de ítems de configuración que han sido construidos y revisados **formalmente**.

Entonces una línea base es una **configuración** que ha sido revisada formalmente y sobre la que se ha llegado un acuerdo

Este conjunto de ítems debe tener una **referencia única**, y esto se logra por medio de **tags**

Básicamente la línea base es como una **foto o screenshot** de los ítems de configuración en un momento dado, y su principal función es que es un **punto de referencia** para:

- **Rollback**
- Futuros desarrollos

- Auditorías al código
- Lanzamiento o release

Si a vos se te ocurre cambiar una línea base, existe un protocolo formal de control de cambios. Dirigido por el **comité de control de cambios**,

Este comité de cambios se encarga de gestionar las **solicitudes de cambio**, y en caso de aceptarse, se acuerdan los nuevos parámetros.

Existen 2 tipos de linea base:

- 1. 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.

- 2. De especificación o documentación:** Son las primeras líneas base, dado que no cuentan con código. Podría contener el documento de especificación de requerimiento.

Se asocian a las fases de requerimientos, diseño, implementación, pruebas, etc

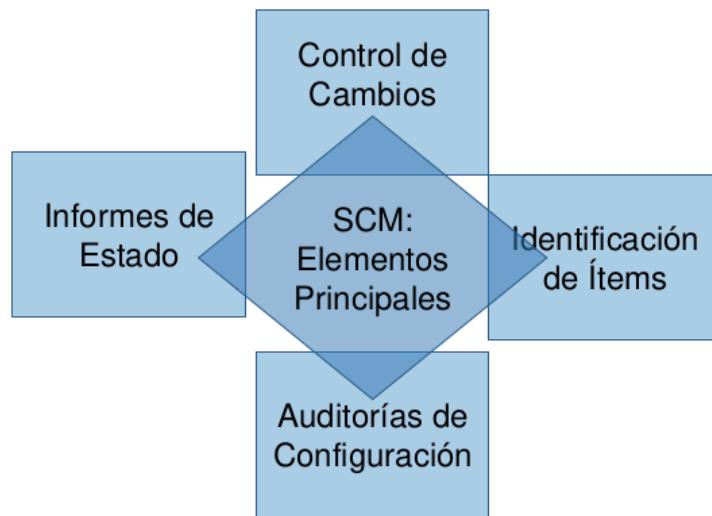


Configuración de software:

Es un conjunto de ítems de configuración con su correspondiente versión en un momento determinado

Es como si fuera una foto de los ítems de configuración en un instante determinado de tiempo

Actividades fundamentales / elementos de SCM:



Identificación de ítems de configuración:

Esto implica una identificación única para cada ítem. Esto implica

- Definir políticas, reglas de nombrado y versionados
- Definir la estructura del repositorio
- Definir la ubicación de los ítems en la estructura

Los ítems de configuración se clasifican según su ciclo de vida

- **Items de producto:** Tienen el ciclo de vida más largo, y se mantienen mientras el producto exista.
- **Items de proyecto:**

- **Ítems de iteración:**



Control de cambios:

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.

Para esto se recurre al **comité de control de cambios**, como ya hemos visto antes. A continuación vamos a profundizar un poco más este proceso:

1. Se recibe una **propuesta de cambio** sobre una línea base determinada, no sobre un ítem en específico
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.

- Finalmente se **notifica** a los interesados los cambios realizados sobre la **línea base**.

El comité de control de cambios está formado por representantes de todas las áreas involucradas en el desarrollo:

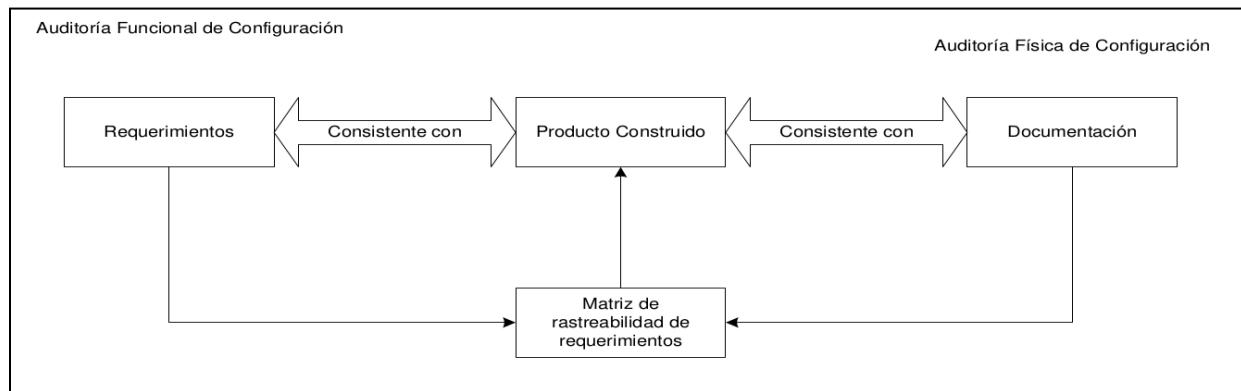
- Análisis, Diseño
- Implementación
- Testing
- Otros interesados

Auditorías de Configuración de Software:

Las auditorías son **evaluaciones independientes** (Esto significa que es realizada por un grupo de personas ajenas al equipo de trabajo) de los productos o procesos de software para asegurar el cumplimiento con estándares, lineamientos, especificaciones y procedimientos, basada en un criterio objetivo.

Es muy importante notar que la auditoría se realiza en torno a una **línea base**

Son un **instrumento para el Aseguramiento de Calidad en el Software**.



Las auditorías se clasifican en:

- **Auditoría física de configuración (PCA):**
 - En la auditoría física se verifica la integridad del repositorio, que los ítems de configuración se encuentren ubicados allí en el lugar correcto

- **Verificación:** Asegura que un producto cumple con los objetivos preestablecidos, definidos en la documentación de líneas base (línea base). Todas las funciones son llevadas a cabo con éxito y los test cases tengan status "ok" o bien consten como "problemas reportados" en la nota de release.

- **Auditoría funcional de configuración (FCA):** Se valida que el producto construido es el correcto. Se contrastan estos ítems de configuración anteriormente identificado, contra los requerimientos para ver si estamos construyendo un producto correcto

Validación: Valida que el producto a construir sea consistente con los requerimientos identificados en la ERS. Se usa una matriz de rastreabilidad para encontrar donde se implementan cada requerimiento

Validación: El problema es resuelto de manera apropiada de manera que el usuario obtenga el producto correcto

¿Y qué es esto de la matriz de trazabilidad de requerimientos?

Bueno no te vas a acordar pero esta herramienta la viste en parte en ASI. Básicamente vos querés rastrear los requerimientos a lo largo de todo el ciclo de vida del software. Mira a continuación te muestro un posible ejemplo

ID Requisito	Descripción del requisito	Casos de uso	Casos de prueba	Ítems de configuración	Estado
REQ-001	El usuario debe poder iniciar sesión	CU-01	TC-001, TC-002	<code>login.js,</code> <code>authController.java,</code> <code>config.yaml</code>	Implementado
REQ-002	El sistema debe bloquear al usuario tras 3 intentos fallidos	CU-02	TC-003	<code>securityService.java,</code> <code>login.js</code>	En desarrollo

[Informes de estado:](#)

La principal razón para usar informes de estado es para aportar **visibilidad** acerca de la gestión de configuración, yo en base a esto puedo tomar decisiones

Ahora te muestro algunos de ejemplos de informes que podes generar:

- Uno de los informes básicos se lo conoce como **inventario** el cual lista todos los ítems de configuración con su estado que tiene el repositorio en un momento de tiempo
- Otro informe podría ser la cantidad de solicitudes de cambio que se atendieron en un periodo de tiempo
- Listar el contenido de una línea base

SCM en ambientes agiles:

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.

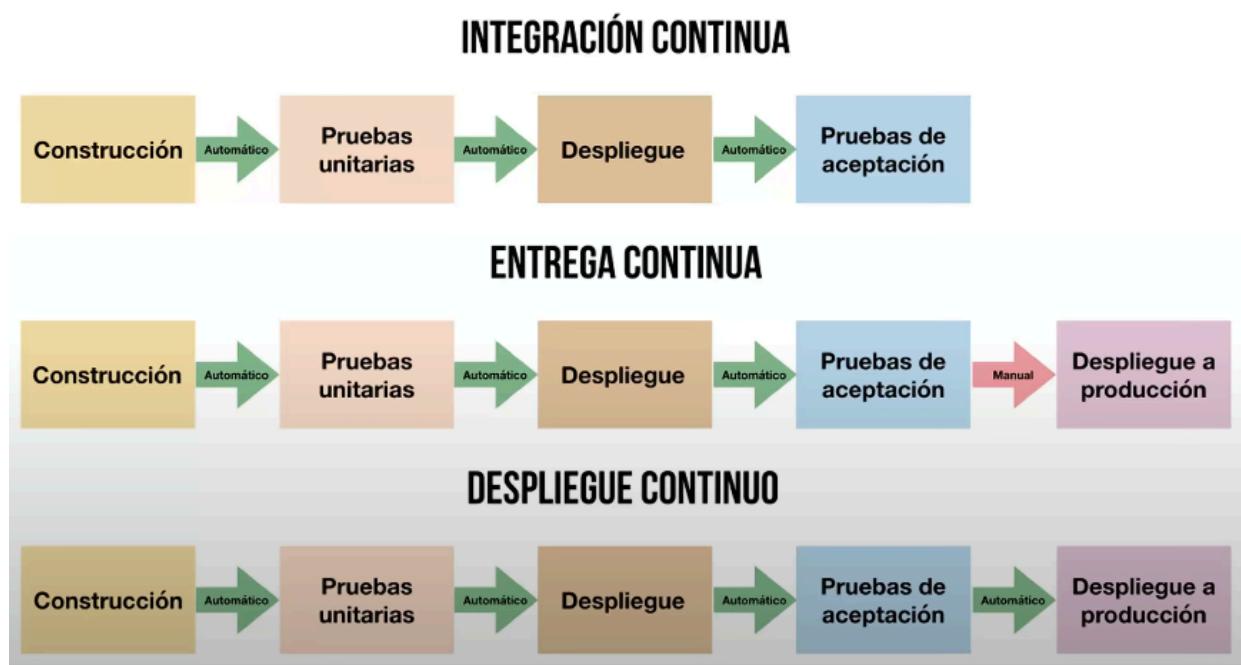
A continuaciones se destacan algunas **características** de SCM ágil:

- Los **equipos** ágiles son **autoorganizados**, por lo que las **Auditorías** de configuración **no** son una actividad propia de la gestión de configuración en los ambientes ágiles.
- **El comité de control de cambios desaparece**, debido a que ahora no tenemos una estructura rígida como lo establecen los procesos definidos, por lo que los equipos se autorregulan
- **Trazabilidad:**
 - **Tradicional (Documentación manual):** Se generan **informes formales** y rastreo detallado a través de herramientas especializadas.
 - **Ágil (Trazabilidad automática):** Se gestiona mediante **herramientas integradas** como Git, Jira, Azure DevOps, etc.

- Dada la naturaleza de los requerimientos ágiles, la **SCM responde a los cambios en lugar de evitarlos**.
- Esforzarse por ser transparente y *sin fricción*, automatizando tanto como sea posible
- Coordinación y automatización frecuente y rápida
- Eliminar el desperdicio → no agregar nada más que valor
- Documentación Lean y Trazabilidad
- Feedback continuo y visible sobre calidad, estabilidad e integridad

Prácticas continuas:

Como introducción a este tema vale la pena decir que las prácticas continuas vienen a ser como una **evolución** de SCM, que ya apuntan a una automatización completa



1. Integración continua:

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.

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.

2. Entrega continua:

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. Esto quiere decir que hay una serie de pruebas para saber si el código que vos subiste se integra correctamente al sistema

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. Es 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.

3. Despliegue continuo:

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

¿Y cuales son las estrategias de despliegue continuo que existen?

Blue - Green:

Es una técnica donde mantienes **dos entornos de producción idénticos**:

- Blue: la versión actual en producción.
- Green: la nueva versión que quieras desplegar.

Cuando estás listo para hacer el cambio, simplemente redireccionar el tráfico del entorno Blue al Green por medio de un **balanceador de cargas**

Canary Deployment:

En esta estrategia de despliegue la nueva versión del **producto se va liberando de manera gradual** a un **subconjunto de usuarios**, es decir que es como un **lanzamiento por etapas**

El funcionamiento sería algo como esto:

1. La nueva versión (v2) es desplegada a solo el **5%** de los usuarios.
2. Observamos cómo se comporta: errores, rendimiento, feedback.
3. Si va bien, incrementar el porcentaje: $25\% \rightarrow 50\% \rightarrow 100\%$.
4. Si hay problemas, puedes **detener o revertir** el despliegue fácilmente.

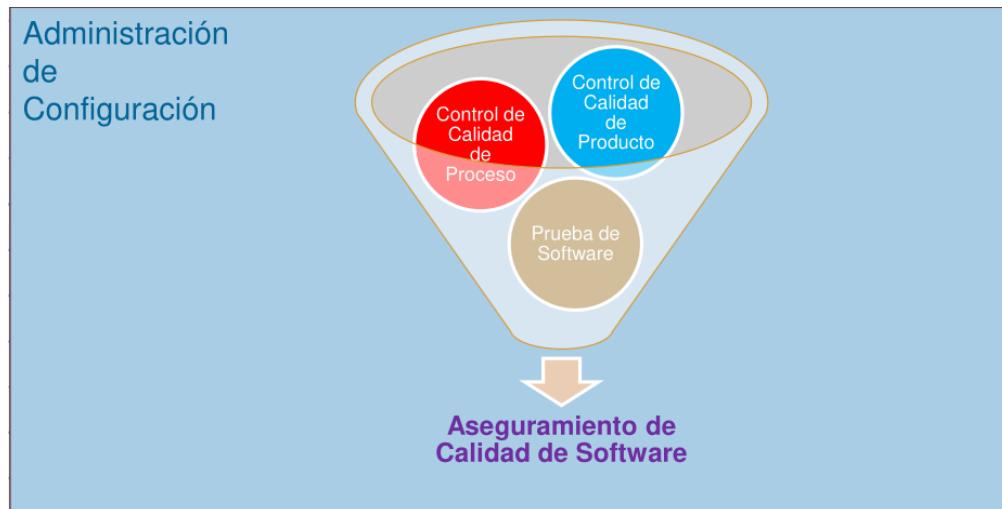
La ventaja de usar canary es que te permite ir obteniendo feedback de manera gradual, te permite monitorear cómo va funcionando la aplicación y si hay algún problema se realiza un **rollback**.

Unidad 4: Aseguramiento de calidad de proceso y de producto (PPQA)

Testing / Prueba de Software:

Testing en ambientes tradicionales:

Contexto:



El testing de software, es una de las tareas a realizar en el ámbito del **aseguramiento de calidad de software**, en conjunto con el control de calidad del proceso y del producto. La calidad del software se obtiene y se logra a lo largo de todo el desarrollo de este (detectar errores en etapas tempranas es siempre menos costoso, que

detectarlos después), por lo que una buena administración de configuración (SCM) es el puntapié inicial para permitir la realización de tareas de aseguramiento de calidad.

Concepto:

Proceso mediante el cual se somete a un **software** o un **componente** a condiciones específicas con el fin de determinar y demostrar si el mismo es válido o no en función de los requerimientos especificados.

La **visión** más apropiada del testing es:

- Proceso **destructivo** de tratar de encontrar los defectos (cuya presencia se asume) **en el código**
- Se debe ir con una **actitud negativa** para demostrar que algo es incorrecto
- **Testing exitoso** → Es el que **encuentra defectos**
- El **costo** del Testing está **entre un 30% y un 50%** del valor del producto.
- El testing **NO asegura** que se tenga un **producto** de calidad (ni la agrega), **ni que el proceso por el que se desarrolló sea de calidad.**

¿Y cuánto testing es suficiente?

El **testing exhaustivo** es imposible por la cantidad de tiempo que requiere. El momento en que se deja de hacer testing depende del nivel de riesgo o costo asociado al proyecto. Los riesgos permiten definir prioridades de qué se debe testear primero y con qué esfuerzo.

- **El Criterio de Aceptación** es lo que comúnmente se usa para resolver el problema de determinar cuándo una determinada fase de testing ha sido completada.

Estos criterios de aceptacion pueden ser definidos en términos de:

- Costos
- % de test corridos sin fallas
- No hay defectos de una determinada severidad en el software

¿Y cómo se relaciona el testing con el ciclo de vida?

Acá siguiendo los lineamientos de los principios del testing, lo ideal es realizarlo lo más temprano posible para poder abaratar costos y recursos.

En realidad no hace falta tener código para poder empezar a realizar testing. Desde que yo ya tengo una historia de usuario o los requerimientos definidos yo ya puedo empezar a probarlos

Nota: Esto es importante, fijate que según los principios el testing se **debería** realizar lo más temprano posible sin embargo hay modelos como en el de cascada que lo implementan en fases tardías y esto evidentemente trae una serie de consecuencias negativas

Algunos mitos sobre el testing:

- El testing es una etapa que comienza al terminar de codificar
- El testing es probar que el software funciona
- Testing = Calidad de producto
- Testing = Calidad de proceso
- El tester es el enemigo del programador

Aclaración:

- EL testing no es calidad de producto ni de proceso, solo me garantiza que el producto sea confiable
- Una de las razones por las cuales el testing no garantiza calidad es porque vos podés realizar un testing super bueno, con 0 errores y el producto puede seguir teniendo fallas en la calidad y esto se debe a que la misma es **subjetivo**, y depende de varios factores como la experiencia de usuario

Principios del Testing:

Principios	Explicacion
El testing muestra la presencia de defectos	El propósito del testing es encontrar defectos , no probar que el software está libre de ellos.
El testing exhaustivo es imposible	Es imposible probar todas las combinaciones posibles de entradas, condiciones y resultados en un sistema.
Testing temprano	El testing debe comenzar lo antes posible en el ciclo de desarrollo (por ejemplo, en la fase de requisitos). Detectar defectos temprano reduce costos y evita problemas mayores en etapas posteriores.
Principio de pareto	Los defectos tienden a concentrarse en un número reducido de módulos o áreas del sistema (Regla 80/20).
Paradoja del pesticida	Si se usan siempre las mismas pruebas, con el tiempo dejarán de encontrar nuevos defectos.
El testing es dependiente del contexto	Las técnicas y enfoques de testing dependen del tipo de sistema (por ejemplo, un sistema bancario requiere pruebas más rigurosas que una aplicación de entretenimiento).
Falacia de la ausencia de errores	Incluso si no se encuentran errores, el software puede no satisfacer las necesidades del cliente.
Un programador / unidad de programación debería evitar probar su propio código	Los programadores tienen un sesgo personal hacia su propio trabajo, lo que puede hacer que pasen por alto errores.

Conceptos a tener en cuenta:

Errores y defectos:

La diferencia es el **momento** en el cual se detectan y luego se solucionan:

- **Error:** Este se descubre en la misma etapa en la que estoy trabajando
- **Defecto:** Es un error pero que no lo pudimos descubrir en su etapa, por ende se trasladó a etapas posteriores.

El defecto es lo que nosotros vamos a buscar en el testing

Ahora bien, los defectos se pueden clasificar según:

- **Severidad:** Tiene que ver con la gravedad del defecto que yo encontré
 - Bloqueante → El defecto no permite que el sistema se pueda ejecutar
 - Crítico
 - Mayor
 - Menor
 - Cosmético → Este tiene que ver con temas como el formato de fechas, formato de números, distribución de componentes en la GUI, etc
- **Prioridad:** Esto tiene que ver con la urgencia que nosotros tenemos a la hora de encontrar el defecto
 - Urgencia
 - Alta
 - Media
 - Bajo

Ahora bien, esto no es tan intuitivo pero no es que mientras más grave es más urgencia tiene. Podríamos tener un defecto cosmético (error de ortografía) pero esto para una empresa podría ser muy urgente solucionarlo ya que no quieren quedar en su página web como si fueran unos analfabetos

Casos de prueba

Un caso de prueba es un conjunto de **condiciones o variables** que me permiten determinar si el software o una parte de él está funcionando correctamente o no

Un caso de prueba consta de 3 partes:

- **Objetivo:** La característica del sistema a comprobar
- **Datos de entrada y de ambiente:** Datos a introducir al sistema que se encuentra en condiciones preestablecidas
- **Comportamiento esperado:** La salida o la acción esperada en el sistema de acuerdo a los requerimientos del mismo

A continuación un **ejemplo** de caso de prueba:

ID	Prioridad (Alta, Media, Baja)	Nombre del caso de prueba	Precondiciones	Pasos	Resultado esperado
1	Media	Taxis libres filtrados por barrio	<ul style="list-style-type: none"> - El usuario "Carlos" se encuentra registrado y logueado con perfil de administrador - El software debe estar configurado ubicación con radio en Córdoba - Los barrios de Córdoba donde se prestan servicios de taxi deben estar cargados (Nueva Córdoba, El Cerro, Alta Córdoba, etc) - Hay taxis conectados al sistema de ubicación, con estado libre, con ubicación en Alta Córdoba y patente AF925ED 	<ol style="list-style-type: none"> 1. El administrador selecciona la opción "ver mapa de taxis" 2. El administrador selecciona el barrio de "Alta Córdoba" 3. El administrador selecciona filtro de estado "libre" 	EL sistema muestra un mapa de la ciudad de Córdoba, realizando un zoom en el barrio de Alta Córdoba visualizando el taxi en estado libre con la patente de AF925ED

¿Y qué inconveniente se nos presenta?

Yo no puedo identificar o definir casos de prueba de manera infinita, o bien de manera exploratoria o según lo que a mí se me vaya ocurriendo.

Entonces surge la necesidad de algún tipo de mecanismo que me permita definir la menor cantidad de casos de prueba pero que me permita cubrir el testing de una forma holística

Testing exhaustivo → Inviable, requiere demasiado esfuerzo

Entonces nos vamos a valer de la siguiente afirmación: **Los bugs se esconden en las esquinas y se congregan en los límites**

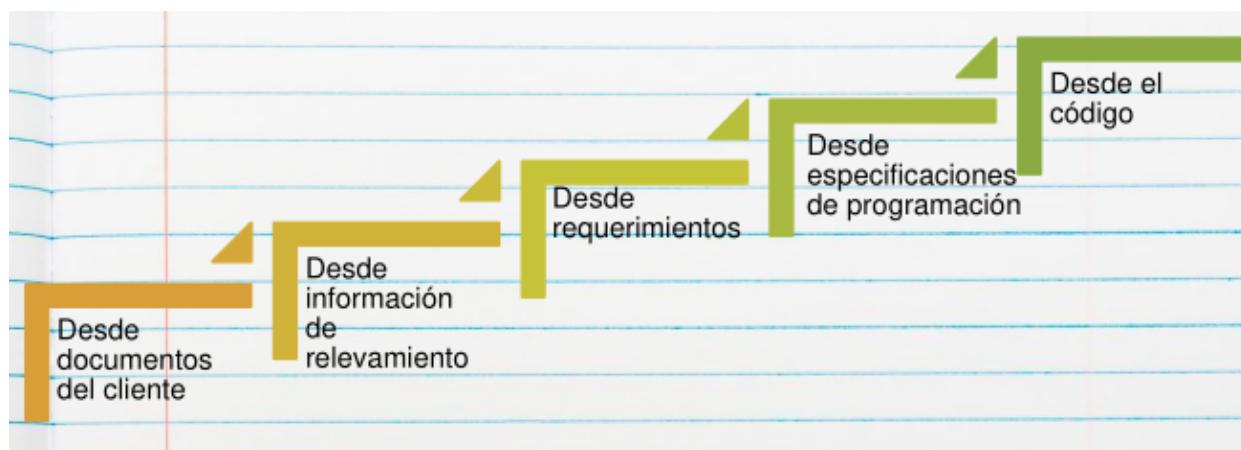
Por ejemplo: Supongamos que vamos a realizar una compra y el valor mínimo es \$0 y con valor maximo de \$100 000. Es pertinente aprobar la compra con valores **esquina**, como:

- \$100 000
- \$ 0
- \$ 1
- \$100 001
- \$100 000,32
- No tiene sentido por ejemplo probar \$43 534

Nota: Esto se vé reflejado en las clases de equivalencias:

Duración	9	Número mayor o igual a 10 y menor o igual a 25 con fracción de 0,5 o 0	10	Número mayor o igual a 10 y menor o igual a 25 con fracción distinta de 0,5 o 0
			11	Número entero menor a 10 o mayor a 25
			12	Otro valor
			13	No ingresa valor

¿Y cómo se deriva un caso de prueba?



Es importante entender que cuanto más documentado está nuestro software, es mucho más fácil derivar un caso de prueba

Fijate que si vos tenes un proyecto con puro código y casi nada escrito en la ERS, sería mucho más complicado derivar casos de prueba

Ciclo de Prueba:

Es la ejecución de un conjunto de casos de prueba en una versión determinada del producto. Generalmente se tienen 2 ciclos, y el primero es conocido como ciclo 0. El ciclo 0 siempre es **manual**, es donde se configura todo y a partir del ciclo 1 ya se pueden **automatizar** las pruebas.

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 regresión de dice, *volve a revisar todo antes de realizar un nuevo ciclo de test ya que al solucionar un error, es probable que hayan aparecido 1 o dos más*

Niveles de prueba:

Básicamente los niveles de prueba son los siguientes:

1. Testing unitario:

- Se prueba cada componente tras su realización / construcción
 - De forma **individual**
 - De forma **independiente**

- Se produce con acceso al código bajo pruebas y con el apoyo del entorno de desarrollo, tales como un framework de pruebas unitaria o herramientas de depuración
- Los errores se suelen reparar tan pronto como se encuentran, **sin constancia oficial** de los incidentes

```
// Función a probar
function sumar(a, b) {
  return a + b;
}

// Test unitario
test('Suma 2 + 3 debe ser 5', () => {
  expect(sumar(2, 3)).toBe(5);
});
```

Nota: Esta notación de *test*, *expect*, *toBe* son parte del framework Jest justamente diseñado para realizar pruebas unitarias

2. Testing de integración:

- Test orientado a verificar que las partes de un sistema que funcionan bien **aisladamente, también lo hagan en conjunto**
- Cualquier estrategia de prueba de versión o de integración **debe ser incremental** para lo que existen 2 esquemas principales:
 - Integración de arriba hacia abajo (top - down): Se comienza probando los componentes de alto nivel (como la interfaz de usuario o módulos principales) y se avanza hacia los componentes de bajo nivel (submódulos o dependencias).
 - Integración de abajo hacia arriba (bottom - up): Se inicia con los componentes de bajo nivel (módulos base, librerías, servicios) y se progresiona hacia los de alto nivel.

- Lo ideal es una combinación de ambos esquemas
- Tener en cuenta que los modelos críticos deben ser probados lo más tempranamente posible
- Los **puntos clave** del test de integración son simples:
 - Conectar de a poco las partes más complejas
 - Minimizar la necesidad de utilizar programas auxiliares

3. Testing de sistema:

- Es la prueba realizada cuando una aplicación está funcionando como un todo (Prueba de la construcción final)
 - Realizada sobre un incremento (Si utilizamos scrum)
 - Realizada sobre un producto (Si utilizamos un modelo de cascada)
- Trata de determinar si el sistema en su **globalidad** opera satisfactoriamente (recuperación de fallas, seguridad y protección, stress, performance, etc)
- Se suelen utilizar **casos de prueba**
- El entorno de prueba debe asimilarse al entorno real en donde se debería ejecutar el sistema
- Deben investigar tantos requerimientos funcionales y no funcionales del sistema

4. Testing de aceptación:

- Es la prueba realizada por el usuario para determinar si la aplicación se ajusta a sus necesidades
- La meta en las pruebas de aceptación es el de establecer confianza en el sistema, las partes del sistema o las características específicas y no funcionales del sistema

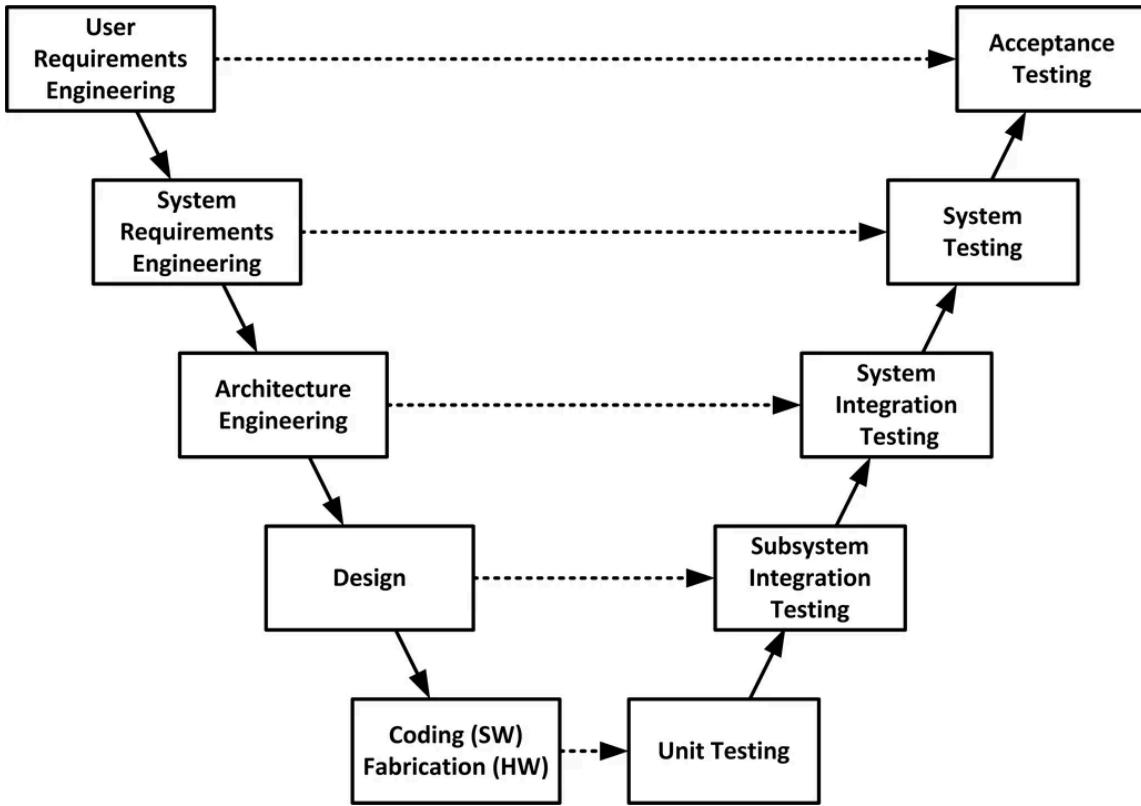
- Encontrar defectos **no es el foco principal** en las pruebas de aceptación
- Aquí se encuentran **dos tipos de pruebas:**
 - **Alfa:** Las realizadas por el usuario en el ambiente de testing.
 - *Una empresa desarrolla un ERP y antes de entregarlo, invita a un grupo reducido de usuarios clave a probarlo en el entorno de pruebas interno. Allí validan procesos y reportan inconvenientes mientras los desarrolladores observan.*
 - **Beta:** Las realizadas por el usuario en un ambiente real de uso
 - *Una app móvil lanza una versión Beta pública para que un grupo de usuarios la usen en sus teléfonos, reporten errores y den opiniones antes del lanzamiento oficial.*

Modelo en V:

El modelo en V representa una **variante** del modelo en cascada, en donde primero hay un **enfoque descendente** que va desde un **nivel de granularidad alto** (**Requerimientos del usuario**) hasta lo más fino que hay que sería el código.

En este enfoque descendente se establece un proceso de **verificación** que significa si estamos construyendo el producto de manera correcta.

En el enfoque ascendente se establece un mapeo de cada actividad del proceso con un nivel de testing y se establece un **proceso de validación** que significa que si el producto que estamos construyendo es el correcto.



Ambientes del testing:

Los ambientes son básicamente los lugares en donde se trabaja en la **construcción de software**:

- **Desarrollo:**

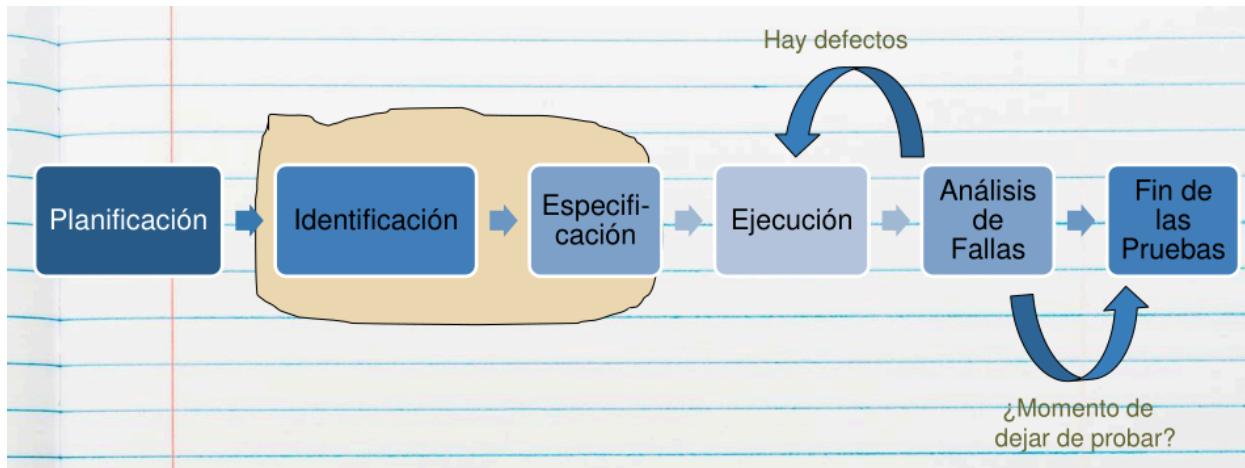
- Implica hardware y software necesarios (librerías, extensiones, IDEs, compiladores, etc.) para poder desarrollar y desplegar el producto y utilizarlo.
- En este ambiente se realizan las pruebas unitarias

- **Pruebas:**

- Es el ambiente que utilizan los **testers** para llevar a cabo las pruebas, y los desarrolladores no deben tener acceso.
 - En este ambiente se realizan **las pruebas de integración**.
- **Preproducción:**
- Debería tener las **mismas características** que el ambiente productivo para poder comprobar que el producto funcionará una vez desplegado en producción de manera correcta.
 - El **problema** de este entorno es que muchas veces **resulta difícil** (o imposible en algunos casos), **debido a que es muy costoso replicar principalmente las configuraciones de hardware**.
 - Simular la concurrencia puede representar un problema a menudo
 - Producción puede tener configuraciones únicas que no siempre se trasladan a Pre-Producción por restricciones de seguridad o arquitectura.
 - En este se realizan las pruebas del sistema. Normalmente acá se realizan las **pruebas de sistema y de aceptación**
- **Producción:** Este entorno, **es la configuración de software y hardware que tienen los usuarios finales del software**, y que utilizan para el desempeño de sus actividades laborales. Obviamente en este entorno no se realizan pruebas (probar en este ambiente tiene grandes consecuencias), ya que en teoría la versión del producto **cumple con los criterios del Definition of Done**.

Proceso de pruebas:

Fijate que al hablar de proceso de pruebas, nos situamos en el contexto de un **proceso definido**



1. Planificación:

Acá básicamente se realiza lo que es el **plan de pruebas**, el cuál contiene:

- Riesgos y Objetivos del testing
- Estrategia de testing
- Recursos
- Criterio de aceptación

2. Diseño (Identificación y Especificación):

- Se identifican los datos necesarios, para así poder **diseñar y priorizar los casos de pruebas** que se van a implementar
- Se **diseña el entorno de pruebas**
- **Evaluar la testeabilidad** de los requerimientos y el sistema
- Se analiza **si se utilizará regresión** o no

3. Ejecución

- Se empiezan a ejecutar los **casos de prueba**
- Creación de **conjuntos de pruebas** de los casos de prueba para la ejecución de la prueba eficientemente (**Ciclos de prueba**)
- **Automatizar** lo que sea necesario

- Registrar el **resultado** de la ejecución de pruebas y registrar la identidad y las versiones del software en las herramientas de pruebas
- Comparar los resultados **reales** con los resultados **esperados**

4. Evaluación y Reporte

- Se identifican y corigen los defectos encontrados hasta que se cierren todos los casos de prueba.
- Para realizar este análisis se recurre a **criterios de aceptación**
- Se confecciona un **informe de reportes**

Testing de caja negra:

En esta estrategia no se dispone de la **estructura interna** de la implementación, sino que se analizan las funcionalidades como una caja negra, en términos de entradas y salidas de esa “caja”.

El proceso consiste en ingresar determinados datos a esa funcionalidad vista como caja negra, y luego comparar los resultados obtenidos con los resultados esperados.

Este método se tiene 2 técnicas:

- **Métodos basados en especificaciones:** Son aquellos que se ejecutan utilizando la documentación de especificaciones realizadas del producto.
 - **Partición de equivalencias:** Se analiza básicamente cuales son las distintas **condiciones externas** asociadas a una funcionalidad del software. Estas condiciones externas vienen a ser como las entradas y salidas de esta funcionalidad.

Lo que se hace en este método es identificar las **clases de equivalencias** (tanto de entrada como de salida) que me definen subconjuntos de datos que pueden ser **válidos o nulos**

EL último paso del método sería **identificar los casos de prueba**

- **Análisis de los valores límites:** Esto es una herramienta que se utiliza en la partición de equivalencias. Utilizar los límites o valores de borde de las clases de equivalencia para la definición de los casos de prueba.
- **Métodos basados en experiencia:** La experiencia y los conocimientos del tester son fundamentales para determinar las entradas del sistema y analizar los resultados
 - **Adivinanza de defectos:** Enfoque basado en la intuición y experiencia para identificar pruebas que probablemente expongan defectos del software, elaborando una lista de defectos posibles o situaciones propensas a error y realizando pruebas a partir de esa lista.
 - *Por ejemplo en los formularios es común que en los mails falte la validación del arroba así que voy a empezar por ahí*
 - **Testing exploratorio:** El tester mientras va probando el software, va aprendiendo a manejar el sistema y junto con su experiencia y creatividad, genera nuevas pruebas a ejecutar.

Testing de caja blanca:

El **método de la caja blanca** es un enfoque de pruebas de software en el que el tester tiene acceso al **código fuente, pseudocódigo o diagramas de flujo** del sistema y diseña casos de prueba basados en su estructura interna. Este método se enfoca en verificar la lógica, los flujos de control y los datos internos del programa para garantizar que todo funcione como se espera

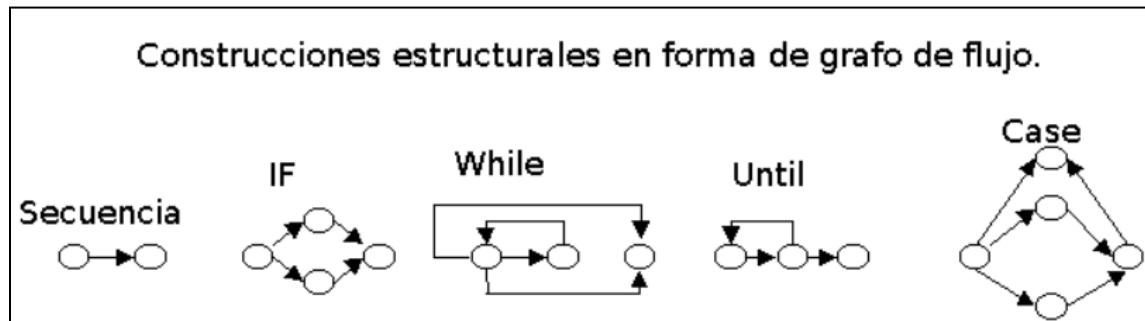
Cobertura de los caminos básicos:

- **Objetivo** → Encontrar todos los caminos independientes de una funcionalidad que deben recorrer (testear) al menos una vez, para probar cada uno de ellos con casos de prueba.

- Permite obtener una métrica denominada **complejidad ciclomática (M)**, que representa la cantidad de caminos independientes que posee una funcionalidad, y nos da un **límite inferior** para el número de casos de prueba que hay que construir, para ejecutar todas las instrucciones de la funcionalidad al menos una vez.
- Mientras **más alto** es el valor de la complejidad ciclomática, **más riesgo** representa para el software, con lo cual es **menos estable**

¿Y cómo sería el procedimiento?

1. Representar la funcionalidad por medio de un **grafo o diagrama de flujo** (se excluyen algoritmos que implementen métodos recursivos)



2. Se calcula la **complejidad ciclomática (M)**, para lo cual tenemos 2 formas:

a. $M = E - N + 2^*P$

- M = Complejidad ciclomática
- E = Número de aristas del grafo
- N = Número de nodos del grafo
- P = Número de componentes conexos o nodos de salida

- b. También se puede obtener la complejidad ciclomática como el **número de regiones cerradas + 1**

Caja Blanca

Cobertura de enunciados o caminos básicos

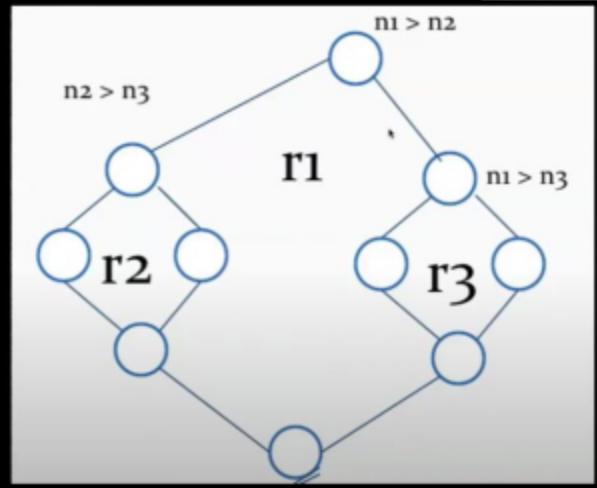
$$M = E - N + 2 \cdot P$$

$$M = 12 - 10 + 2 \cdot 1$$

$$M = 4$$

$$M = \text{Número de regiones} + 1$$

$$M = 3 + 1$$



3. Se define el **conjunto mínimo de caminos independientes** (asignando valores que provocan ir tomando cada uno de los caminos de la funcionalidad). **No son casos de prueba**. Siguiendo el ejemplo:

TC 1	TC 2	TC 3	TC 4
N1 = 8	N1 = 8	N1 = 4	N1 = 4
N2 = 4	N2 = 4	N2 = 8	N2 = 8
N3 = 4	N3 = 8	N3 = 4	N3 = 8

- Cada **columna** es un **Test Case** distinto
- Cada test case tiene un conjunto de valores que se la activa, por ejemplo en el TC 1, los valores que la activan son (8,4,4)

Cobertura de sentencias:

- **Sentencia:** Cualquier instrucción que involucra acciones como asignación de variables, operaciones numéricas, invocación de métodos, etc
- **Objetivo:** Buscar la cantidad **mínima** de casos de pruebas que me permitan ejecutar cada **línea de código** al menos una vez

- Ejemplo:

```
if x > 0:  
    print("Positivo")  
print("Fin")
```

- Para $x = 1$ y $x = -1$, se logra cobertura de sentencias.

Cobertura de decisión:

- Garantiza que se evalúe cada rama de las estructuras de control (como if, while, for) al menos una vez como verdadera y falsa.
- **Límite:** No evalúa condiciones internas dentro de las decisiones compuestas
- Ejemplo:

```
if x > 0 and y < 5:  
    print("Caso 1")  
else:  
    print("Caso 2")
```

- Caso 1: $x = 1$, $y = 3$ (verdadero).
- Caso 2: $x = -1$, $y = 6$ (falso).

Cobertura de condición:

- Asegura que cada **condición individual** dentro de una decisión compuesta se evalúe como verdadera y falsa.
- Ejemplo:

```
if x > 0 and y < 5:  
    print("Cumple")
```

- $x > 0$ como verdadero y falso.
- $y < 5$ como verdadero y falso.

Nota: Cabe destacar que existe la **cobertura de decisión / condición** en la cual evidentemente se cubren ambos aspectos

Tipos de pruebas:

Smoke test:

Es una **primera corrida de los tests** de sistema que provee cierto aseguramiento de que el software que está siendo probado no provoca una falla catastrófica.

Son pruebas muy **superficiales**

¿ Y qué es lo que se suele probar?

- ¿La aplicación arranca?
-
- ¿Se puede iniciar sesión?
-
- ¿Carga la pantalla principal?
-
- ¿Responde el backend?
-
- ¿Se conecta a la base de datos?
-
- ¿No hay errores fatales?

Testing funcional:

- Las pruebas se basan en funciones y características (descrita en los documentos o entendidas por los testers) y su interoperabilidad con sistemas específicos
- Basado en requerimientos

- Basado en los procesos de negocio
- Evidentemente derivan de requerimientos funcionales
- Por ejemplo: En una aplicación bancaria, las pruebas funcionales verificarán si los usuarios pueden realizar correctamente transacciones como transferencias de dinero, pagos de facturas, consultas de saldo, etc.

Testing no funcional:

- Es la prueba de **cómo** funciona el sistema
- No hay que olvidarlas, los requerimientos no funcionales son tan importantes como los no funcionales
- Las pruebas que podemos realizar son:
 - **Performance:** Se analiza el tiempo de respuesta y la concurrencia
 - **Carga:** Se mira el comportamiento de dispositivos de hardware (procesadores, memorias, etc)
 - **Stress:** Se fuerza al sistema a que falle, se lo somete a condiciones más allá de las normales
 - **Usabilidad:** Si es cómodo de utilizar para el usuario
 - **Portabilidad:** Se prueba en distintos entornos

TDD (Test Driven Development):

- Este es un tipo de **proceso** en el que nos enfocamos en construir primero la prueba unitaria cuando tengo los requerimientos y luego codear el componente.
- Básicamente se **escribe código** a partir de la ejecución de casos de prueba

- El ciclo de TDD sigue un proceso conocido como **Red-Green-Refactor**, que implica tres pasos fundamentales:
 1. **Rojo:** Escribe un caso de prueba basado en una funcionalidad que quieras implementar. Al principio, este test fallará porque aún no has escrito el código para esa funcionalidad.
 2. **Verde:** Escribe el código mínimo necesario para hacer que la prueba pase (que el test sea exitoso).
 3. **Refactorizar:** Mejoras el código para hacerlo más limpio y eficiente, sin que las pruebas dejen de pasar.

Testing en ambientes ágiles:

Nota: Todos los conceptos que vayamos a ver a continuación provienen del libro de Agile Testing escrito por Lisa Crispin y Janet Gregory

Introducción:

Con respecto a las principales diferencias entre testing en ambientes tradicionales vs agiles, nos encontramos con:

- El ciclo de vida
- **Quien** es la persona que realiza testing
- En ambientes ágiles se promueve mucho lo que es la **automatización** del testing, esto trae muchos beneficios como reducción de la latencia del feedback, capacidad de solucionar y detectar defectos antes, etc.

Valores del manifiesto del testing ágil:

1. Testing a lo largo del proceso y no al final

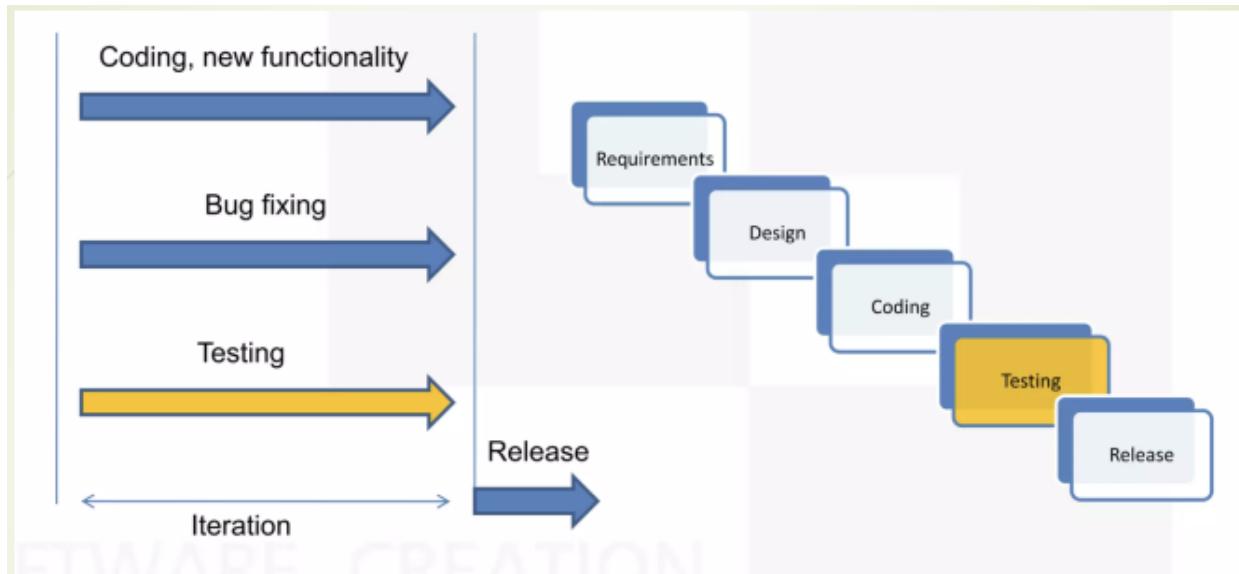
2. Prevenir bugs en lugar de buscar bugs
3. Construir el mejor sistema en lugar de romperlo
4. Todo el equipo es responsable de la calidad, en lugar de solo el tester
 - **Covaro:** *Esto tiene directa relación con scrum en donde se entendía como desarrollador a todos los que de alguna forma le aportan valor al producto*
5. Entendimiento del testing en lugar de chequear la funcionalidad

Los principios del manifiesto del testing ágil:

1. Testing se mueve hacia adelante en el proyecto
2. Testing no es una fase
3. Todos hacen testing
4. Reducir la latencia del feedback
5. Las pruebas representan expectativas
6. Mantener el código limpio, corregir los defectos rápido
7. Reducir la sobrecarga de documentación de las pruebas
8. Las pruebas son parte del done
9. En lugar de probar al final se utiliza TDD

Ciclo de vida (Agil vs Tradicional)

Aca mandale fruta con diferencias, no me voy a explayar, esto es sencillo



Prácticas concretas:

1. Pruebas unitarias y de integración automatizadas: Concepto ya visto pero ahora es automatizado
2. Pruebas de regresión a nivel de sistema automatizadas:

El objetivo de las pruebas de regresión es verificar que los cambios implementados en el código en el código (nuevas funcionalidades, correcciones, refactorizaciones) no rompan el funcionamiento existente del sistema

Este concepto ya lo habíamos visto, pero una vez ejecutado un ciclo de prueba de conviene hacer pruebas de regresión para ver si no salieron nuevos errores a flote

3. Pruebas explotadoras:

Son pruebas no planificadas y realizadas manualmente por testers con el objetivo de "explorar" el sistema, buscando errores inesperados o fallos que no se detectan con pruebas automatizadas o tradicionales.

Se basan en la **creatividad y juicio** del tester

4. TDD (Test Driven Development): Concepto ya visto

5. ATDD (Desarrollo conducido por pruebas de aceptación):

Similar a TDD, pero se enfoca en pruebas de **aceptación**, las cuales reflejan los requisitos del cliente o usuario final. Este enfoque es mucho más colaborativo que TDD porque no se cierra en un dev sino que involucra ya al cliente (O el PO que representa sus intereses)

Entonces partimos de una **user story** por ejemplo y luego de eso vamos a definir los **criterios de aceptación** utilizando lenguaje natural con el programa Gherkin siguiendo la estructura:

Es un formato narrativo que expresa el contexto, la acción y el resultado esperado de una funcionalidad:

Given (Dado) → Contexto inicial o precondiciones.

- *Dado que el usuario está logueado*
- *Dado que el carrito está vacío*

When (Cuándo) → Acción principal que dispara el comportamiento.

- *Cuando agrega un producto al carrito*

Then (Entonces) → Resultado esperado o verificación.

- *Entonces el carrito debe mostrar 1 producto*
- *Y el total debe ser 100*

La clave está cuando **se utilizan framework como por ejemplo cucumber** que básicamente transforman este tipo de especificaciones en lenguaje natural de Gherkins en test de código:

```
@Given("el usuario está logueado")  
public void elUsuarioEstaLogueado() {
```

```
    usuario = new Usuario("Juan", true);  
}
```

6. Control de versión de las pruebas con el código:

Guardar los scripts de pruebas (unitarias, integración, regresión, etc.) en el mismo sistema de control de versiones que el código (como Git).

Funciones de los roles en los equipos ágiles

El rol del **tester** desempeña las siguientes funciones:

- Preparación y mantenimiento de ambiente de pruebas
- Casos de pruebas con respecto a historia de usuarios
- Validación del *Done* de las historias
- Registra y notifica bugs
- Aceptación de la historia con el PO
- Métricas de calidad

Con respecto al **product owner**:

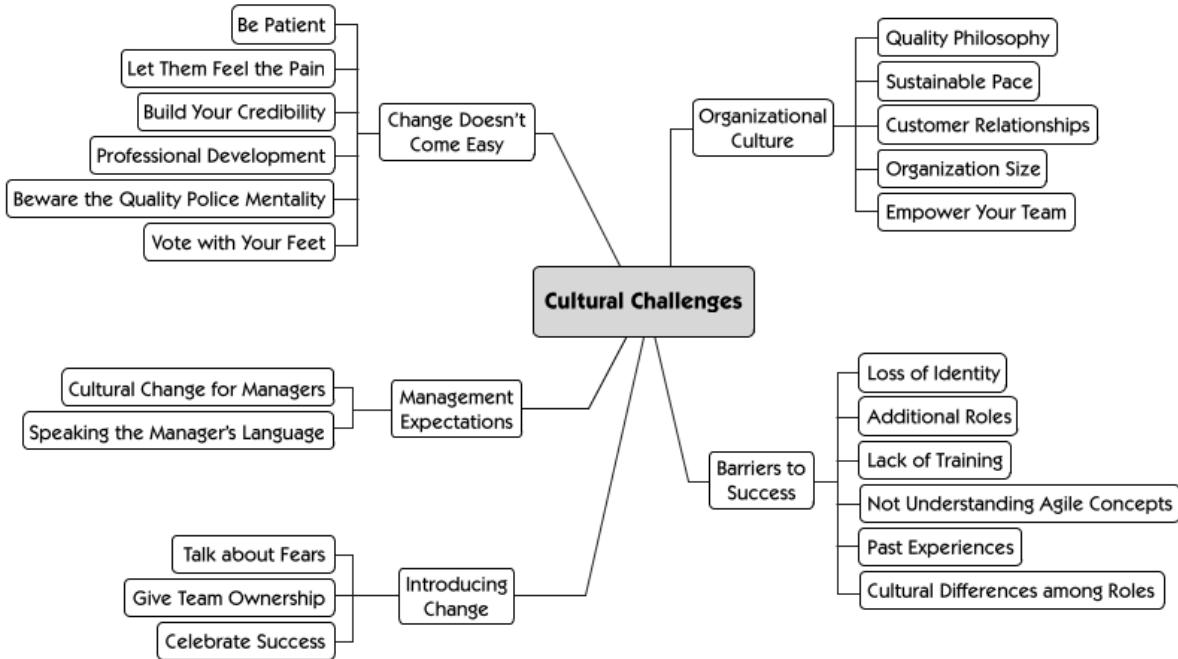
- Desarrollo de las historias de usuario con sus criterios de aceptación
- Mejor entendimiento del dominio del negocio
- Identificar dependencias

Con respecto a los **developers**:

- Definir estrategias de prueba
- Automatización de las ATDD
- Detección y notificación de bugs
- Feedback temprano de las historias
- Sugiere mejoras funcionales y de usabilidad

Cambios culturales:

Entonces implementar esto que venimos diciendo impone un **cambio cultural** a nivel de la organización, y es algo complejo de manejar.



Ejemplos:

Con respecto a los equipos:

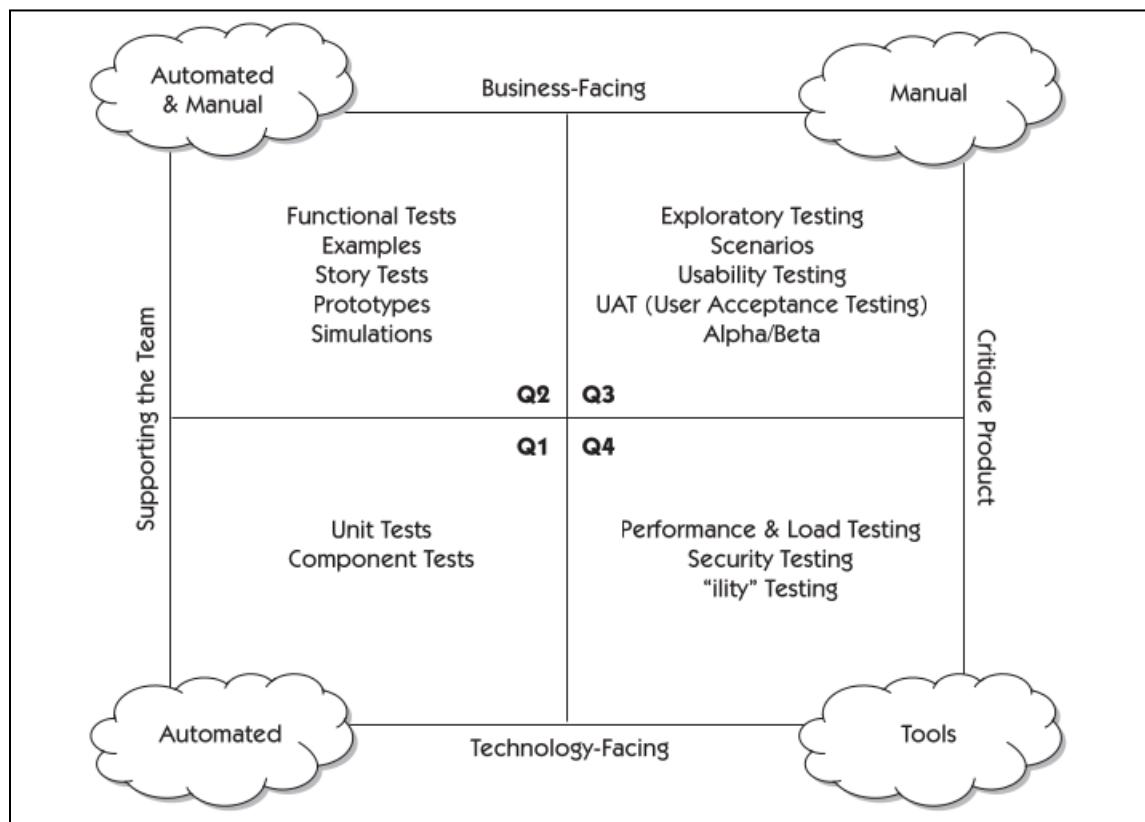
- En culturas tradicionales, los gerentes suelen ejercer un control estricto sobre el trabajo del equipo, mientras que el agilismo fomenta la **autonomía** y la **autogestión** de los equipos.
- Esto puede generar resistencia, especialmente en líderes acostumbrados a una estructura jerárquica rígida.

Con respecto a los roles:

- En el agilismo, los roles tradicionales de gerencia (como jefes de proyecto) pueden verse transformados o incluso eliminados, dando lugar a nuevos roles como **Scrum Masters** do **Product Owners**.
- Esto puede causar confusión o incluso resistencia en personas que perciben una pérdida de poder o estatus.

Cuadrantes del testing ágil:

Mira basicamente este gráfico te explica los **distintos tipos de testing** pero según el foco que vos le quieras hacer (dependiendo del cuadrante)



Cuadrante 1:

- Este cuadrante hace referencia principal al **Test Driven Development**
- Las pruebas unitarias me permiten testear funcionalidades muy pequeñas como objetos o métodos

- Las pruebas de componentes me permiten testear funcionalidades un poco más abarcativas como los son un conjunto de clases que pueden otorgar determinado **servicio**

Cuadrante 2:

- Estos test le siguen dando soporte al equipo pero a un nivel superior al del Q1. Esto quiere decir que son pruebas a un nivel **funcional** en el que se **testean distintas condiciones de satisfacción del negocio**
-

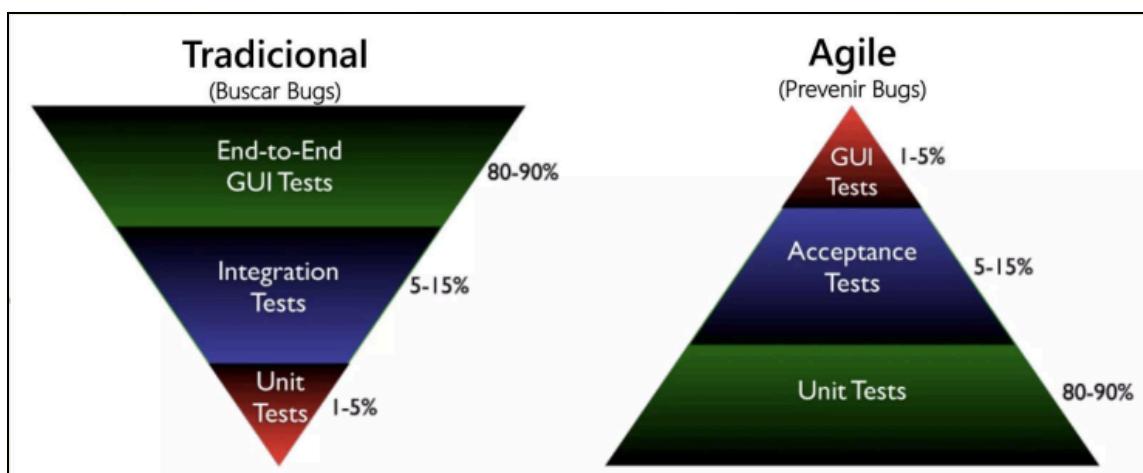
Cuadrante 3:

- En este cuadrante se agarran y clasifican las pruebas enfocadas al negocio para ver si cumplen o no con lo que quieren los usuarios o si va a hacerle frente a la competencia
- Cuando hacemos pruebas orientadas al negocio para criticar el producto, tratamos de emular la forma en que un usuario real usaría la aplicación. Estas son pruebas manuales que solo una persona puede realizar.
- La **prueba de usabilidad** es un ejemplo de un tipo de prueba que incluso tiene toda una ciencia propia. Se pueden reunir grupos focales (focus groups), observarlos mientras usan la aplicación y entrevistarlos para recopilar sus reacciones.
- **Pruebas alfa:** Son las primeras pruebas **realizadas por el equipo de desarrollo o un grupo selecto de usuarios dentro de la misma empresa**, generalmente antes de que el software sea lanzado al público.
- **Pruebas beta:** Son las pruebas realizadas por un grupo más grande y externo de usuarios (usuarios finales o clientes potenciales) fuera de la empresa de desarrollo, en un entorno real o cercano a producción.

Cuadrante 4:

- Security
- Availability
- Maintainability

Pirámide de testing (Tradicional vs Ágil):



Como **conclusión** podemos decir que:

- Tradicional: Caracterizado por testing end-to-end el cual es lento, costoso, y se realiza sobre una versión ya pulida del sistema
- Ágil: Más pruebas unitarias (80-90%), menos pruebas de GUI. Se previenen bugs desde el desarrollo. Esto va en línea con los principios y valores del manifiesto de testing ágil que dice que hagamos testing a lo largo del proceso y no solo al final

También se relaciona con las prácticas de CI/CD

PPQA:

Calidad:

Concepto:

Todos los **aspectos** y **características** de un producto o servicio que se relacionan con su habilidad de alcanzar las necesidades explícitas o implícitas

La calidad es **relativa** a las personas, a su edad, a las circunstancias de trabajo, el tiempo, etc

¿Y qué se entiende por calidad en el desarrollo de software?

- Las expectativas del cliente
- Las expectativas del usuario
- Las necesidades de la gerencia
- Las necesidades del equipo de desarrollo y mantenimiento
- Otros interesados

¿Qué problemas presentan los proyectos sin calidad?

- Atrasos en las entregas
- Costos excedidos
- Falta de cumplimiento en los compromisos
- No están claros los requerimientos
- El software no hace lo que tiene que hacer
- Trabajo fuera de hora
- Fenómeno del 90-90
- No se aplica bien el SCM
 - ¿Dónde está este componente?

Principios de calidad:

- La calidad no se **inyecta** ni se compra, debe ser **embebida**: No es que vos agarras un proyecto y le *metes* la calidad, sino que la misma debe ser concebida desde el momento 0, es decir desde los requerimientos

- Es justamente embebida porque pertenece a las actividades que se desarrollan en la disciplina de soporte de la ingeniería (Por lo tanto es una actividad transversal)
- Es un esfuerzo de todos
- Las personas son la clave para lograrlo → **Capacitación**
- Se necesita apoyo a nivel gerencial → **Pero se puede empezar por uno**
- Se debe liderar con el ejemplo
- No se puede controlar lo que no se mide, entonces debemos utilizar **métricas**
- **Simplicidad**, empezar por lo básico
- El aseguramiento de la calidad debe **planificarse**
- EL aumento de las pruebas no aumenta la calidad
- Debe ser **razonable** para mi negocio

Visiones de la calidad:

A la calidad se la puede analizar desde distintas perspectivas o visiones:

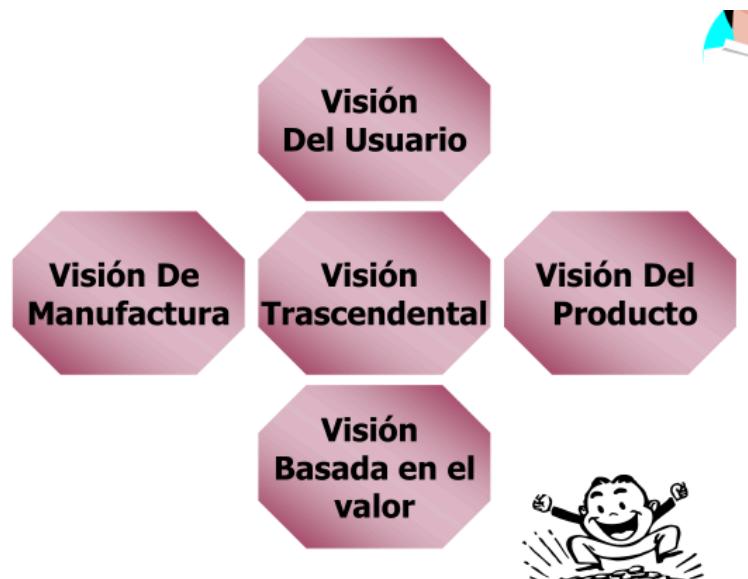
- **Visión del usuario:** Esto tiene que ver con las **expectativas** que tiene el usuario para con el producto, y si este las satisface.

Esta es quizás la más **complicada** de establecer, porque está en la cabeza de los usuarios, lo cual es una razón más por la que el principio de **comunicación constante** es crucial, para ir validando en todo momento si estamos en el camino correcto.

Esta visión se ve reflejada en el **agilismo** el cual supone que la **comunicación cara a cara** es fundamental para la transferencia de información.

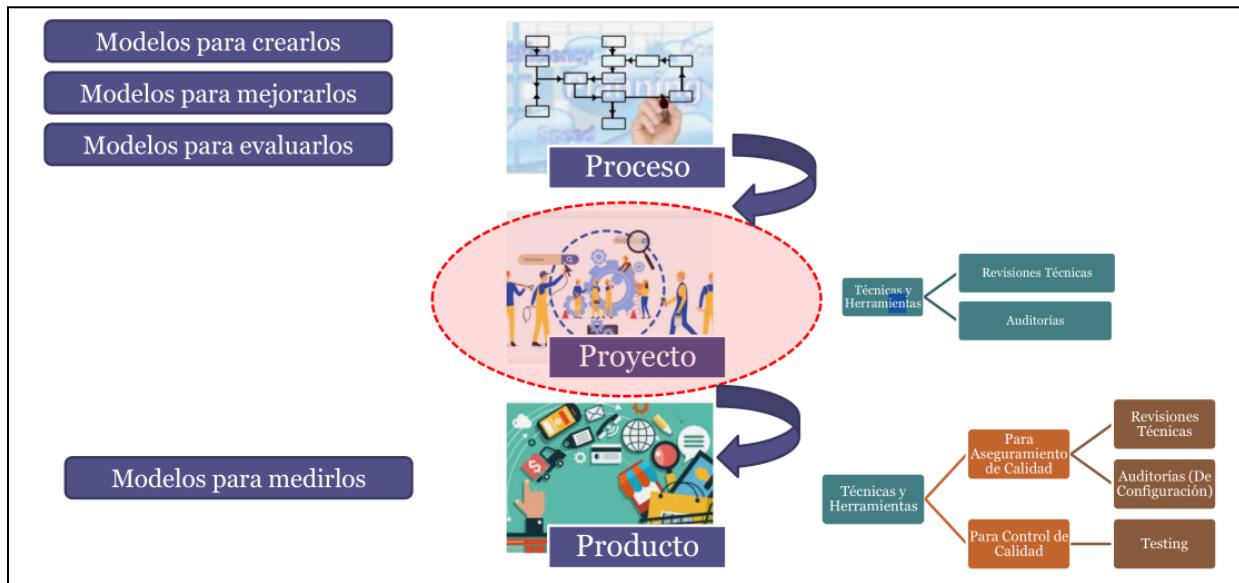
- **Visión del proceso (manufactura):** Si el proceso de desarrollo utilizado es el correcto para el producto que se desea desarrollar, es decir, el proceso aporta valor al producto y no produce desperdicios.
- **Visión del producto:** Esto se asocia al nivel de satisfacción de los requerimientos particulares de cada producto.
- **Visión del valor:** Encontrar un equilibrio en la relación costo-beneficio, para obtener siempre el mayor valor para el cliente posible y, obviamente generar ganancias con el desarrollo del software.
- **Visión trascendental:** Esta visión es un tanto utópica, ya que se asocia a objetivos difíciles de alcanzar.

Por ejemplo 0 defectos en el producto, pero son aquellos objetivos que motivan a seguir en busca de la mejora continua.



Calidad en el software:

En primer lugar vamos a explicar la siguiente imagen, que es bastante importante:



La gente necesita **definir un proceso** para llevar a cabo el software (a para realizar cualquier otra actividad). Este proceso es algo así como una guia que me permite ir **desde acá hasta allá**

Si quiero funcionar con **mejora continua** tengo modelos para mejorar esos procesos (Acá tenemos modelos como el IDEAL o el SPICE).

Nota:

Con respectos los modelos para mejorar procesos tenemos a Kanban por ejemplo que funciona bajo los lineamientos de Lean, mientras que el IDEAL o el SPICE funcionan bajo el marco de los procesos definidos

También tenemos **modelos de evaluación** que ven el grado de adherencia del proceso al modelo que se tomó de referencia.

- Por ejemplo, Un modelo **para evaluar, es la ISO 9001**, si llamo a una auditoría de ISO esa auditoría debería decirte todas las diferencias que se tienen en ese proceso definido respecto de lo que deberías tener.

Los procesos se instancian en los proyectos. Los procesos son sólo una indicación de cómo hacer las cosas, pero es el proyecto que al ejecutarse se insertan actividades para ir regulando si lo que estoy haciendo es lo que se había pensado.

A lo largo del proyecto y del producto que se va generando nosotros tenemos distintas **herramientas**:

Para el aseguramiento de calidad::

- Tenemos las **revisiones técnicas** que se hacen entre pares, no se somete a la persona a evaluación sino el producto. Se puede hacer sobre cualquier artefacto que queramos: requerimientos, etc.
- Luego tenemos las **auditorías** que son realizadas por personas externas (los empíricos están muy en contra porque creen que los equipos son capaces de reconocer y corregir por sí mismos, esto puede verse en la retrospectiva).

Es importante notar que las revisiones técnicas y las auditorías se pueden realizar tanto a nivel de producto como proyecto

- A nivel de producto puedo hacer una auditoría por ejemplo llamando a un experto en patrones de diseño de software y que me corrija
- A nivel de proyecto puedo realizar una auditoría para ver si mi proyecto está respetando el proceso que dijo que iba a utilizar
-

Para el control de calidad::

- **El testing**

Calidad en el producto:

Aclaracion 1:

Otro tema importante para hacer hincapié es que cuando hablamos de calidad en el producto, a diferencia de calidad en el proceso todavía en la industria no se ha diseñado una plantilla que vos puedas aplicar por así decir y que te sirva para todos los productos como lo tenemos con CMMI, SPICE

Estos modelos que vamos a ver son **teóricos** y queda en nuestra práctica definir los aspectos más importantes para hacer hincapié en la calidad

Por otro lado es sumamente complejo **entregar un certificado** que diga que tu producto tiene calidad porque la misma depende de la que usa el producto y cada producto tiene sus propias características

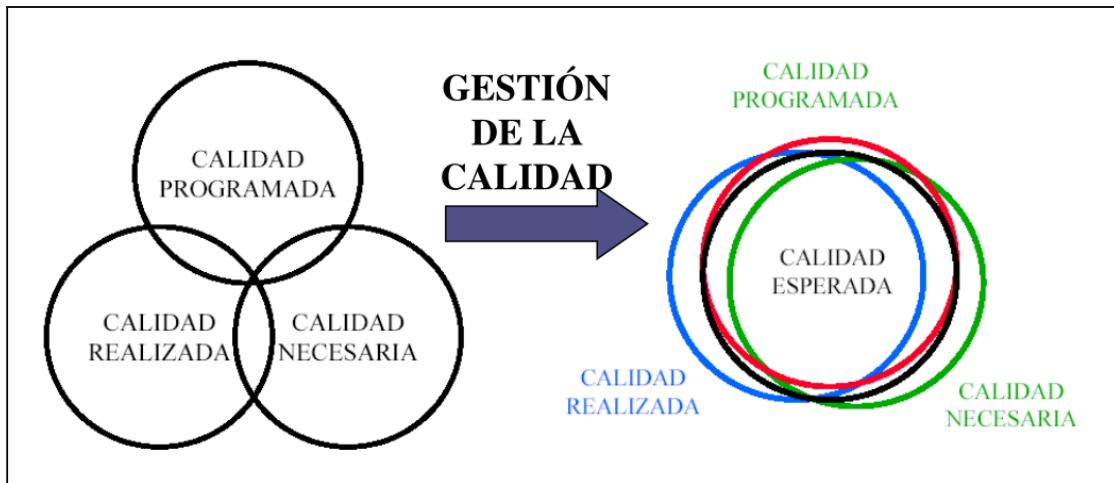
Definimos anteriormente que la calidad es el nivel de cumplimiento o adecuación a las necesidades (explícitas e implícitas) y para el caso del producto estas deberían estar explicitadas en los requerimientos.

Es por esto que los requerimientos son tan importantes, porque si no están o están mal definidos, no tenemos contra qué validar. Esto quiere decir que para que los requerimientos nos sirvan estos tienen que ser medibles (verificables) y objetivos (no ambiguos).

La **gestión de calidad en productos** son acciones sistemáticas de las organizaciones para lograr calidad, las cuales requieren equilibrio entre:

- **Calidad programada:** Los alcances del producto planificados
- **Calidad necesario:** Mínimas características que el producto debe tener, para que este satisfaga los requerimientos
- **Calidad realizada:** Lo que realmente se ha desarrollado del producto

En el equilibrio obtenemos las expectativas de calidad que esperamos del producto y todo lo que esté por fuera de esto es desperdicio o insatisfacción.



¿Qué modelos de calidad del producto tenemos?

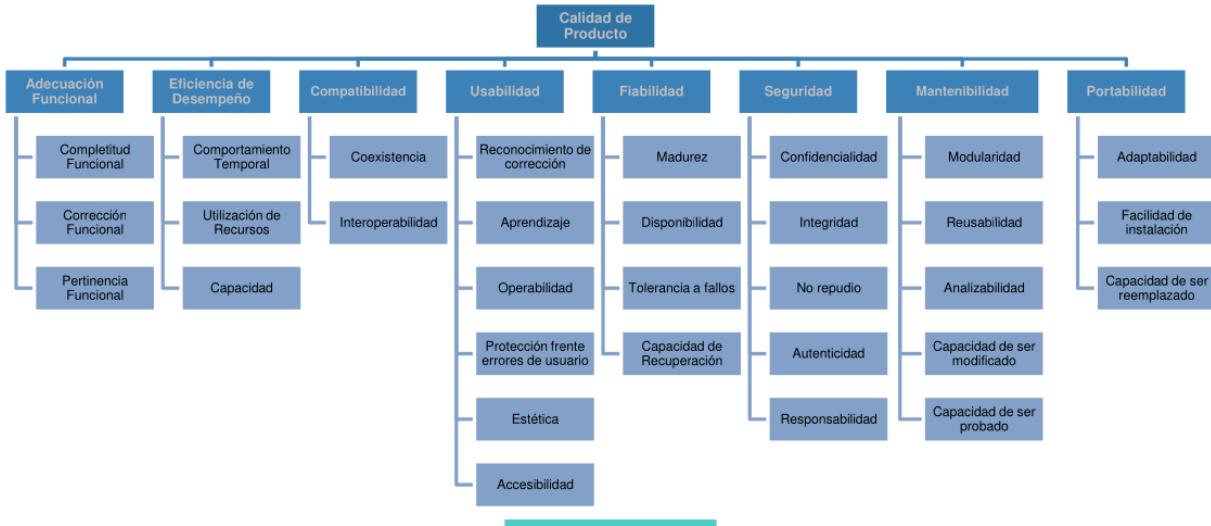
1. ISO 25000 - RNF:

Nota:

Los requerimientos no funcionales son condiciones o restricciones que definen cómo debe comportarse un sistema, en lugar de qué debe hacer.

Especifican atributos de calidad, como rendimiento, seguridad, usabilidad, disponibilidad, escalabilidad, entre otros.

Fíjate que este modelo te define **8 características**

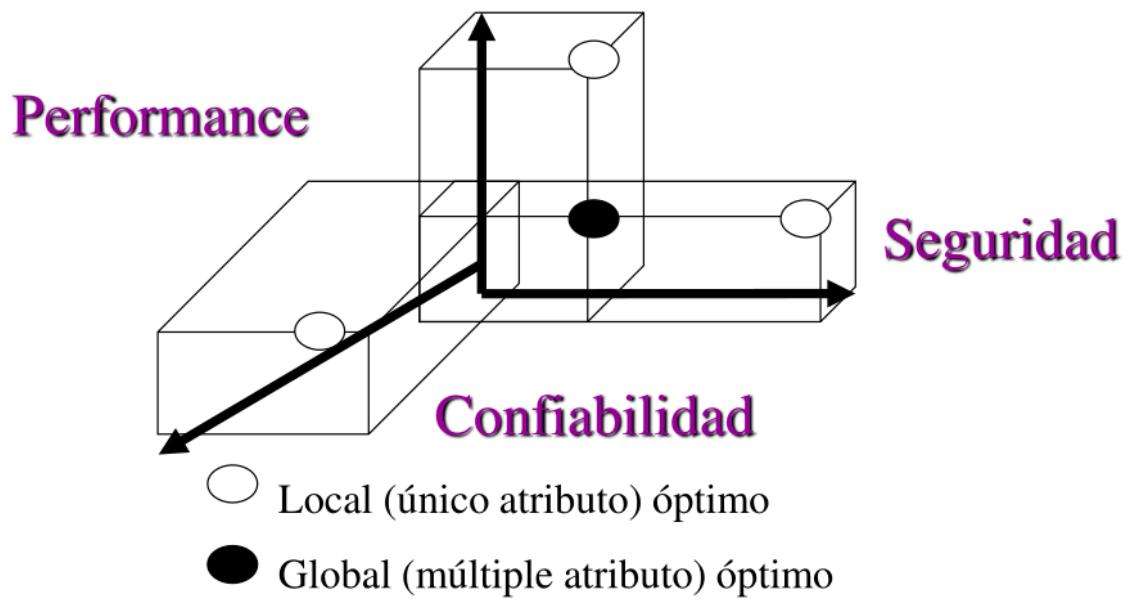


Voy a explicar algunos RNF a grandes rasgos:

- **Adecuación funcional:** Es la capacidad del software para proporcionar funciones que satisfacen las necesidades explícitas e implícitas del usuario, bajo condiciones específicas de uso.
Básicamente que el sistema haga lo que tenga que hacer
- **Eficiencia de desempeño:** Está asociada la utilización de recursos, tiempo de respuesta, etc
- **Compatibilidad:** La capacidad que tiene nuestro producto (software) para interactuar con otros sistemas
- **Usabilidad:** Facilidad con la que un usuario utiliza el sistema. Acá se incluyen varias cuestiones como que se integre a personas con discapacidades (daltonismo, dificultad para aprendizaje) o la capacidad misma que tiene la app de darte un feedback a medida que la vas usando
- **Fiabilidad:** Que tan confiable es la aplicación. Eso se suele medir por ejemplo en la cantidad de tiempo que al app funciona sin tener fallos
- **Seguridad:** Bueno lo dice el título y esta se compone de:
 - **Confidencialidad:** Asegura que la información solo sea accesible por personas autorizadas.

- **Integridad:** Garantiza que la información **no sea alterada** o modificada de forma no autorizada.
- **Disponibilidad:** Asegura que la información y los sistemas estén disponibles para los usuarios autorizados **cuando los necesiten**.
- **Mantenibilidad:** Está relacionado con que tan fácil es realizar cambios sobre el software
- **Portabilidad:** Hace referencia a que tan fácil es transferir un software de un entorno a otro
 - De un Linux a un Windows
 - Cambias el hardware
 - De navegador a celular

2. Modelo de Barbacci:



Busca el equilibrio entre la **Performance**, la **Confiabilidad** y la **Seguridad** para evaluar la calidad del producto.

Evidentemente confiabilidad y seguridad no son lo mismo:

- **Confiabilidad:** Se refiere a la capacidad del software para funcionar sin fallos durante un periodo de tiempo dado, es decir, qué tan estable y resistente es el sistema frente a errores.
- **Seguridad:** Implica la capacidad del software para protegerse contra accesos no autorizados y para asegurar que los datos y los procesos estén protegidos contra vulnerabilidades.

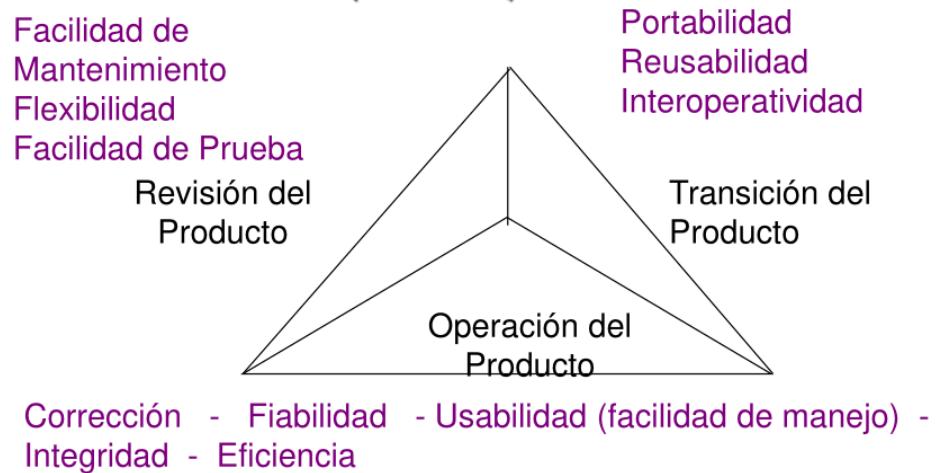
En este modelo vos podes optar por usar **puntos locales** (es decir mejorar un solo aspecto ponele tenes un super sistema seguro pero la performance no es la mejor) o usar **puntos globales** (es decir garantizar la mejoría de múltiples atributos)

3. Calidad del SW McCall:

Nuevamente la **calidad** ahora se descompone en 3 áreas principales

- **Revisión del producto:** Se asocia con la **capacidad del software a cambiar o mantenerse**
 - Facilidad de mantenimiento
 - Flexibilidad
 - Facilidad de prueba:
- **Operación del producto:** Esta se relaciona más con aspecto que percibe el usuario, funciones que las notamos cuando el software está en uso
 - Fiabilidad
 - Usabilidad
 - Integridad: Que tan bien protege el software de accesos no autorizados o la seguridad misma de los datos
 - Eficiencia: Que tan bien el sistema utiliza los recursos (cpu, ram, etc)

- **Transición del producto:** Relacionada a la capacidad de adaptarse al software a otros ambientes
 - Portabilidad: Capacidad para moverse a otro hardware/OS
 - Reusabilidad: Usar partes del software en otros productos
 - Interoperabilidad: Ya la vimos



Calidad del Proceso:

Aclaración:

- Procesos definidos:

En los procesos definidos se considera que el proceso es el único factor controlable, por lo tanto, se asume que la mejora de la calidad continua se realiza sobre el proceso. Si el proceso cuenta con calidad, entonces el producto será de calidad

Todos los modelos de calidad están basados en procesos definidos, por lo que asumen que la calidad del producto se obtiene si se tiene un proceso de calidad para construir el producto

Esto **en la práctica** no resulta tan así

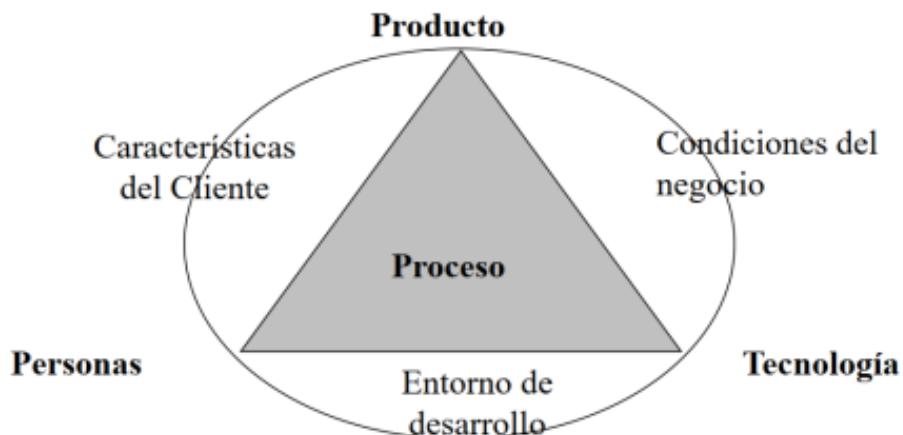
- Procesos empíricos:

Acá lo que se dice es que si el producto tiene éxito, la razón principal no es tanto un buen proceso sino **el equipo**.

Los procesos empíricos están basados en la experiencia, por lo que no consideran que la calidad en el proceso determine la calidad en el producto. Por otra parte, no están de acuerdo con las auditorías, ya que asumen que los equipos son autoorganizados. Sin embargo, la calidad en estos procesos se lleva a cabo mediante revisiones técnicas y la constante inspección y adaptación del trabajo.

Según los empíricos, **mientras que las personas hagan lo que tienen que hacer, el producto va a tener calidad**

Se insiste tanto en la calidad del proceso, debido a que es el único factor **controlable** para mejorar la calidad del software



- La tecnología y como esta avanza no podes controlarlo
- Las personas o tu **equipo de trabajo** no los podes controlar a tu voluntad

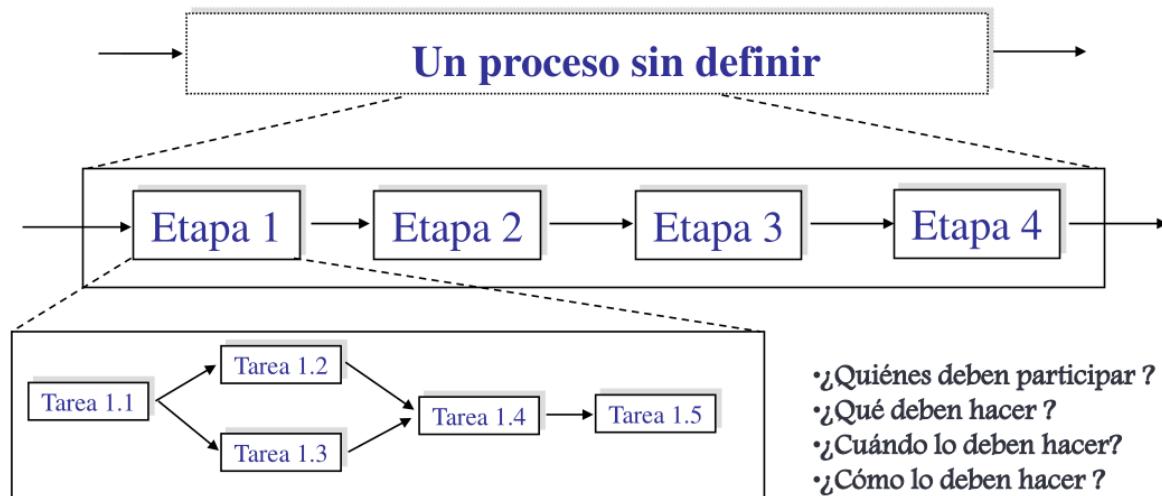
- Y tampoco podes ejercer un control sobre el producto o básicamente **las necesidades** del cliente

Se aplica calidad en el producto y en el proceso. No se habla de aseguramiento de calidad en el proyecto, dado que esta se encuentra implícita por el hecho de que el proyecto implementa un proceso.

La calidad del producto se realiza mediante actividades de revisiones técnicas y de auditorías. Estas son las herramientas principales para el seguimiento.

Definición de procesos:

Lo primero que se debe realizar en una organización para lograr tener un proceso de calidad, es **definirlo** de forma explícita y **comunicarlo** a todo el equipo, para que así todos tengan una base y el conocimiento de la forma de trabajar. Se deben definir aspectos como etapas, subetapas, roles, qué se debe hacer, en qué momentos se deben hacer, cómo lo deben hacer, etc.



Dentro de esa definición, la cual plantea **las etapas para construir la parte técnica** (Ingeniería de Requerimientos, Análisis, Diseño, Implementación, Prueba y Despliegue).

Lo importante es incorporar también estas **disciplinas de gestión y de soporte** que vimos a principios de la unidad 1

El proceso definido y adoptado, debe aportar valor al producto (es posible utilizar modelos de mejora de proceso como Kanban) y utilizarlos en los distintos proyectos de forma ADAPTADA.

La adaptación de los procesos se da en aspectos negociables del mismo. Por ejemplo, el realizar Testing no es algo negociable y que se pueda eliminar del mismo.



Administración de calidad del software:

- Concerniente con asegurar que se alcancen los niveles requeridos de calidad para el producto/proceso 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.
- **Hacer Calidad:** Insertar en cada una de las actividades del proceso acciones tendientes a detectar lo más temprano posible **oportunidades de mejora** sobre el producto y el proceso

¿Y que es el GAC?

El **GAC** o **Grupo de Aseguramiento de Calidad** es un **equipo especializado e independiente** dentro de una organización que se encarga de **verificar objetivamente que el software y los procesos de desarrollo cumplan con los estándares de calidad definidos**.

Hay ciertas **consideraciones** respecto al **reporte del grupo de aseguramiento de calidad (GAC)**

- NO debería reportar al gerente de proyectos (porque le quita independencia y libertad)
 - *Evidentemente, es muy creto decirle a tu jefe si mira hicimos todo mal, porque nos va a sacar el bono o bajar el sueldo, etc*
 - Entonces el **reporte de calidad tiene que ser independiente del reporte del proyecto**
- **No debería haber más de una posición entre la gerencia de primer nivel y el GAC**
- Cuando sea posible, el GAC debería reportar alguien realmente interesado en la calidad del software

La administración de la calidad debería estar separada de la administración de proyectos para asegurar independencia

¿Cuáles son las **actividades de administración de calidad**?

- **Aseguramiento de calidad:** Define estándares, procesos, procedimientos y modelos de calidad sobre los cuáles se van a realizar las comparaciones.

Y esto tiene sentido, si vos vas a realizar tanto auditorias o revisiones técnicas tanto en producto o proceso tenes que tener **algo con que comparar**, ya sea con algún requerimiento si se trata de un producto o de algún proceso definido si se trata de proceso

Algunas de las **funciones** del aseguramiento de calidad son:

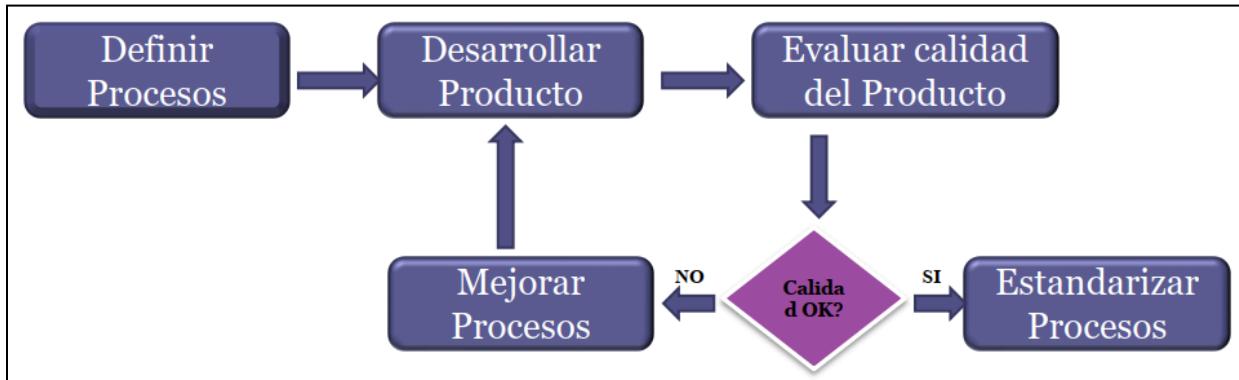
- Evaluación del **plan del proyecto** de software
- Evaluación del **proceso de diseño de software**
- Evaluación de los **requerimientos**
- Evaluación de las **prácticas de programación**

- **Planificación de la calidad:** **Selecciona los procedimientos y estándares que son aplicables para un proyecto en particular** y los modifica en caso de ser necesario
- **Control de calidad:** Es la ejecución de lo planificado, y se revisa en qué situación está el proyecto. **Asegura que los procedimientos y estándares son respetados por el equipo de desarrollo de SW.**

Acá tenemos 2 enfoques para garantizar el control de calidad:

- Revisiones de calidad:
 - Este es el principal método de validación de la calidad de un proceso o un producto.
 - Un **grupo examina parte de un proceso o producto y su documentación para encontrar potenciales problemas.**
 - Existen diferentes tipos de revisiones con diferentes objetivos:
 - Inspecciones para remoción de defectos (producto);
 - Revisiones para evaluación de progreso (producto y proceso);
 - Revisiones de Calidad (producto y estándares).
- Evaluaciones de Software Automáticas y mediciones.

Procesos con calidad:



Esta imagen describe una característica muy importante y es que no hay que olvidar que estamos trabajando con **procesos definidos**, por lo que nosotros definimos un proceso y luego de eso lo mejoramos hasta que la calidad sea de nuestro agrado

Pero una vez hecho eso este proceso se **estandariza** y se distribuye por la organización para que todos los demás equipos lo puedan usar y acá es donde **difiere ágil** con los procesos empíricos que nos dicen que la experiencia **no es extrapolable**.

Modelos de Mejora de procesos:

Algunos modelos para la mejora de procesos son:

- **SPICE:** Software Process Improvement Capability Evaluation
- **IDEAL:** Initiating Diagnosing Establishing Acting Leveraging → *En este se hace incipie en cátedra*

La mejora continua de procesos significa comprender los procesos existentes y cambiarlos para incrementar la calidad del producto o reducir los costos y el tiempo de desarrollo. Los procesos definidos están de acuerdo con esta definición, ya que consideran que la calidad del producto final depende de la calidad del proceso

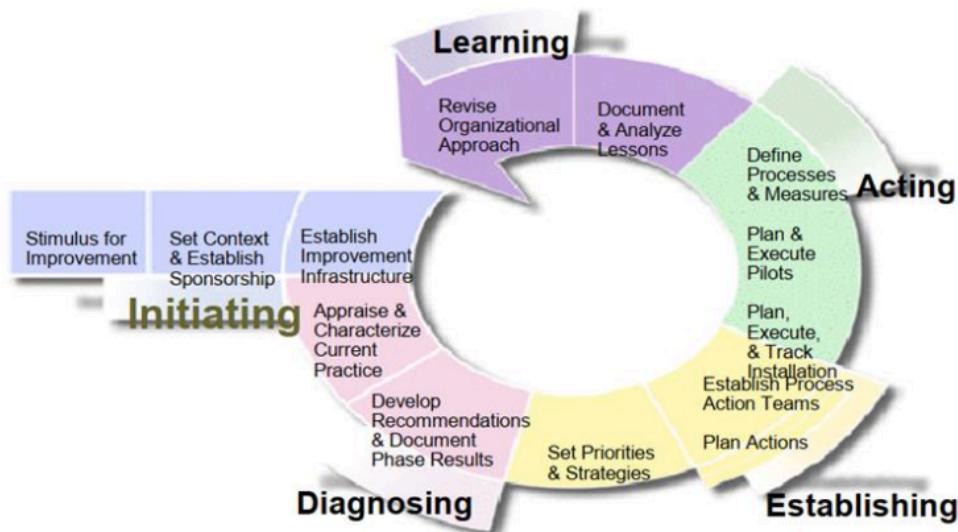
Los modelos NO te dicen cómo hacer las cosas. Son modelos descriptivos

El **propósito** de los modelos de mejora es **analizar el proceso que tiene la organización y armar un proyecto cuyo resultado, en lugar de ser un producto, es un proceso**

(definido) mejorado que se vuelca de nuevo a la organización con la idea de que se produzca un producto mejor.

Todos estos modelos parten de la filosofía de que si tengo un proceso mejorado de calidad el resultado es un producto de calidad

Modelo IDEAL:



Nos da el contexto para crear un proyecto cuyo resultado va a ser un proceso definido. Es un modelo cíclico que mejora el proceso existente en una organización. Su nombre se debe a las 5 fases que lo componen:

- **Inicialización:** Se reconocen las necesidades de cambio en la organización, razones para iniciar, determinar metas buscadas al proponer un cambio en el proceso.

Es clave que para los proyectos de mejora de procesos se tenga un **sponsoreo o aval** de la alta gerencia y esto sucede porque a menudo siempre está este dicho de *bueno pero hay algo más importante que hacer antes que esto*, entonces con esa lógica nunca se termina mejorando el proceso

- **Diagnóstico:** Establecer la madurez actual de la organización y los riesgos asociados al proceso de mejora (revisar estado actual y futuro de la org.).

- Acá es donde entra en juego el modelo **SPICE**; este nos ayuda a determinar el nivel de madurez de la organización

Para realizar el diagnóstico resultan claves **herramientas como auditorías, métricas, etc**

- **Establecimiento:** Durante la fase se elabora un **plan de acción** detallado con **acciones específicas, entregables y responsabilidades** para el programa de mejora basado en los resultados del diagnóstico y en los objetivos que se quieren alcanzar.

También se definen prioridades y estrategias

- **Ejecutar / Acción:** Efectuar los cambios y reunir información para aprender de **la mejora**. Se implementan las acciones planeadas en un **proyecto piloto** (no en todos los procesos de la organización, ya que es una prueba). Si la solución es satisfactoria para la organización, se implanta en la empresa.
- **Aprendizaje:** Busca garantizar que el próximo ciclo sea más efectivo. Durante la misma se revisa toda la información recolectada en los pasos anteriores y se evalúan los logros y objetivos alcanzados para lograr implementar el cambio de manera más efectiva y eficiente en el futuro. **Extrapolar la mejora al resto de proyectos en caso de que sea exitosa la ejecución o realizar correcciones en caso de que fracase el proyecto piloto.**

Modelo SPICE:

El modelo SPICE o también conocido como la norma **ISO 15504** es un modelo para la mejora, evaluación de los procesos de desarrollo, mantenimiento de sistemas de información y productos de software.

Tiene una arquitectura basada en **dos dimensiones**, la de procesos y la de capacidad

Con respecto a la dimensión de **procesos**, los mismos se pueden agrupar dentro de una serie de categorías las cuales son:

Procesos Primarios:

- ACQ: Procesos de Cliente

- SPL: Procesos de Proveedor
- ENG: Ingeniería
- OPE: Procesos de operación

Procesos de soporte

- SUP: Soporte

Procesos de organización

- MAN: Procesos de Gestión
- REU: Procesos de Recursos humanos
- RIN: Procesos de Infraestructura
- PIM: Procesos de mejora de procesos
- Dimensión de la capacidad

La otra dimensión evalúa justamente la **capacidad de los procesos** y la misma se encuadra dentro de:

- Nivel 0: Incompleto
- Nivel 1: Realizado
- Nivel 2: Gestionado
- Nivel 3: Establecido
- Nivel 4: Predictable
- Nivel 5: En optimización

Para cada nivel existen unos atributos de procesos estándar que ayudan a evaluar los niveles de capacidad.

Modelos de calidad para evaluar procesos:

Básicamente lo que hace un modelo de evaluación es medir el **grado de adherencia** de un proceso a un modelo que se ha tomado como referencia en la organización.

Por ejemplo, digamos que tomamos como referencia la norma ISO 9001 de 2015 que fija pautas de calidad para implementar un proceso. Entonces si yo llamo a una auditoría de ISO, se va a verificar que tanto mi proceso se adhiere a estas pautas ya definidas

¡No es lo mismo que un estándar de proceso! El objetivo de estos modelos de calidad es acreditar que una organización tiene cierto nivel de madurez en su proceso de desarrollo adoptado, o bien que cierta área de esa organización tiene determinado nivel de madurez.

Capability Maturity Model Integration – CMMI:

Modelo Integrado de Capacidad y Madurez

Es un **modelo de calidad** para la mejora y evaluación de procesos de una organización

Es un **marco de referencia** desarrollado por el Software Engineering Institute (SEI). Que sea un marco de referencia quiere decir que es un **modelo descriptivo**, CMMI no te dice **cómo hacer las cosas**, sino qué deberías lograr para mejorar tus procesos.

La **acreditación** se logra evaluando de manera objetiva (comparación con el modelo de evaluación SCAMPI) y subjetiva (Experiencia y pensamiento del evaluador) **el proceso, y el uso de este proceso instanciado en proyectos.**

CMMI **no certifica** empresas de la misma manera que, por ejemplo, la ISO 9001. Lo que hace CMMI es evaluar la madurez de los procesos de una organización mediante una "Appraisal" (evaluación formal), y como resultado, la organización puede obtener un nivel de madurez (del 1 al 5) reconocido internacionalmente.

¿Y qué es SCAMPI?

SCAMPI (Standard CMMI Appraisal Method for Process Improvement) es el **método oficial** utilizado para evaluar los procesos de una organización según el modelo CMMI. Es una herramienta clave para **determinar el nivel de madurez o capacidad** que tiene una organización en sus procesos de desarrollo, gestión o servicios.

SCAMPI propone **3 tipos de evaluaciones** (A,B,C) que van desde muy rigurosas hasta menos rigurosas

Constelaciones:

Existen 3 constelaciones o modelos de negocio que cubren los modelos de CMMI, pero básicamente **los podemos entender como ámbitos de mejora** (Nosotros el que utilizamos más comúnmente y también en la cátedra es el **dev**)

- **Desarrollo (CMMI-DEV):**
 - Provee la **guía para medir, monitorear y administrar los procesos de desarrollos de software**
 - Tiene **dos representaciones, por etapas y continua**
- **Adquisición (CMMI-ACQ):**
 - Provee la **guía para permitir seleccionar, administrar y adquirir productos y servicios a otra empresa que posee su propia forma de trabajo, delegando el control de ese proyecto para obtener el producto o servicio final.**
 - Es decir, cuando la empresa no hace cosas, sino que contrata gente que haga las cosas/trabajos por ellos (no es lo mismo que subcontratación de personal).
- **Servicios (CMMI-SVC):** Provee la **guía para entregar servicios, externos o internos.**
 - Por ejemplo: Una compañía telefónica puede acudir a una certificación de CMMI-SVC para evaluar la calidad de su proceso de atención al clientes (cantidad de gente a la que se le resuelve el problema, tiempo de respuesta, etc)

Representaciones CMMI-DEV:

Por etapas:



CMM (el antecesor a CMMI), se basaba mucho en por etapas. Identificaba a las organizaciones y las dividía en maduras (eran las del nivel del 2 al 5) e inmaduras (nivel 1), mientras más maduras más capacidad de lograr sus objetivos, bajando sus riesgos.

La **ventaja** es que como analizaba a toda la madurez de la organización, vos después podías ir y comparar distintas organizaciones según este parámetro

Es necesario cumplir con los lineamientos definidos para cada área de proceso de los niveles inferiores, y de las áreas de proceso definidas en el nivel que se está acreditando. Es decir, la acreditación es acumulativa para los niveles inferiores.

Entonces se distinguen los siguientes niveles:

- **Inicial:**
 - Procesos ad hoc y caóticos, no hay prácticas consistentes. No hay ninguna estructura y no se cumplen los objetivos
 - El éxito depende de algunos **héroes**, que terminan asumiendo mucha **responsabilidad** y salvando a todos
 - No se utilizan **métricas**

Nota:

Un proceso ad hoc es una solución que se crea para un fin o problema específico, y no se puede utilizar para otros propósitos. Se trata de una solución temporal que no suele formar parte de los procedimientos formales de una organización

- **Administrado:**

- Acá los procesos están caracterizados por proyectos los cuales se pueden **gestionar** de manera muy básica. Hay un enfoque en los requerimientos, la planificación y el seguimiento

Esto quiere decir que hemos establecido algunos procesos básicos pero no son comunes a todos los proyectos de la organización por el momento, quizás solo algunos

- Empiezan a utilizarse métricas pero de una forma muy básica
- Si bien tenemos una planificación mínima. este nivel sigue siendo **reactivo**

- **Definido:**

- En este nivel, la organización ha definido **procesos estandarizados y documentados** para el desarrollo de productos y servicios, que son implementados de manera consistente en todos los proyectos.
- También se realiza **formación del personal**, técnicas de ingeniería más detalladas y un nivel más avanzado de métricas en los procesos. Se implementan **técnicas de revisión de a pares** a fin de poder prevenir errores de manera temprana
- Acá ya se empiezan a implementar lo que serían las **verificaciones y validaciones** lo cual nos permite ir aumentando la calidad

- **Cuantitativamente administrado:**

- Se caracteriza porque las organizaciones disponen de un **conjunto de métricas significativas de calidad** y productividad, que se usan de modo sistemático para la toma de decisiones y la gestión de riesgos. El software resultante es de alta calidad.
- Se utilizan **datos estadísticos históricos para realizar una calidad predictiva**, es decir no solo solucionamos errores sino que evitamos que aparezcan
- Las medidas detalladas del rendimiento de los procesos son recogidas y analizadas estadísticamente.
- **Optimizado:** Este nivel se centra en mejora continua del rendimiento de los procesos a través de los aumentos y mejoras tecnológicas innovadoras.

Acá ya nos avocamos a **solucionar la causa raíz** de los problemas

Los objetivos cuantitativos de mejora de procesos para la organización se establecen y se revisan de forma continua a fin de reflejar los cambios objetivos de negocio, y se utilizan como criterios para la administración de la mejora de procesos. El alcanzar estas áreas se detecta mediante la satisfacción o insatisfacción de varias metas claras y cuantificables.

Una cosa interesante a destacar es que si vos estás haciendo un CMM por etapas y justo da la casualidad de que todas las áreas de proceso que presentas son las correspondientes a las de un nivel determinadas, entonces te dan las 2 acreditaciones por así decir, tanto la del nivel de madurez como la de la capacidad de los procesos

Nota:

Fíjate que a medida que vamos subiendo en el nivel de madurez, pasamos de un proceso reactivo a un proceso proactivo.

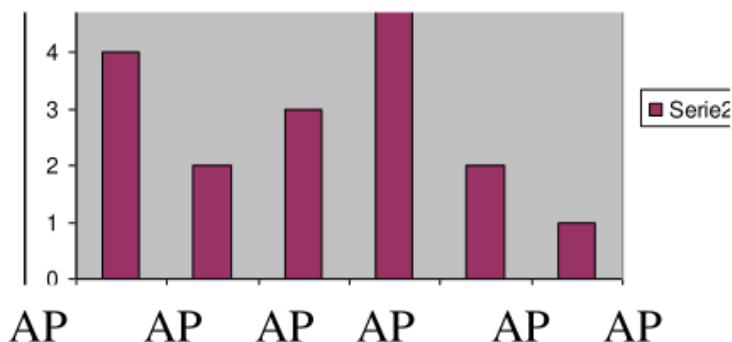
Un proceso reactivo significa que no está planificado, y todo se ve sobre la mano. En lugar de anticipar problemas o crear planes detallados para abordarlos antes de que ocurran, las organizaciones reaccionan a medida que surgen las situaciones.

Para ser de un nivel, se debe cumplir con los requisitos de ese nivel y con los de los anteriores.

Nosotros hacemos foco en el nivel 2

Continua:

Continua



El objetivo del uso de esta representación, es acreditar la capacidad de la organización ante cada una de las áreas de proceso de forma individual.

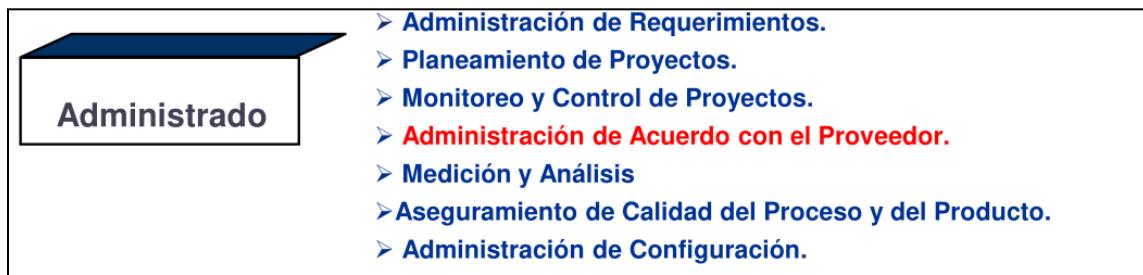
En lugar de medir la madurez de toda la organización, mido la **capacidad** de un **proceso en particular**. Procesos de nivel cero son los que no se ejecutan (Por ejemplo: si le pregunto a una empresa si hace gestión de configuración de software y me dice que no lo hace, entonces, es nivel 0 en esa área).

Apunta a mejorar las áreas de proceso que uno elige. Se centra en la mejora de un proceso o un conjunto de ellos relacionados a un área de proceso que una organización desea mejorar, una organización puede obtener la certificación para un área de proceso en cierto nivel de capacidad.

Áreas del proceso:

- Un área de proceso es un **conjunto de prácticas** relacionadas , cuando se implementan en conjunto, satisfacen un conjunto de objetivos considerados importantes para hacer mejoras significativas en el mismo proceso.

- Existen en total 22 áreas de procesos en todos los niveles de madurez, las cuales son iguales en ambas representaciones.
- 16 de esas 22 áreas de procesos son comunes a todas las constelaciones CMMI
- **Nivel 1:** No existen áreas de procesos, ya que se considera que la organización está en estado de inmadurez
- **Nivel 2:** Son 7 en total de las cuales la última es opcional (Depende si la organización realiza o no actividades relacionadas)



Administración de acuerdo con el proveedor no es obligatoria. Es necesaria si se contrata a una empresa externa como proveedora, ya que se debe comprobar que cumpla con el nivel de calidad que se tiene internamente.

Nivel	Categoría			
	Administración de Proyectos	Soporte	Administración de Procesos	Ingeniería
5 Optimizado		▪ Análisis y Causal y Resolución (CAR)	▪ Administración de Performance Organizacional (OID)	
4 Cuantitativamente Administrado	▪ Administración Cuantitativa del Proyecto (QPM)		▪ Performance del Proceso Organizacional (OPP)	
3 Definido	▪ Administración de Riesgos (RSKM) ▪ Administración Integrada de Proyectos (IPM)	▪ Análisis y Resolución de Decisión (DAR)	▪ Definición del Proceso Organizacional (OPD) ▪ Foco en el Proceso Organizacional (OPF) ▪ Capacitación Organizacional (OT)	▪ Desarrollo de Requerimientos (RD) ▪ Solución Técnica (TD) ▪ Integración de Producto (PI) ▪ Verificación (VER) ▪ Validación (VAL)
2 Administrado	▪ Administración de Requerimientos (REQM) ▪ Planificación de Proyectos (PP) ▪ Monitoreo y Control de Proyectos (PMC) ▪ Administración de Acuerdo con el Proveedor (SAM)	▪ Aseguramiento de calidad de Proceso y de Producto (PPQA) ▪ Administración de Configuración (CM) ▪ Medición y Análisis (MA)		
1 Inicial	Procesos sin definir o improvisados			

Roles - Grupos:

Los roles se adaptan a las necesidades de la organización. Cuando hablamos de **grupo** no nos referimos a conjunto de personas, sino de alguien que asuma ese rol.

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

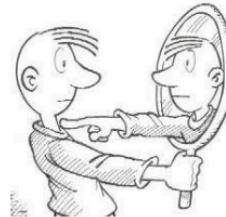
Relación de CMMI con lo ágil:

CMMI cara a cara con Ágil

- “Nivel 1”
 - Identificar el alcance del trabajo
 - Realizar el trabajo
- “Nivel 2”
 - Política Organizacional para planear y ejecutar
 - Requerimientos, objetivos o planes
 - Recursos adecuados
 - Asignar responsabilidad y autoridad
 - Capacitar a las personas
 - Administración de Configuración para productos de trabajo elegidos
 - Identificar y participar involucrados
 - Monitorear y controlar el plan y tomar acciones correctivas si es necesario
 - **Objetivamente monitorear adherencia a los procesos y QA de productos y/o servicios**
 - Revisar y resolver aspectos con el nivel de administración más alto

Referencias:

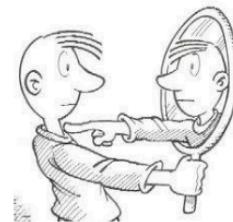
Verde : Da soporte,
Negro: Neutral,
Rojo: Desigual



CMMI cara a cara con Ágil

Referencias:
Verde : Da soporte,
Negro: Neutral,
Rojo: Desigual

- “Nivel 3”
 - Mantener un proceso definido
 - **Medir la performance del proceso**
- “Nivel 4”
 - Establecer y mantener objetivos cuantitativos para el proceso
 - **Estabilizar la performance para uno o más subprocesos para determinar su habilidad para alcanzar logros**
- “Nivel 5”
 - **Asegurar mejora continua para dar soporte a los objetivos**
 - Identificar y corregir causa raíz de los defectos



¿Por qué no coincide a medida que subimos de nivel?

CMMI dice que tienes que definir un proceso y que después lo tienes que cumplir (lo que se verifica con las auditorías).

En cambio, ágil en ningún momento evalúa que hayas seguido el proceso que definiste. CMMI a partir de nivel 3 o 4 plantea métricas/estadísticas de los proyectos y productos, y en base a la comparación de esas medidas se arma una medida del proceso. Ágil plantea que la experiencia no es extrapolable a otros proyectos.

Revisiones Técnicas:

Verificación y Validación:

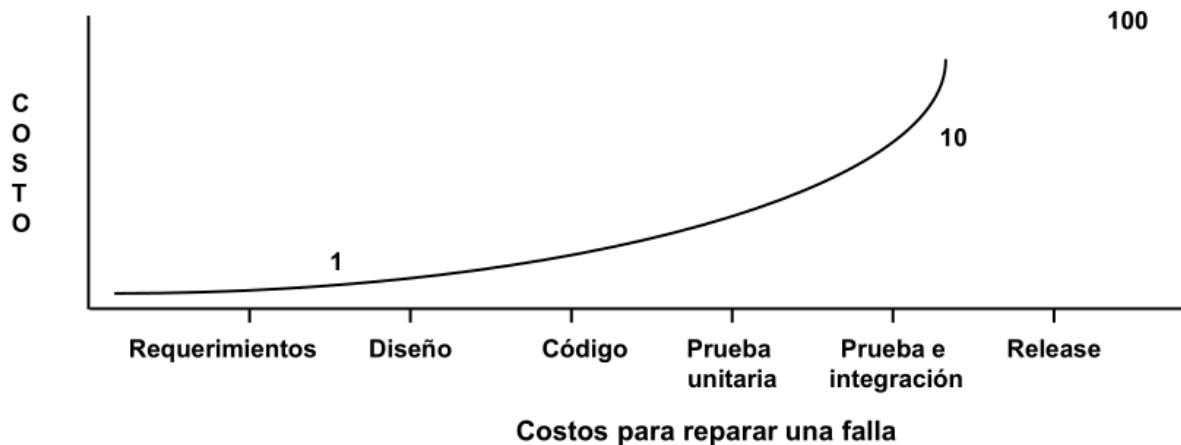
Introducción:

- Es un **proceso de ciclo de vida completo**
- Inicia con las revisiones de los requerimientos y continúa con las revisiones del diseño, inspecciones del código hasta la prueba.

- Validación: ¿Estamos construyendo el producto correcto?
- Verificación: ¿Estamos construyendo el producto correctamente?
- **Falla:** Error en un producto de trabajo
 - Acordate que si el error es detectado en la misma etapa en la que lo encuentro es un error (Caso contrario sería un defecto)
- **Producto de trabajo:** Salida de cualquier actividad correspondiente al ciclo de vida del desarrollo

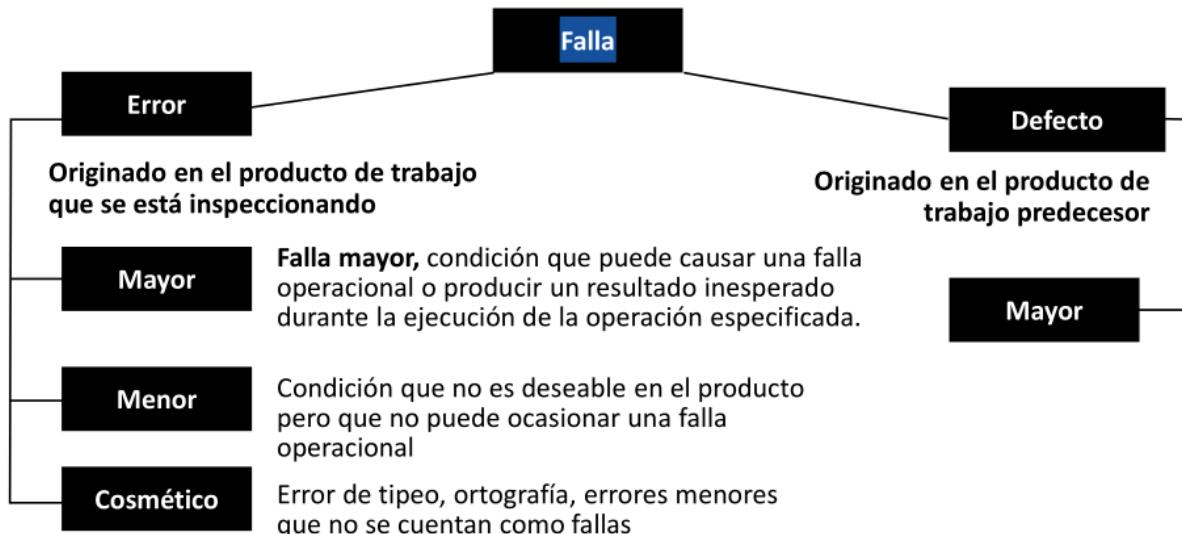
Es importante planificar !

- Es un proceso caro
- Se debe comenzar en etapas tempranas
- Fijate que vos no vas a agarrar la etapa de requerimientos e implementar testing porque no podes hacer testing ahí. Lo que se hace es implementar revisiones técnicas (que son una forma de hacer verificación y validación)



Fijate que este gráfico explica muy bien que a medida que vamos pasando por las distintas partes de un proyecto, el costo va aumentando

¿Y cuál era la diferencia entre error y defecto?



Principios:

- La **prevención** es mejor que la cura
- **Evitar** es más efectivo que eliminar
- La **retroalimentación** enseña efectivamente
- Priorizar lo **rentable**
- Olvidarse de la **perfección**, no se puede conseguir
- **Enseñar a pescar**, en lugar de dar el pescado

Revisiones Técnicas - Peer Review:

Es una actividad realizada por un **colega**, cuyo propósito es mejorar la calidad de software, mediante la detección temprana de errores en cualquier artefacto que se

genere, por ejemplo, en el código, requerimientos, diseño, arquitectura, riesgos, estimaciones, planes, etc.

Es un proceso **estático** de validación y verificación; y no corrige errores.

- **Estático** significa que no se analiza el producto en estado de ejecución (es decir no se necesita ejecutar el código fuente por ejemplo) ya que se analiza de a partes o de fragmentos de manera visual

Nunca se pone en juicio el autor del artefacto, sólo el artefacto en sí, ya que es necesario que se revelen todos los errores entre los miembros del equipo. Se debe desarrollar una cultura de trabajo que brinde apoyo y no buscar culpables cuando se descubran errores.

Es una actividad que trascendió cualquier metodología, filosofía y en muchas ocasiones la utiliza ágil, para transformar las auditorías en peer reviews evitando así traer a alguien externo al equipo.

Quiero recalcar esto de que el proceso este de verificación y validación se realiza **a lo largo del ciclo de vida** del producto, con lo cual tendríamos:

- Revisiones en la **etapa de requerimientos**; quizas algun requerimiento esté mal explicado o sea demasiado ambiguo
- Revisiones en la **parte de diseño**; quizás no se entiende bien la interacción entre algunos componentes
- A nivel **código** podríamos realizar revisiones acerca de patrones de diseño que no sean respetados

Por ejemplo, siendo un programador Jr, solicitar a un Sr una revisión técnica respecto a un patrón de diseño arquitectónico que se quiere implementar.

Ventajas:

- Pueden descubrirse muchos errores
- Pueden inspeccionarse versiones incompletas
- Pueden considerarse otro atributo de calidad

Desventajas:

- Es difícil introducir las inspecciones formales
- Sobrecargan al inicio los costos y conducen a un ahorro solo después de que los equipos adquieran experiencia en su uso
- Requieren tiempo para organizarse y parecen ralentizar el proceso de desarrollo

Costos:

- Infraestructura:
 - Entrenamiento
 - Desarrollo / ajuste de plantillas e informes
 - Desarrollo / ajuste de guías de lectura
 - Implementación de programas de medición
 - Herramientas de soporte
- Operacionales:
 - Tiempo individual y grupal
 - Tiempo en completar informes
- Adicionales:
 - Preparar material, arreglar calendario, recolectar datos, etc
 - Tiempo dedicado a la mejora de calidad

Existen 2 tipos de revisiones:

- Formales: Tienen un proceso definido con roles:
 - Inspecciones (Inspección de código de Fagan e inspección de Giib)
- Informales: Cuando no existe un proceso de como realizarlo
 - Walkthrough o recorrido

Walkthrough o recorrido (Informal)

No existe un proceso formal. Consiste en **reuniones informales** de colegas donde se debaten las correcciones a aplicar al producto de trabajo. No hay control del proceso.

Aquí los participantes formula preguntas y realizan comentarios acerca de posibles errores, violación de estándares de desarrollo y otros problemas

Tiene los siguientes **objetivos**:

- **Mínima sobrecarga** → Se trata de que la revisión no sea muy entorpecedora y les quite mucho tiempo de trabajo
- **Capacitación de los desarrolladores** → Permiten a los desarrolladores aprender sobre la marcha acerca de los errores que se van cometiendo
- **Rápido retorno**

Esta técnica no obtiene métricas para aprender y dejar registros. Sin embargo, es una de las técnicas más elegidas en los enfoques ágiles. Hay una junta de revisión entre colegas después de completar cada iteración del software (una revisión rápida), en la que pueden exponerse los conflictos y problemas de calidad sobre el producto.

Inspecciones (Formal):

Tiene un proceso formal y cuenta con un conjunto de roles. Es necesario la utilización de un checklist, que ayuda a la memoria para saber qué cosas controlar. Se toman métricas y finalmente se realiza un reporte de la revisión al final de la inspección para analizar los defectos encontrados.

Algunos de los **objetivos** de las inspecciones son:

- Detección de errores
- Hacer tus proyectos más manejables

- Verificar que el software alcanza ciertos requisitos

Son procesos time boxing y exigen un alto esfuerzo intelectual.

Roles participantes:

- **Autor:** Creador o encargado de mantener el producto a inspeccionar. Inicia el proceso seleccionando a un moderador y junto a este eligen al resto de los roles. Entrega el producto a ser inspeccionado al moderador.
- **Moderador:**
 - Planifica y lidera la revisión
 - Trabaja junto al autor para elegir los demás roles.
 - Entrega el producto a inspeccionar al inspector 2 días antes de la reunión
 - Coordina la reunión de forma que no ocurran conductas inapropiadas
 - Realiza un seguimiento de los defectos encontrados.
- **Anotador:** Registra los hallazgos de la inspección. Usualmente termina confeccionando el **reporte de la revisión**.
- **Lector:** Lee el producto a ser inspeccionado. Este rol es necesario para que los participantes no se dispersen.
- **Inspector:** Es el encargado de examinar el producto y todos pueden asumir este rol

Importante recalcar es que al margen de todo esto, cada integrante de la reunión tiene una copia del producto para que todos estén al tanto de lo que está pasando

Etapas del proceso de inspección: → **PPI/C**

- **Planificación:** El moderador, a pedido del autor, planifica la inspección definiendo el lugar, el tiempo de duración y los roles. La duración de las

reuniones no debe superar las 2 horas, dado que la inspección es un proceso de alto esfuerzo intelectual.

- **Preparación:**

- Cada rol adquiere una copia del producto de trabajo que deberá leer y analizar, con el fin de encontrar potenciales defectos.
 - Esto permite que la reunión fluya más eficientemente

- **Inspección:**

- El equipo realiza un análisis para recolectar los potenciales defectos previos y descartar falsos positivos. El lector lee el producto de trabajo y los inspectores comparten los defectos encontrados, los cuales son registrados por el anotador.
 - La reunión finaliza con una conclusión acerca de si se acepta o no el producto de trabajo inspeccionado.
 - Finalmente se realiza un informe detallando que se revisó, por quién, que se descubrió y que se concluyó.
- **Corrección:** Finalizada la reunión, el autor realiza las correcciones de los defectos encontradas

Puede pasar que los errores sean muy graves, con lo que es necesario realizar una segunda vez la etapa de inspección.⁷

¿Cuáles son algunas métricas sugeridas para las revisiones técnicas?

Métricas Sugeridas	Fórmula
Densidad de defectos	Total de defectos encontrados / tamaño actual
Total de defectos encontrados	Defectos.Mayor + Defectos.Menor
Esfuerzo de la inspección	Esfuerzo.Planning + Esfuerzo.Preparación + Esfuerzo.reunion + Esfuerzo.Retrabajo
Esfuerzo por defecto	Esfuerzo.Inspeccion / Total de def encontrados
Porcentaje de reinspecciones	Cantidad Reinspecciones / Cantidad Inspecciones
Defectos Corregidos sobre Total de Defectos.	Esfuerzo.Inspeccion / tamaño actual

¿Qué tipos de documentos hay en revisiones técnicas?

Tipo de documento	Revisores sugeridos
Arquitectura o Diseño de alto nivel	Arquitecto, analista de requerimientos, diseñador, líder de proyecto, testers.
Diseño detallado	Diseñador, arquitecto, programadores, testers
Planes de proyecto	Líder de proyecto, stakeholders, representante de ventas o marketing, líder técnico, representante del área de calidad,
Especificación de requerimientos	Analista de requerimientos, líder de proyecto, arquitecto, diseñador, testers, representante de ventas y/o marketing
Código fuente	Programador, diseñador, testers, analista de requerimientos
Plan de testing	Tester, programador, arquitecto, diseñador, representante del área de calidad, analista de requerimientos

Auditorías de Software:

Auditoría de Calidad de Software:

Concepto:

Las auditorías son **evaluaciones independientes** (Esto significa que es realizada por un grupo de personas ajenas al equipo de trabajo) de los productos o procesos de software para asegurar el cumplimiento con estándares, lineamientos, especificaciones y procedimientos, basada en un criterio objetivo.

Fijate que a diferencia de las revisiones técnicas, las auditorias buscan la **máxima objetividad**, la cual es lograda cuando la evaluación la realiza alguien que no pertenece al proyecto

Son un instrumento para el Aseguramiento de Calidad en el Software.

Las auditorías se clasifican en:

- **Auditoría física de configuración (PCA):**

- Asegura que lo que está indicado para cada **ítem de configuración** en la línea base o en la actualización se ha alcanzado rezago que definimos en el **plan de configuración** para cada IC, con lo que tengo en el repositorio

Con respecto al código, lo que se hace se **comparar al mismo con la documentación de soporte**.

Verificar que todos los artefactos (documentos, código, manuales, pruebas, etc.) **estén entregados y organizados**.

- Nivel general: Se compara el código con la documentación de soporte y se verifica que sea consistente
- Verificación: Asegura que un producto cumple con los objetivos pre establecidos, definidos en la documentación de líneas base (línea base). Todas las funciones son llevadas a cabo con éxito y los test cases tengan status “ok” o bien consten como “problemas reportados” en la nota de release.
- **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**.

Valida que el producto a construir sea consistente con los requerimientos identificados en la ERS Se usa una matriz de rastreabilidad para encontrar donde se implementan cada requerimiento

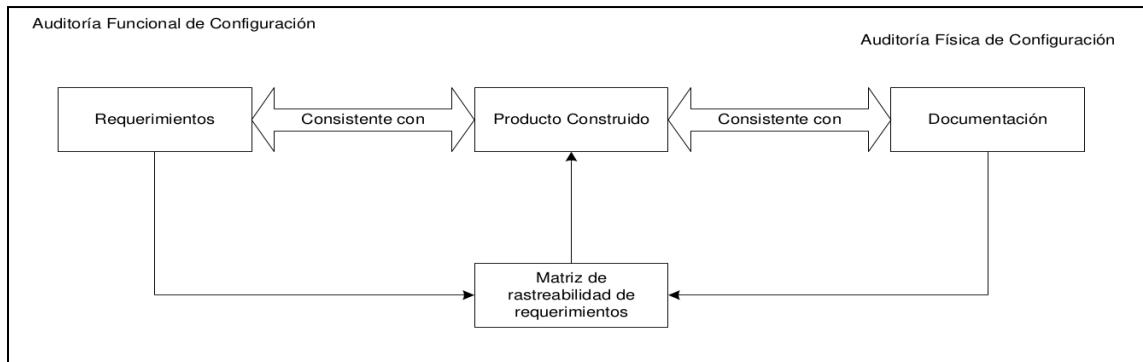
Validación: El problema es resuelto de manera apropiada de manera que el usuario obtenga el producto correcto

- **Auditorías de Proyecto**

- **Valida el cumplimiento del proceso de desarrollo.** Básicamente estamos verificando que el proyecto se ha ejecutado con el proceso que se dijo que se iba a ejecutar
 - ¿Y donde está definido qué proceso voy a utilizar? → **Usualmente en el plan del proyecto**
- **Las auditorías de proyecto se llevan a cabo de acuerdo a lo establecido en el PACS (Plan de Aseguramiento de Calidad de Software).**
- El PACS debería indicar la **persona responsable de realizar estas auditorías.**
- Las inspecciones de software y las revisiones de la documentación de diseño y prueba deberían incluirse en esta auditoría.
- El **objetivo** de esta auditoría es **verificar objetivamente la consistencia del producto** a medida que evoluciona a lo largo del proceso de desarrollo
- Supongamos que implementamos una metodología tradicional como waterfall o modelo en V
 - **Roles:**
 - ¿Quién es el líder del proyecto designado?
 - ¿El líder hace lo que tiene que hacer?
 - ¿Cuáles son los roles definidos?

- **Documentación**

- Requerimientos → ERS
- Diseño → SDD (Software Design Document)
- ¿Cada requisito de la ERS tiene trazabilidad hacia el diseño, código y pruebas?



Roles:

- **Gerente de SQA:**

- Prepara el plan de auditorías
- Calcula el costo de las auditorías
- Asigna recursos
- Responsable de resolver las no-conformidades

- **Auditor:**

- Acuerdo la fecha de la auditoría
- Comunica el alcance de la auditoría
- Recolecta y analiza la evidencia objetiva que es relevante y suficiente para tomar conclusiones acerca del proyecto auditado
- Realiza la auditoría
- Prepara el reporte

- Realiza el seguimiento de los planes de acciones acordados con el auditado
- **Auditado:**
 - Suele ser el líder del proyecto
 - Acuerda la fecha de la auditoría
 - Participa en la auditoría
 - Proporciona evidencia al auditor
 - Contesta al reporte de auditoría
 - Propone el plan de acción para deficiencias citadas en el reporte
 - Comunica el cumplimiento del plan de acción

Proceso de Auditoría:

1. **Planificación y Preparación:** Acá es donde se solicita una auditoría (El auditado la solicita), se coordina una fecha, los participantes y se proporciona una documentación preliminar
2. **Ejecución:** Acá el auditor va realizando pregunta y los auditados responden, el auditor solicita evidencia la cual tiene que ser mostrada
Entonces a la par de todo esto el auditor va completando un checklist
3. **Generación de Reporte:** Se analizan los resultados y se genera un reporte preliminar (Después te entregan un reporte final en una fecha determinada)
4. **Seguimiento:** Acá es donde se analizan las desviaciones y el auditor va a generar un **plan de acción** para justamente corregir estas desviaciones

Técnicas utilizadas en auditorías:

- **Checklists:** El contenido general de un checklist será:
 - Fecha de auditoria

- Lista de auditados
- Nombre del auditor
- Nombre del proyecto
- Fase actual del proyecto (si aplica)
- Objetivo y alcance de la auditoría
- Lista de preguntas

Ejemplos de Preguntas – Planificación de Proyectos

- *¿Existe un plan de proyecto?*
- *¿Está actualizado el plan de proyecto?*
- *¿Existe un responsable para cada actividad?*
- *¿Se han asignado recursos para las actividades de soporte?*
- *¿Están disponibles los planes para todos los involucrados?*

- **El auditor deberá asegurarse que**
 - Los planes estén basados en los requerimientos
 - Las actividades planificadas se hayan llevado a cabo
 - Todos los involucrados se han comprometido con la última versión de los planes
 - Los cambios a los planes se hayan aprobado por todos los involucrados
 - La decisión de esos cambios se haya documentado oportunamente
 - Se han identificado y comunicado los riesgos del proyecto

26

- **Muestreo:** Consiste en seleccionar una muestra representativa de los productos y/o procesos a auditar.
- **Revisión de registros**
- **Herramientas automatizadas**

Análisis y Reporte de Resultados:

Los contenidos básicos de un **reporte de auditorías** serían:

- Identificación de la auditoría
- Fecha de la auditoría
- Auditor

- Auditados
- Nombre del proyecto auditado
- Fase actual del proyecto
- Lista de resultados
- Comentarios
- Solicitud de planes de acción 3

Tipos de resultados:

- **Buenos prácticas:** Práctica procedimiento o instrucción que se ha desarrollado mucho mejor de los esperado

Se deben reservar para cuando el auditado:

- Ha establecido un sistema efectivo
 - Ha desarrollado un alto grado de conocimiento y cooperación interna
 - Ha adoptado una práctica superior a cualquier otra que se haya visto
- **Desviaciones:** Requieren un plan de acción por parte del auditado
 - Cualquier desviación que resulta en la disconformidad de un producto respecto de sus requerimientos
 - Falta de control para satisfacer los requerimientos
 - Cualquier desviación al proceso definido o a los requerimientos documentados que cause incertidumbre sobre la calidad del producto, las prácticas o las actividades
- **Observaciones:** Sobre condiciones que deberían mejorarse pero no requieren un plan de acción
 - Opinión acerca de una condición incumplida
 - Práctica que debe mejorarse
 - Condición que puede resultar en una futura desviación

Métricas de auditoria:

- Cada organización deberá establecer las métricas más apropiadas. Algunas ejemplos serían:
 - Esfuerzo por auditoría
 - Cantidad de desviaciones
 - Duración de auditoria
 - Cantidad De desviaciones clasificadas por PA de CMMI