

Unidad 1: Ingeniería de Software en Contexto

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

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.

Sin embargo, 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.
- **Sobrecostos:** 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.
- **No conformidad con los requisitos:** El software desarrollado muchas veces no lograba satisfacer completamente las necesidades del cliente o los requisitos inicialmente especificados.

¿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:

- Metodologías ágiles
- Automatización y herramientas avanzadas como por ejemplos los entornos integrados de desarrollo (IDE), y sistemas de gestión de versiones como GIT

El concepto “**ingeniería de software**” se propuso originalmente en 1968, en una conferencia de la OTAN realizada para discutir lo que entonces se llamaba la “crisis del software” (Naur y Randell, 1969).

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.

El **enfoque sistemático** que utiliza la ingeniería de software se conoce en ocasiones como el proceso de software.

Multiperson construction of multi-version software - Parmas, 1987

De acá se desprenden dos conceptos clave:

- **Función del ingeniero:** Los ingenieros aplican teorías, métodos y herramientas de la manera más conveniente siempre tratando de descubrir soluciones a los problemas teniendo en cuenta que deben trabajar con restricciones financieras y organizacionales por lo que buscan soluciones contemplando estas restricciones
- **Disciplinas que conforman la ingeniería de software:**
 - **Técnicas:** Ayudan a construir el producto.
Ejemplo: Análisis, Diseño, Implementación, Pruebas
 - **De gestión:** Planificación, monitoreo, control
 - **De soporte:** Gestión de configuración, métricas, aseguramiento de calidad
 -

Es importante entender que la Ingeniería de Software utiliza el **paradigma orientado a objetos**, el cual es un aspecto clave para atacar la complejidad innata del software. Se destacan características como:

- **Modularidad:** Dividir el software en componentes más pequeños y reutilizables que se pueden desarrollar y probar de forma independiente.

- **Abstracción:** Ocultar los detalles de implementación de un componente y exponer solo la funcionalidad necesaria a otras partes del software.
- **Encapsulamiento:** Agrupar los datos y funciones de un objeto en una sola unidad, y proteger el estado interno de un objeto de modificaciones externas.
- **Reutilización:** Crear componentes que se puedan utilizar en múltiples proyectos, lo que puede ahorrar tiempo y recursos.
- **Mantenimiento:** Actualizar y mejorar regularmente el software para corregir errores, añadir nuevas características y abordar vulnerabilidades de seguridad.

Ejemplos de softwares:

Healthcare.gov (2013) (fracaso)

- Este fue el sitio web oficial del mercado de seguros de salud de EE. UU., lanzado como parte del Affordable Care Act (ACA).
- **Problemas:** El lanzamiento inicial estuvo plagado de problemas técnicos, con fallos en el registro de usuarios, tiempos de carga extremadamente lentos y una infraestructura que no soportaba la alta demanda. La mala gestión del proyecto, la falta de pruebas adecuadas y la presión para cumplir plazos políticos contribuyeron a su fracaso inicial.

Linux (éxito)

- Un sistema operativo de código abierto iniciado por Linus Torvalds como un proyecto personal. Ha crecido para convertirse en uno de los sistemas operativos más importantes del mundo.

Componentes de Proyecto de Desarrollo de Software

Proceso de desarrollo de Software

Somerville

Un proceso de software es una serie de actividades relacionadas que conducen a la elaboración de un producto de software.

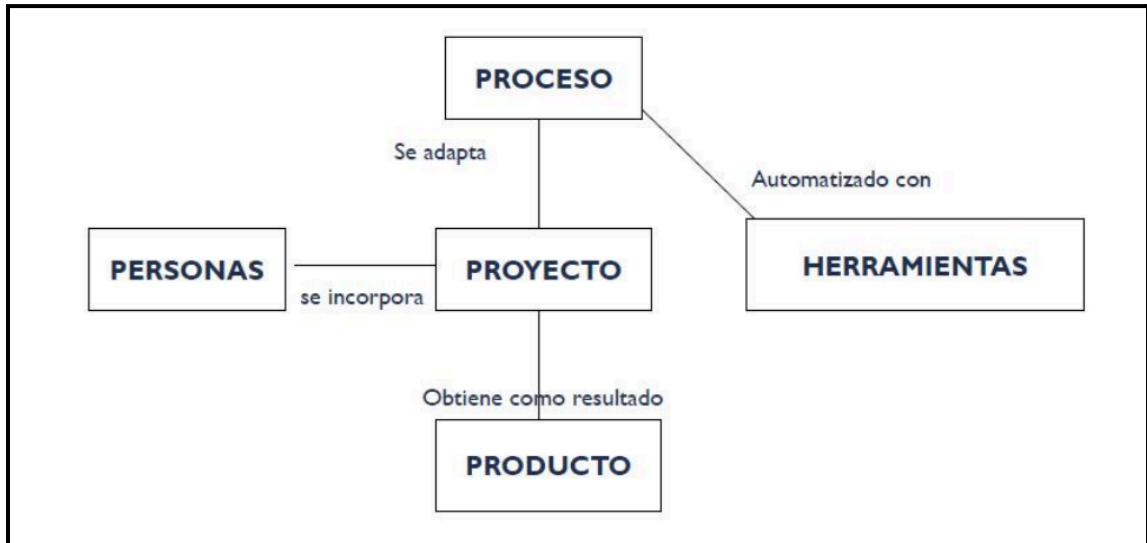
El proceso es una implementación del ciclo de vida que tiene como objetivo final construir un producto de software

Procesos definidos se pueden combinar con cualquier tipo de ciclo de vida, los procesos empíricos, como requieren de su constante inspección y adaptación solo se combinan con el ciclo de vida iterativo e incremental.

Existen muchos diferentes procesos de software, pero todos deben incluir cuatro actividades fundamentales para la ingeniería de software:

1. **Especificación de software:** Tienen que definirse tanto la funcionalidad del software como las restricciones de su operación
2. **Diseño e implementación del software:** Debe desarrollarse el software para cumplir con las especificaciones
3. **Validación del software:** Hay que validar el software para asegurarse de que cumple lo que el cliente quiere
4. **Evolución del software:** EL software tiene que evolucionar para satisfacer las necesidades cambiantes del cliente

¿Cómo entran en juego las 4 P 's?



- **Personas:** Los principales autores de un proyecto Software son los arquitectos, desarrolladores, ingenieros de prueba y el personal de gestión.
- **Proyecto:** Elemento organizativo a través del cual se gestiona el desarrollo del software.
- **Producto:** Artefactos que se crean durante la vida del proyecto.
- **Proceso:** Un proceso de ingeniería de software es una definición del conjunto completo de actividades necesarias para transformar los requisitos de usuario en un producto.

Es una plantilla que establece un conjunto de actividades que nos sirven como **guia** para border trabajar

- **Herramientas:** Software que se utiliza para automatizar las actividades definidas en el proceso

Proceso definido:

Este es básicamente el enfoque que nosotros venimos trabajando. Ejemplos de esto pueden ser RUP, PUD.

El proceso definido es un proceso en el que están bien definidos un conjunto de actividades a seguir. Dados una serie de inputs, un proceso definido produce el mismo output cada vez basado en la repetición y la naturaleza predictiva

Tradicionalmente, una vez que comienza un proyecto, se crea un paquete de requisitos y luego se «aprueba». El gerente del proyecto asume que esta aprobación da como resultado un conjunto fijo de requisitos y que ahora puede comenzar la planificación.

El **defecto fundamental** de este enfoque es que el plan, que impulsa todo, se basa en la suposición de que los requisitos son fijos y no cambiarán.

Asume que podemos repetir el mismo proceso una y otra vez, indefinidamente, y obtener los mismos resultados.

La administración y control provienen de la predictibilidad del proceso definido.

Entradas identificadas → Proceso definido → Salidas esperadas

Proceso empírico:

El enfoque ágil impulsado por el valor se basa en el **empirismo** que cambia toda la mentalidad. Asume desde el principio que los requisitos que existen por adelantado no son fijos y que cambiarán.

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

Asume procesos complicados con variables cambiantes. Cuando se repite el proceso, se pueden llegar a obtener resultados diferentes.

La administración y control se realiza través de inspecciones frecuentes y adaptaciones

Son procesos que trabajan bien con procesos **creativos y complejos**

La **experiencia no es extrapolable** → Esto quiere decir que los conocimientos 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**:

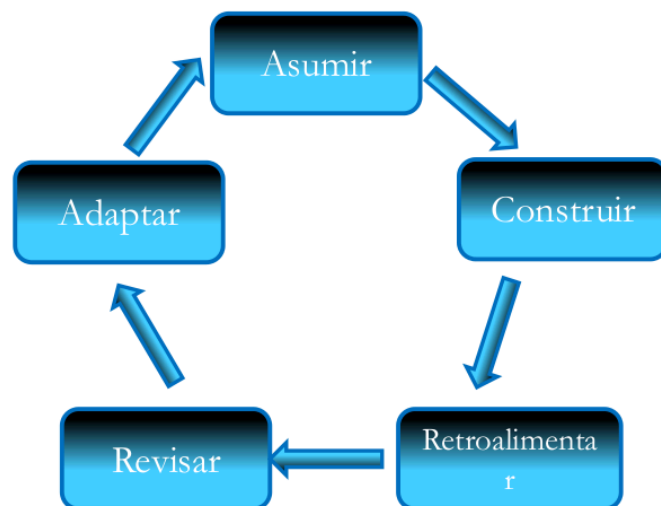
- **Transparencia:** En el centro del scrum se encuentra la transparencia, un principio fundamental que se centra en la comunicación abierta y sin obstáculos.

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 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 **patron del conocimiento** de los procesos empíricos se basa en:



Diferencias clave entre ambos procesos:

Andy

- Optimización de recursos:

En un **proceso definido** claramente si tengo las mismas entradas y buscas las mismas salidas, para poder mejorar el proceso voy a ir a cada una de las etapas intermedias y las voy a tratar de optimizar

En un **proceso empírico**, el punto de mejora del proceso difícilmente sea tan tangible, voy a tener que ir atacando diferentes artistas para poder optimizar el proceso y principalmente centrarme en las personas, que son parte del empirismo y quienes capitalizan la experiencia.

- Visibilidad y repetición:

Otra de las diferencia entre ambos procesos es la intención que tienen los procesos definidos de ser **repetibles**, y esa repetibilidad la quieren lograr para tener una visibilidad, es decir, para saber qué es lo que pueden esperar en cada momento y de esta forma predecir qué es lo que va a suceder

Básicamente los procesos definidos tienen la **ilusión del control**.

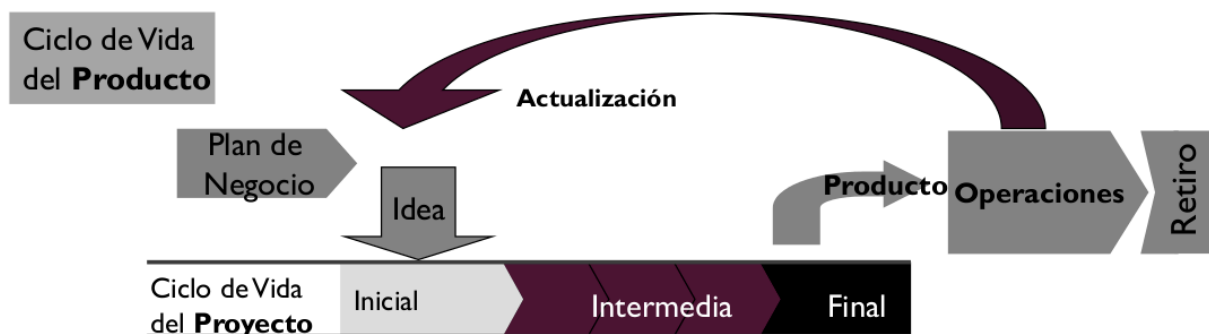
Los procesos definidos intentan ser procesos completos, que describen la mayor cantidad de cosas posibles que se deben hacer para desarrollar software, en contraposición a los procesos empíricos que son procesos que no están completos, esto deja que cada equipo al iniciar un trabajo decida basado en la experiencia que es lo que quiere hacer, cuando y como. El empirismo se basa en ciclos de entrega cortos para poder generar realimentación que sirva como experiencia para evolucionar y seguir avanzando.

Los procesos empíricos dicen que la experiencia es aplicable al mismo equipo, este equipo puede generar su propia experiencia en este proyecto, en este contexto particular para los distintos ciclos, pero la experiencia de ese equipo no se puede

extrapolar en otros equipos, proyectos, que es lo que sí esperan los procesos definidos. Los procesos definidos esperan repetibilidad.

Ciclos de Vida:

- La serie de pasos a través de los cuales el producto o proyecto progresa
- Los productos y proyectos tienen su ciclo de vida



Evidentemente el ciclo de vida del **producto** es más largo que el del proyecto. Fíjate que dentro del ciclo de vida del proyecto puede haber varios proyectos

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

Un ciclo de vida de un proyecto software es una representación de un proceso. Gráfica una descripción del proceso desde una perspectiva particular

Los modelos especifican:

- Las fases del proceso.
 - Ejemplo: Requerimientos, especificaciones, diseño
- El orden en el cual se llevan a cabo

¿Cómo se clasifican los ciclos de vida?

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

- **Secuencial:** Cada fase comienza solo cuando la anterior ha sido completada. Las fases típicas incluyen requisitos, diseño, implementación, pruebas, despliegue y mantenimiento.
 - Cascada
- **Iterativo** → En este enfoque, el desarrollo se realiza en ciclos repetidos (iteraciones). Cada iteración produce una versión incremental del software, que se revisa y refina en ciclos posteriores hasta alcanzar el producto final.
- **Recurso:** se utilizan para casos bastantes particulares como proyectos que tienen bastantes riesgos. Ejemplo: ciclo de vida en espiral que se basa en la mitigación de riesgos.

Proyecto:

Un proyecto es el **elemento organizativo** a través del cual se gestiona el desarrollo del software.

El proyecto es una **unidad de gestión**, es un medio por el cual yo administro los recursos que necesito y las personas que van a formar parte, para obtener como resultado un producto o servicio de software.

Un proyecto es llevado a cabo por **personas** que implementan **herramientas** para automatizar los procesos y que obtiene como resultado un producto.

¿Cuáles son las 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**

Duracion Limitada:

- 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:

- Todos los proyectos por similares que sean tienen características que los hacen únicos

Administración de Proyectos:

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

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

Administrar un proyecto incluye:

- Identificar los requerimientos
- Establecer objetivos claros y alcanzables

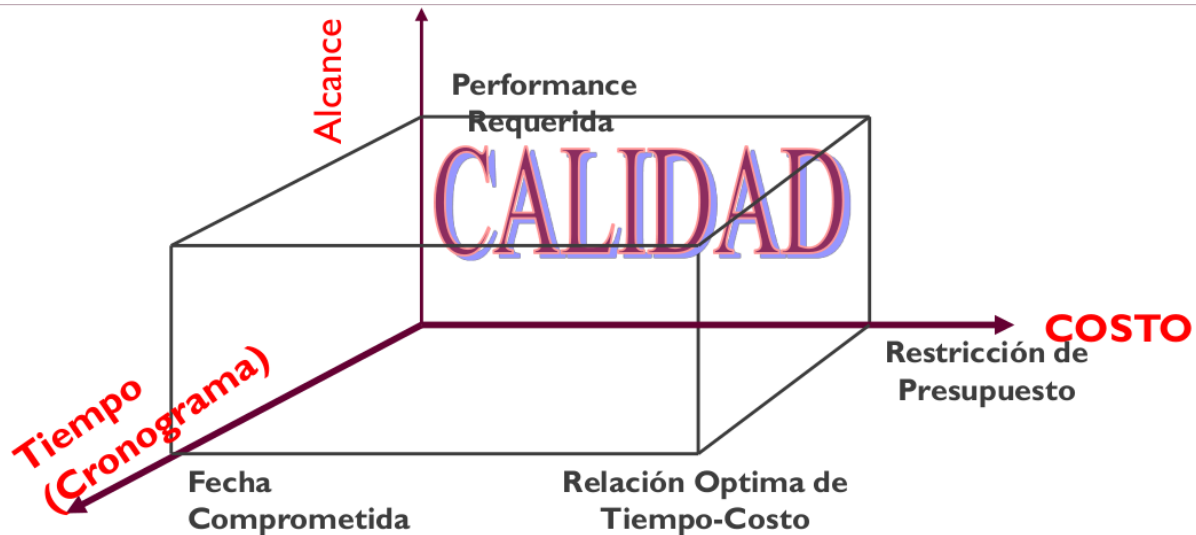
- Adaptar las especificaciones, planes y el enfoque a los diferentes intereses de los involucrados (stakeholders)

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)



Líder del proyecto

En la gestión tradicional tenemos un equipo de proyecto y un líder de proyecto. El líder es quien se relaciona con los demás involucrados; es conductista, le dice a la gente

que es lo que tiene que hacer, para cuando lo tiene que hacer, y el equipo recibe las asignaciones del trabajo y trabaja. Informa cuando ya está terminado, informa cuánto se demoró, informa el porcentaje de avance y si hay una desviación respecto a los planes informar al líder para que tome ciertas correcciones

Dentro de este enfoque, como líderes tenemos la responsabilidad de estimar, planificar, asignar trabajo a la gente y hacer seguimiento de lo que está pasando.

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.

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

El plan de proyecto:

Un plan es a un proyecto lo que una hoja de ruta es a un viaje

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
- Estimacion
- 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; 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 proyecto → Se mide contra el plan del proyecto (o plan de desarrollo de software)

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

2. Definir un ciclo de vida

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*

3. Estimaciones de Software

Estimar es predecir el trabajo, esfuerzo que será necesario realizar en un momento donde no se tiene conocimiento sobre lo que se estima.

Por definición una estimación no es precisa y acarrea cierto riesgo debido a la incertidumbre del proyecto

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

1. **Tamaño:** Definir el producto a construir.

Algunos ejemplos pueden ser:

- Líneas de código (Así se hacía antes y es muy pobre este método)
- Por casos de uso (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.
4. **Costo**
5. **Recursos críticos:** Son estos elementos esenciales para poder desarrollar el software.

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 escencia 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

5. Métricas de software:

Las métricas son útiles para saber si nuestro proyecto está en línea con lo que nosotros planificación 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
- **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

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

¿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

Factores clave para el éxito de un proyecto:

- Monitoreo y Feedback: Tener un monitoreo permanente e ir generando acciones correctivas cuando se necesario
- Tener una misión / objetivo claro
- Comunicación: FUndamental una buena comunicaciones entre el equipo, el líder y los stakeholders

Causas de fracasos en un proyecto:

- Fallas al definir el problema
- Planificar basado en datos insuficientes
- La planificación la hizo el grupo de planificaciones
- NO hay seguimiento del plan del proyecto

- Plan del proyecto pobre en detalles
- Planificación de recursos inadecuada
- Las estimaciones se basaron en “supuestos” sin consultar datos históricos
- Nadie estaba a cargo

Unidad 2: Gestión LEAN/Ágil de productos de Software

Filosofía Ágil - Manifiesto Ágil

Agilismo:

Introducción:

El agilismo es un movimiento que se gestó desde los desarrolladores.

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

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

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:



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

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

Por supuesto que los procesos ayudan al trabajo. Son una guía de operación. Las herramientas mejoran la eficiencia, pero sin personas con conocimiento técnico y actitud adecuada, no producen resultados.

- ii. **Valorar más el software funcionando que la documentación exhaustiva**

El manifiesto no afirma que no hagan falta. Los documentos son soporte del software, permiten la transferencia del conocimiento, registran información histórica, y en muchas cuestiones legales o normativas son obligatorios, pero se resalta que son menos importantes que los productos que funcionan. Menos trascendentales para aportar valor al producto.

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

Significa que se da prioridad a una relación cercana y continua con el cliente, enfocada en satisfacer sus necesidades y adaptar el proyecto según evolucionen los requisitos, en lugar de centrarse estrictamente en los términos y condiciones especificados en un contrato formal.

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

Agilismo - Concepto:

NO es una metodología o un proceso

Ágil es una ideología con un conjunto definido de principios que guían el desarrollo del producto

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 concurrente

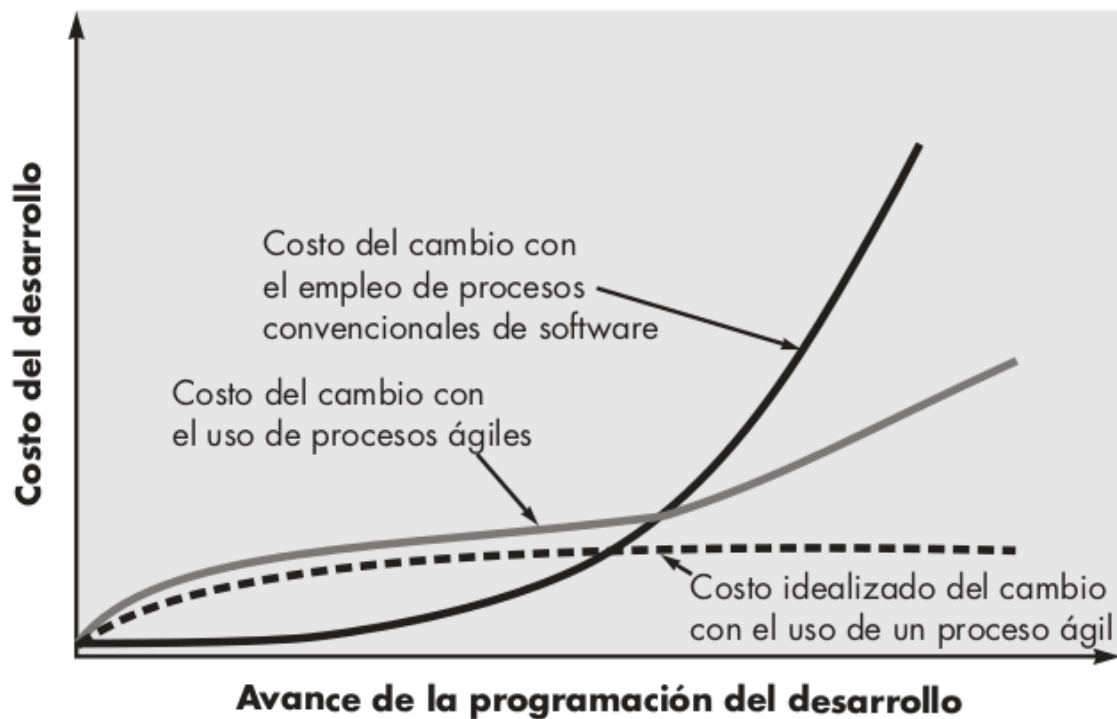
¿Pero qué significa ágil?

Balance entre ningún proceso y demasiado proceso. La diferencia inmediata es la exigencia de una menor cantidad de documentación, sin embargo no es eso lo más importante:

- Los métodos ágiles son **adaptables** en lugar de predictivos.
- Los métodos ágiles son **orientados a la gente** en lugar de orientados al 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.*

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

Requerimientos ágiles:

Vamos a identificar las siguiente **características:**

- Usan el **valor** para construir el producto correcto. Nosotros como ingenieros en sistemas somos profesionales de soporte, esto quiere decir que ayudamos a otra profesiones a **obtener algo valioso** del sistemas 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 que construir
- 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)
- Es importante ir construyendo el producto **en conjunto con el cliente**, los stakeholder tienen una mirada interesante y hay que tenerlos en cuenta

Acordate → Técnicos y no técnicos trabajando juntos

Vale la pena decir que:

- Los cambios son la única constante

- Stakeholder: no son todos los que están
- Siempre se cumple eso de que *El usuario dice lo que quiere cuando recibe lo que pidió*
- No hay técnicas ni herramientas que sirvan para todos los casos
- Lo importante no es entregar una salida, un requerimiento, lo importante es entregar un resultado, una solución que tenga **valor**

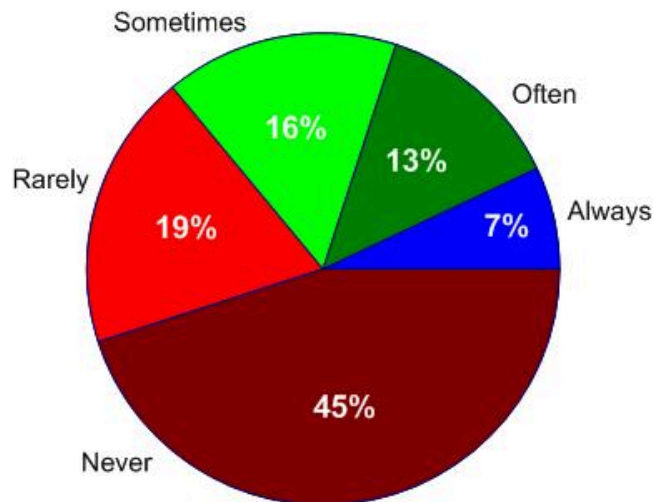
¿Y por qué queríamos cambiar el enfoque tradicional?

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
- **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 a partir de casos de uso

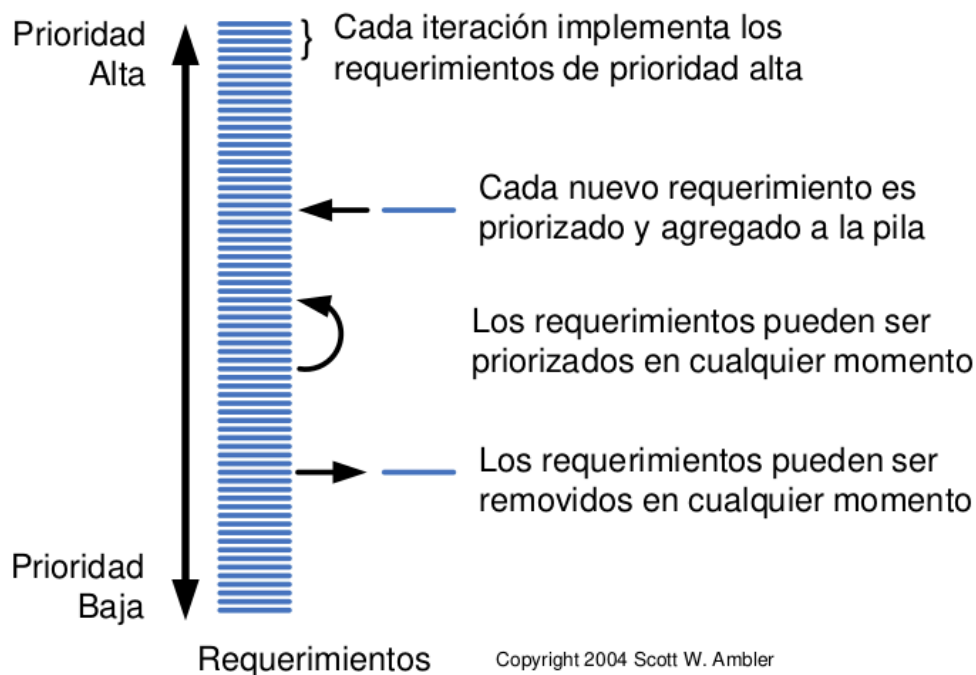
Gestión ágil de requerimientos:

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

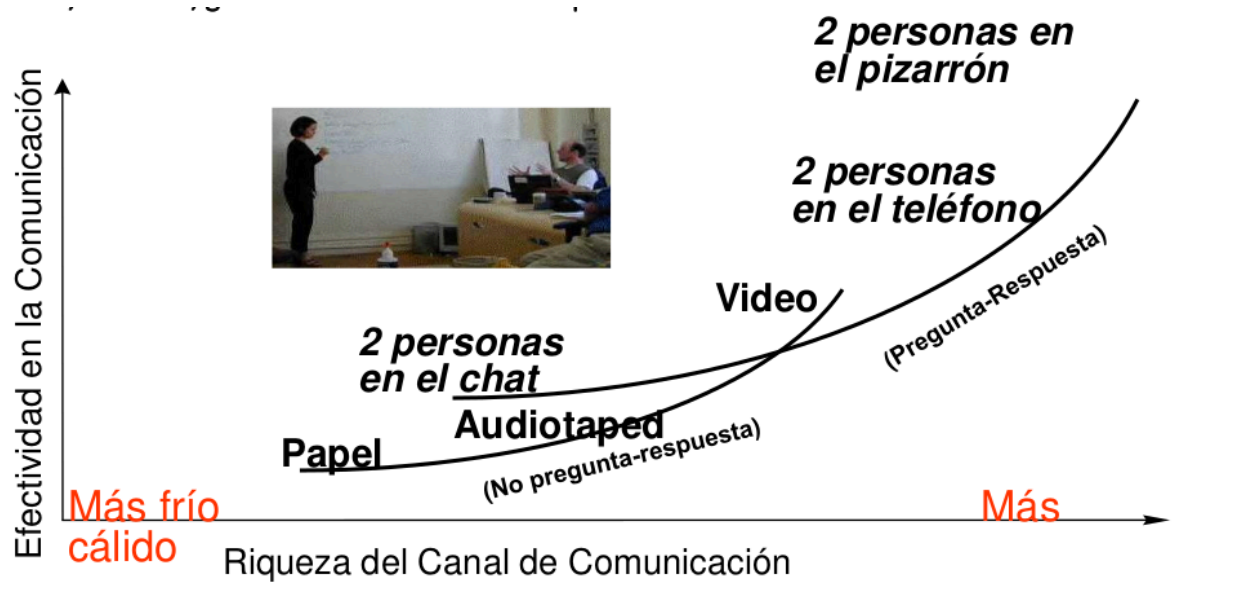
Cada ítem en el Product Backlog se denomina un **Product Backlog Item (PBI)** y debe estar descrito de manera clara para que el equipo de desarrollo entienda lo que se necesita.



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

Tradicional vs Ágil:



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

Fijate que siempre que trabajamos en un contexto de proceso **tradicional**, suelo dejar fijos los requisitos 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?**

Principios ágiles relacionados a los requerimientos ágiles:

- La prioridad es satisfacer al cliente a través de releases tempranos y frecuentes (2 semanas a un mes)
- Recibir cambios de requerimientos, aun en etapas finales
- Técnicos y no técnicos trabajando juntos todo el proyecto
- El medio de comunicación por excelencia es cara a cara
- Las mejores arquitecturas, diseños y requerimientos emergen de equipos autoorganizados

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.

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.

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

- **Tarjeta:** Es la parte visible, donde escribimos la historia de usuario. La tarjeta está compuesta por:
 - 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

- 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.

Se especifican entonces si fallan o si pasan

- Frase verbal: Suele agregarse una frase resumen de los que se trata la user story a modo de título

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
- **Confirmación** → Las pruebas de usuarios que se identifican necesarias para hacerle a la funcionalidad una vez construida la misma y que me sirven para que el product owner me acepte a mi la story

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

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.



Fijate 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 poder 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):

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 limites 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 detalles

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 especificaciones 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 construye el equipo para poder determinar que la user está en condiciones de entrar en una iteración de desarrollo. La user **está lista** cuando cumple con la definición de listo (Ready) impuesta por el equipo.

Para saber si la user **está ready** se utiliza el modelo INVEST

Definition of Done:

Nos define si la historia está de decentemente terminada para poder presentársela al Product Owner

Ultimas consideraciones:

- No son especificaciones detalladas de requerimientos (como los casos de uso)
- Son expresiones de intención, “es necesario que haga algo como esto...”
- No están detallados al principio del proyecto, elaborados evitando especificaciones anticipadas, demoras en el desarrollo, inventario de requerimientos y una definición limitada de la solución.
- Junto con el código, sirven de entrada a la documentación que se desarrolla incrementalmente después
- Tenemos **distintos niveles de abstracción**:
 - La user story normal
 - **Tema**: Una colección de historias de usuario relacionadas
 - **Épica**: Es una historia de usuario muy larga

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

Técnicas	Funcionales
<ul style="list-style-type: none"> - Utilizadas para investigar enfoques técnicos en el dominio de la solución <ul style="list-style-type: none"> - Evaluar performance potencial - Decisión hacer o comprar - Evaluar la implementación de cierta tecnología - Cualquier situación en la que el equipo necesita una comprensión más fiable antes de comprometerse a una nueva funcionalidad en un tiempo fijo 	<ul style="list-style-type: none"> - Utilizadas cuando hay cierta incertidumbre respecto de cómo el usuario interactuara con el sistema - Usualmente son mejor evaluadas con prototipos para obtener realimentación de los usuarios involucrados

Algunas user stories requieren de ambos tipos de spikes. Por ejemplo:

- Como un cliente, quiero ver mi uso diario de energía en un histograma, para poder comprender rápidamente mi consumo de energía pasado, presente y proyectado

En este caso un equipo puede crear dos spikes:

- Spike tecnico
 - Investigar cuánto tiempo requiere actualizar un display de un cliente al uso actual, determinando requerimientos de comunicación, ancho de banda y si los datos se actualizan en formato push o pull.

- Spike funcional
 - Crear un prototipo de histograma en el portal web y obtener la retroalimentación de algunos usuarios respecto del tamaño, el estilo de la presentación y los atributos gráficos.

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 opcion
- Implementar la spike en una iteración separada de las historias resultantes
 - Salvo que el spike sea pequeño y sencillo y sea probable encontrar una solución rápida en cuyo caso, spike e historia pueden incluirse en la misma iteración.

Estimaciones de Software:

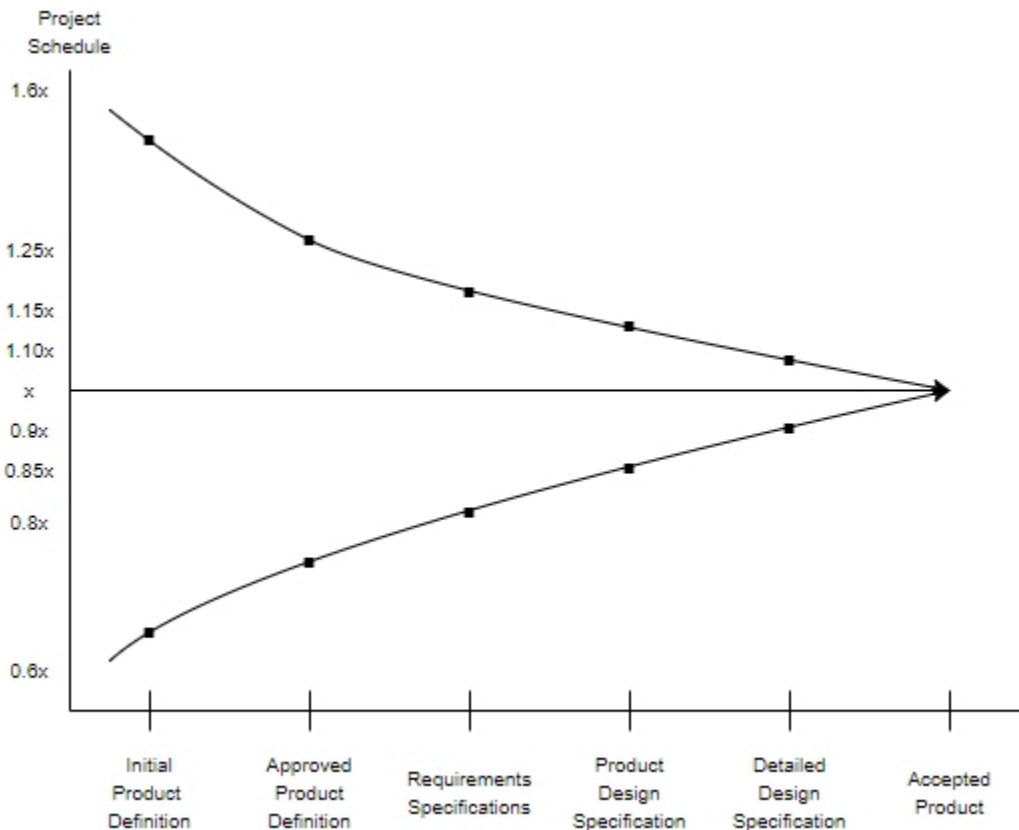
Introduccion

Algunas consideraciones:

- 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 como por ejemplo:
 - 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 sí el proceso de estimación siempre va a tener un sesgo obviamente
- **Falta información:** Entonces cuando estimamos, estamos asumiendo ciertas cosas pero que en realidad no las sabemos.
- **No tenemos claro el proceso a utilizar**

Métodos utilizados para la estimación:

- Basados en la experiencia:
 - Datos históricos

- Juicio experto
 - Puro
 - Delphi
- Analogía
- 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:

Acá partimos de la idea de que la experiencia de otros equipos en otros proyectos, a mi me puede servir.

Entonces podemos recabar información de otros proyectos que ha sido cerrados

Pero fijate una cosa, a diferencia de la analogía (que aparece después), los datos históricos utilizan información detallada y cuantitativa de proyectos anteriores.

Se basa en el análisis de registros y métricas específicas, como tiempos de entrega, costos y problemas encontrados en proyectos anteriores

Juicio experto → Puro:

- Un experto 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:

Un grupo de personas son informadas y tratan de adivinar lo que costara el desarrollo tanto en esfuerzo como en duración

Las estimaciones en grupo suelen ser 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 ajenas pero de forma 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:

Al igual que antes (datos históricos) nos basamos en información de proyectos y equipos anteriores que sabemos que han cerrado proyectos y que les han ido bien.

Sin embargo, la diferencia con los datos históricos es que utiliza la comparación cualitativa entre proyectos similares para hacer estimaciones.

No Estimate:

Una cosa importante a destacar con respecto a las estimaciones es **entender la principal diferencia entre el enfoque tradicional y las estimaciones ágiles**

Básicamente en el **enfoque ágil** estima la persona que va a realizar el trabajo, mientras que en el **enfoque tradicional** estima cualquier menos este, es decir el project manager, el Gurú estimador, etc

Otra característica del **agilismo**, es que vamos a estimar como equipo y mientras estamos en sesiones de estimación, no nos aferramos a una idea.

Tips respecto de las estimaciones:

- Si las estimaciones se utilizan como compromisos son muy peligrosas y perjudiciales para cualquier organización
- Lo más beneficioso en las estimaciones es el *proceso de hacerlas*
- La estimación podría servir como una gran respuesta temprana sobre si el trabajo planificado es factible o no
- La estimación puede servir como una gran protección para el equipo

Estimaciones Ágiles:

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.**

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

Con respecto al tamaño...

- “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**
- El tamaño indica:
 - Cuán compleja es una feature/story
 - Cuan trabajo es requerido para hacer o completar una feature/story
 - Y cuán grande es una feature/story

Tamaño vs Esfuerzo:

Las estimaciones basadas en tiempo son más propensas a errores debido a varias razones:

- Habilidades
- Conocimiento
- Asunciones
- Experiencia
- Familiaridad con los dominios de aplicación/negocio

Tamaño **no es** esfuerzo

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 asi vos en la empresa no estas todo el tiempo trabajando pueden pasar impedimentos o puede venir el jefe y te manda a hacer otra tarea, etc

Escalas utilizadas en Poker estimation:

- Tamaño por números: De 1 a 10
- Talles 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

Story Points:

Es una unidad de medida específica (del equipo), de complejidad, riesgo y esfuerzo, es lo que es *el kilo* a la unidad de nuestro sistema de medición de peso

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

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

Entonces para calcular la agarramos las user stories que el product owner me ha aceptado durante el sprint y sumó sus story points

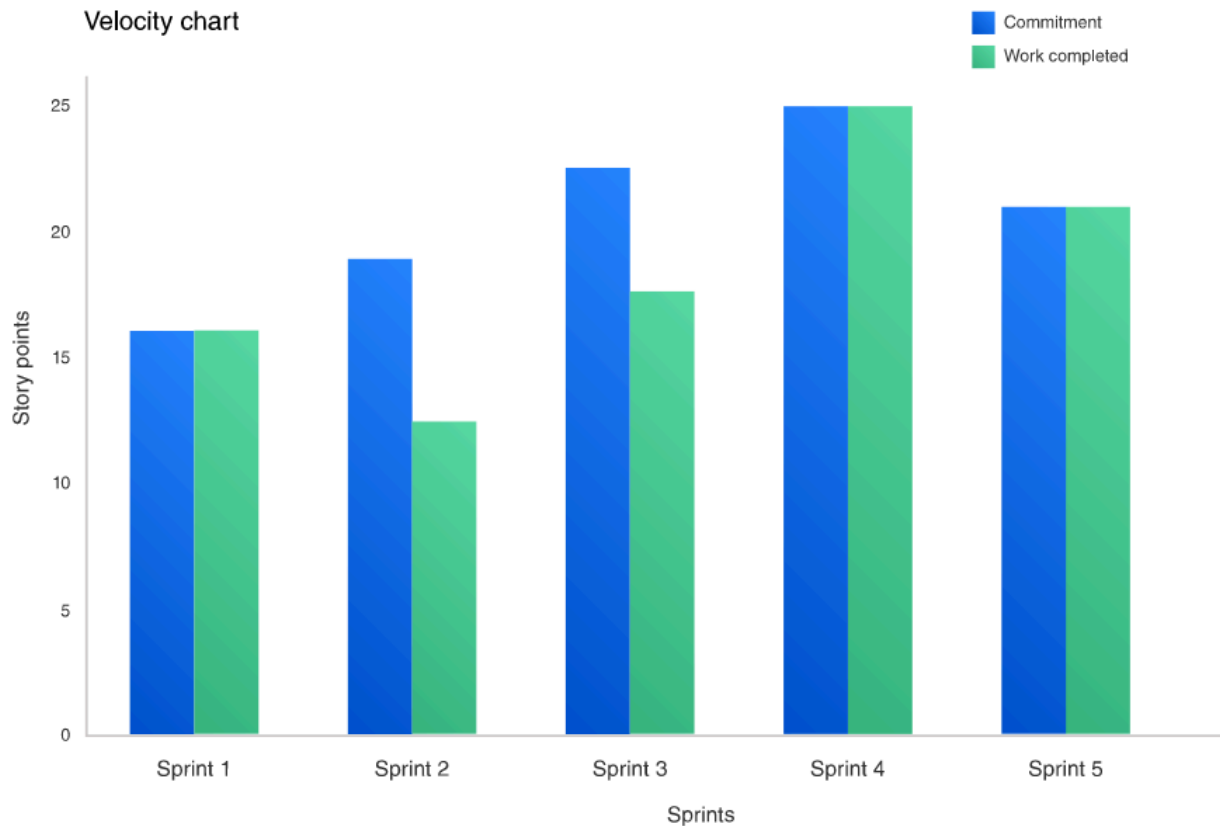
Se cuentan los story points de las user stories que están completas, no parcialmente completas

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:

Combina opinión de experto, analogía y segregación

Participantes en el *planning poker* son desarrolladores

- *Las personas mas competentes en resolver una tarea deben ser quienes las estiman*

Prerrequisitos:

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

Pasos:

1. Determinar la **base story** (la canónica) que será utilizada para comparar con las otras historias. Dlgamos, Story Z
 - 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. Después 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 está story comparada con la canónica?

Nunca elegir transacciones como canónicas

Suelen ser registros de entidades de negocio muy pequeñas

Goto 1.3

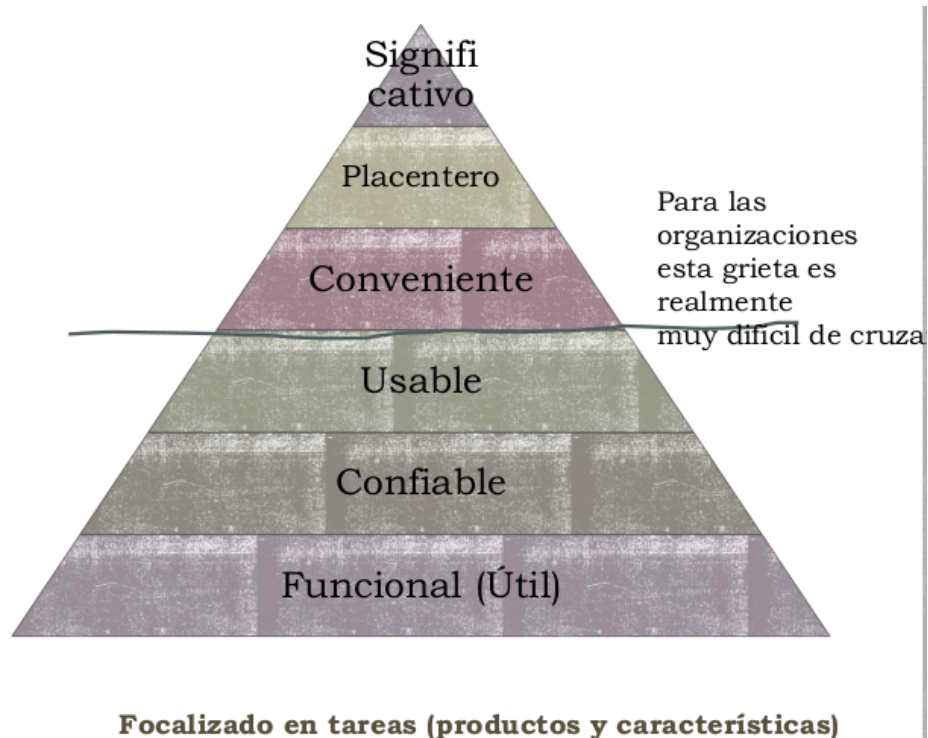
¿Cómo decodificar las estimaciones?

- **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:

Los mínimos:

Evolución de los productos de Software:



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):

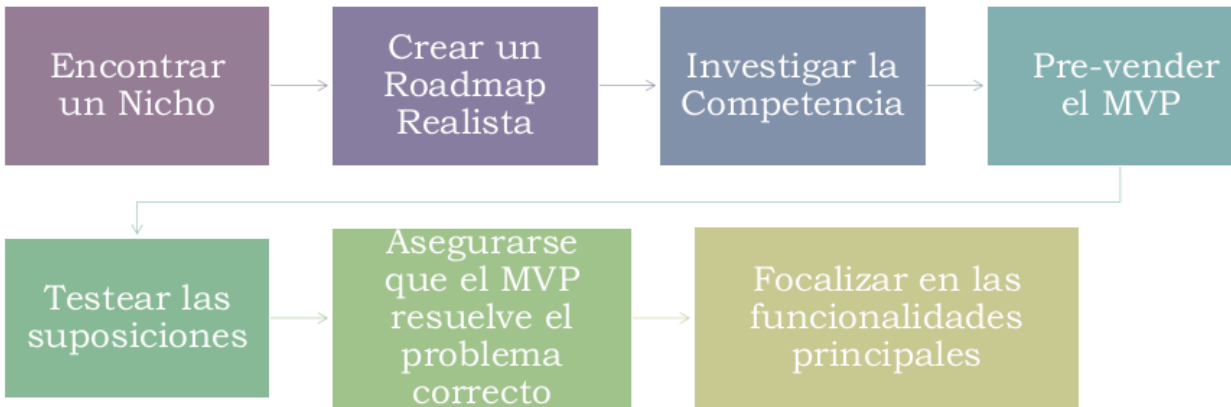
Eric Ries: *Versión de un nuevo producto que permite a un equipo recopilar la cantidad máxima de aprendizaje validado sobre clientes con el menor esfuerzo.*

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

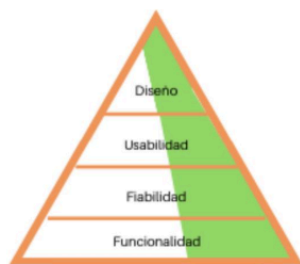
- Dirigido a un **subconjunto** de clientes potenciales
- Utilizado para obtener **aprendizaje validado**
- Más cercano a los **prototipos** que a una versión real funcionando de un producto
- Tiene el **valor suficiente** para que las personas estén dispuestas a usarlo o comprarlo inicialmente
- Demuestra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



¿Cuáles son algunas características extras de un MVP?



Cómo SÍ hacer un MVP



Cómo NO hacer un MVP

Diseño

- Diseño adecuado.
- Consigue una UX que deleita.
- Logra satisfacer el aspecto visual y de interacción.

Usabilidad

- Tiene suficiente valor para que la gente esté dispuesta a usarlo/comprarlo.
- Resulta útil para su público objetivo.

Confiabilidad

- Involucra a los early adopters puedan confiar en la solución plenamente.
- Incluso cuando tenga poco tiempo en el mercado.

Funcionalidad

- Tiene las funciones necesarias para solucionar un problema específico.
- Satisface las demandas y permite evaluar las funciones a implementar más adelante.

¿Cuál es la matriz de priorización para un MVP?

	Urgente	No urgente
Importante	1.Hacer Incluir en el MVP	2. Planear Desarrollar para un lanzamiento beta
No Importante	3.Delegar Considerar integraciones de terceras partes.	4. Eliminar Remover del Roadmap del producto

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 Comerciable (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

Ejemplo: Siguiendo con la aplicación de pedidos de comida, una MMF podría ser la integración de pagos en línea. Esta funcionalidad por sí sola agrega valor significativo al producto, ya que permite a los usuarios completar el proceso de compra de manera más conveniente.

Producto Mínimo Comerciales (MMP):

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.

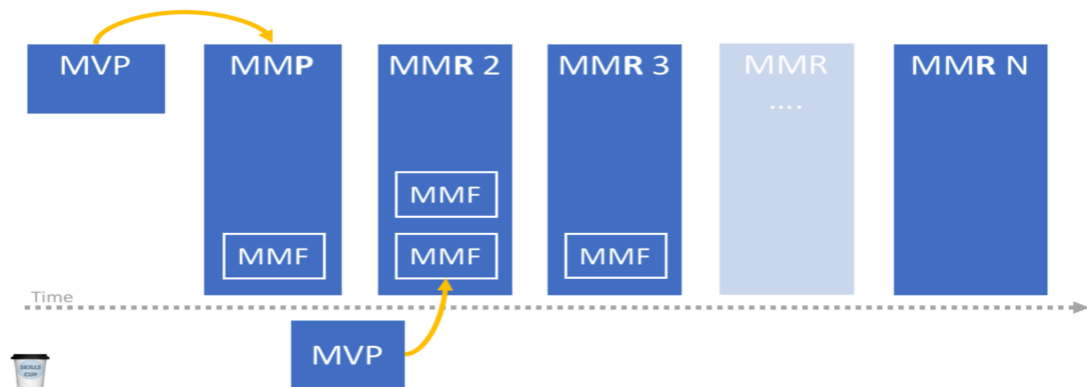
- Primer release de un MMR dirigido a **primeros usuarios - early adopters**
- Focalizado en características clave que satisfagan a ese grupo clave

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**

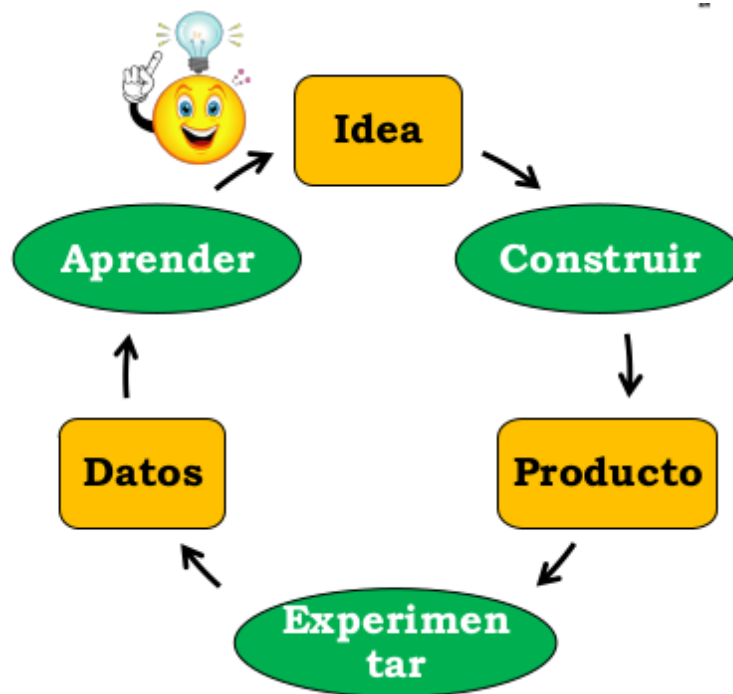
Valor vs Desperdicio:

¿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 averiguar la cosa correcta a construir cada día

Build - Experiment - Learn



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

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

The **Build-Experiment-Learn feedback loop** permite descubrir las necesidades del cliente y alinearlas metodológicamente

Fase de Construcción → MVP:

Ingresar lo más rápido posible con un Producto mínimo viable (MVP).

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

Unidad 3: Gestión del Software como Producto

Software Configuration Management (SCM)

Introduccion

Entendemos a SCM como una **disciplina de soporte** → Es una actividad paraguas, transversal a todo el proyecto, relevante para el producto a lo largo de su ciclo de vida

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.

SCM es una disciplina que aplica dirección y monitoreo administrativo y técnico a:

- Identificar y documentar las características funcionales y técnicas de los ítems de configuración

- Controlar los cambios de esas características, registrar y reportar los cambios y su estado de implementación
- Verificar correspondencia con los requerimientos

(ANSI / IEEE 828,
1990)

¿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**

Involucra para la configuración:

- Identificarla en un momento dado
- Controlar sistemáticamente sus cambios
- Mantener su integridad y origen

La **Integridad del Producto**, es importante porque:

- Satisface las necesidades de los usuarios
- Permite que el software sea completamente rastreado durante su ciclo de vida
- Satisface criterios de performance
- Cumple con sus expectativas de costo

¿Qué problemas surgen en el manejo de componentes?

- Pérdida de un componente
- Pérdida de cambios (el componente que tengo no es el último)
- Sincronía fuente - objeto - ejecutable
- Regresión de fallas

- Doble mantenimiento
 - Superposición de cambios
 - Cambios no validos
-

Es importante entender que los **desarrolladores de software** en realidad son un conjunto complejo de entidades organizacionales que interactúan entre sí.

Cuando nosotros emprendemos un proyecto de software, **estos desarrolladores siempre se van a estructurar dentro de 3 disciplinas básicas:**

- **Administración del proyecto:** Las disciplinas de gestión de proyectos están dirigidas tanto hacia dentro como hacia fuera.

Básicamente le sirven a la gestión general para ver qué es lo que sucede en el proyecto y asegurarse de que la organización desarrolle el producto con integridad

Además, estas disciplinas nos otorgan una visión interna del proyecto, en función de la asignación de los recursos del proyecto

- **Desarrollo:** Representan las disciplinas tradicionalmente aplicadas al proyecto de software
 - Analisis
 - Diseño
 - Ingenieria
 - Produccion
 - TEsting
 - Entre otras

Son básicamente lo que se requiere para llevar un concepto de un sistema hacia un ciclo de vida de desarrollo

- **Aseguramiento del producto:** Estas disciplinas son utilizadas por los administradores del proyecto para ganar visibilidad dentro del desarrollo del proceso. Aquí se incluyen:
 - **Gestión de configuración:** Esto es básicamente una disciplina que se encarga de identificar la configuración de un sistema a momentos discretos en el tiempo con el propósito de controlar sistemáticamente los cambios y mantener una trazabilidad a lo largo del ciclo de vida
 - **Aseguramiento de calidad:**
 - **Validación y verificación:** Acá lidiamos con el problema de que tan bien el software cumple con los requerimientos funcionales y de performance y que los requerimientos hayan sido interpretados correctamente
 - **Testing y evaluación:** Es una disciplina impuesta por fuera del desarrollo del proyecto para que de manera independiente evaluar si el producto cumple o está en línea con los objetivos

El uso adecuado de estas disciplinas de aseguramientos de proyectos por parte del project manager son la base para el éxito del proyecto.

Algunos conceptos de SCM

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

Ítem de Configuración de Software (SCI):

Documentos de diseño, código fuente, código ejecutable, etc

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.

Algunos **ejemplos** de SCI puede ser:

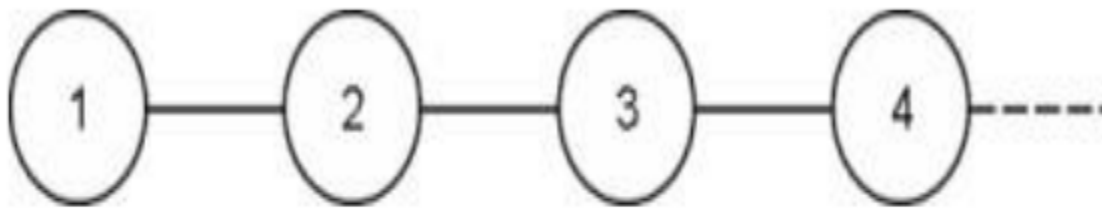
- ❖ Plan de CM
- ❖ Propuestas de Cambio
- ❖ Visión
- ❖ Riesgos
- ❖ Plan de desarrollo
- ❖ Prototipo de Interfaz
- ❖ Guía de Estilo de IHM
- ❖ Manual de Usuario
- ❖ Requerimientos
- ❖ Plan de Calidad
- ❖ Arquitectura del Software
- ❖ Plan de Integración
- ❖ Planes de Iteración
- ❖ Estándares de codificación
- ❖ Casos de prueba
- ❖ Código fuente
- ❖ Gráficos, iconos, ...
- ❖ Instructivo de ensamble
- ❖ Programa de instalación
- ❖ Documento de despliegue
- ❖ Lista de Control de entrega
- ❖ Formulario de aceptación
- ❖ Registro del proyecto

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

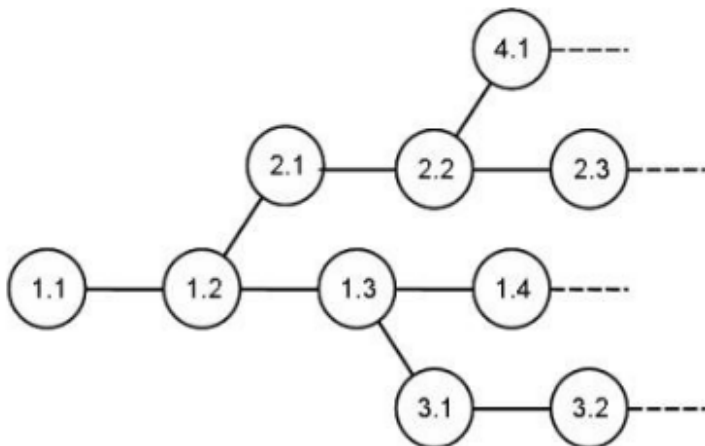
Variante:

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.Ó.) que la soporta, o de las funciones opcionales que haya de realizar o no.



Variante de un ítem de configuración

Repositorio:

- Un repositorio de información conteniendo los ítems de configuración
- Mantiene la historio de cada SCI 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 archivos con sus versiones
- Los administradores tienen mayor control sobre el repositorio
- 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

Linea Base

Se utilizan etiquetas para *marcar* las baselines

No debemos confundir con la versión del producto

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

Sirve como base para desarrollos posteriores y puede cambiarse sólo a través de un procedimiento formal de control de cambios

Permiten ir atrás en el tiempo y reproducir el entorno de desarrollo en un momento dado del proyecto

Las líneas base puede ser:

- De especificación DE (Requerimientos, Diseño)
- De productos que han pasado por un control de calidad definido previamente

Ramas

Creación de ramas:

- Existen una rama principal (trunk, master)
- Sirven para bifurcar el desarrollo

- Pueden tener razones de creación con semántica
- Permiten la experimentación
- Pueden ser descartadas o integradas

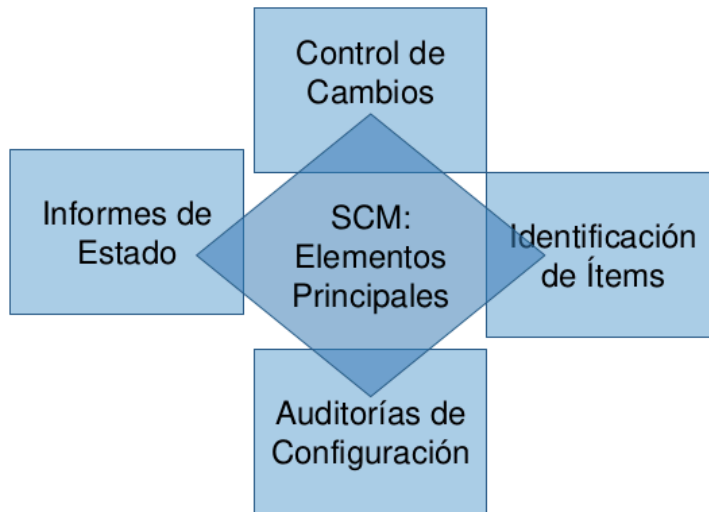
Integración de ramas:

- 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

Configuración de software:

Es un conjunto de ítems de configuración con su correspondiente versión en un momento determinado

Actividades fundamentales / elementos de SCM:



Identificación de ítems de configuración:

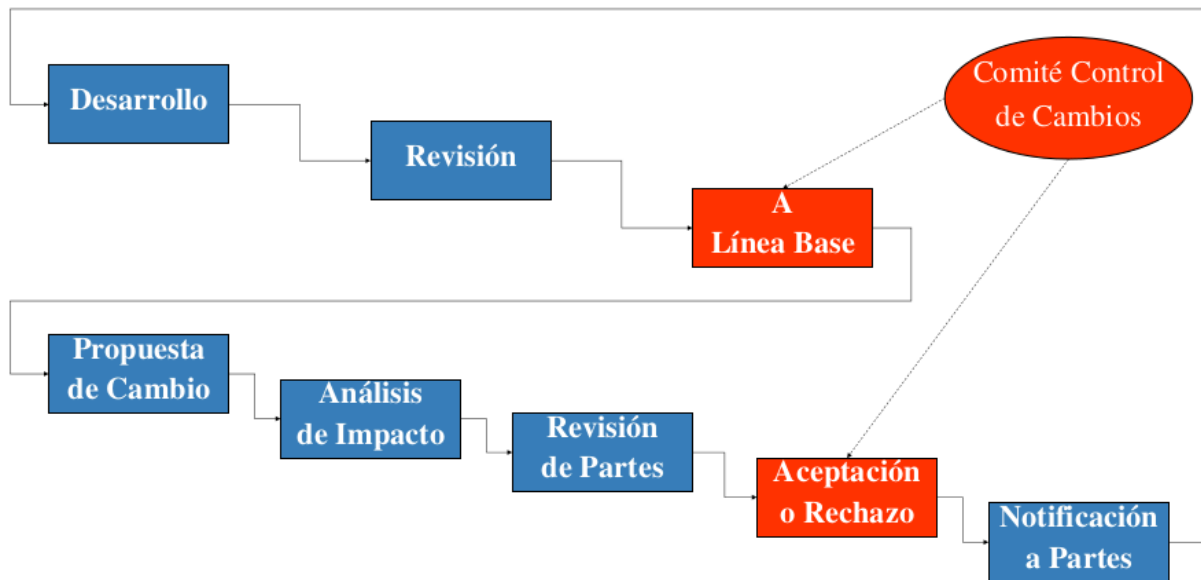
- Identificación unívoca de cada ítem de configuración
- Convenciones y reglas de nombrado
- Definición de la Estructura del Repositorio
- Ubicación dentro de la estructura del repositorio

Los IC puede ser de diferentes tipos:



Control de cambios:

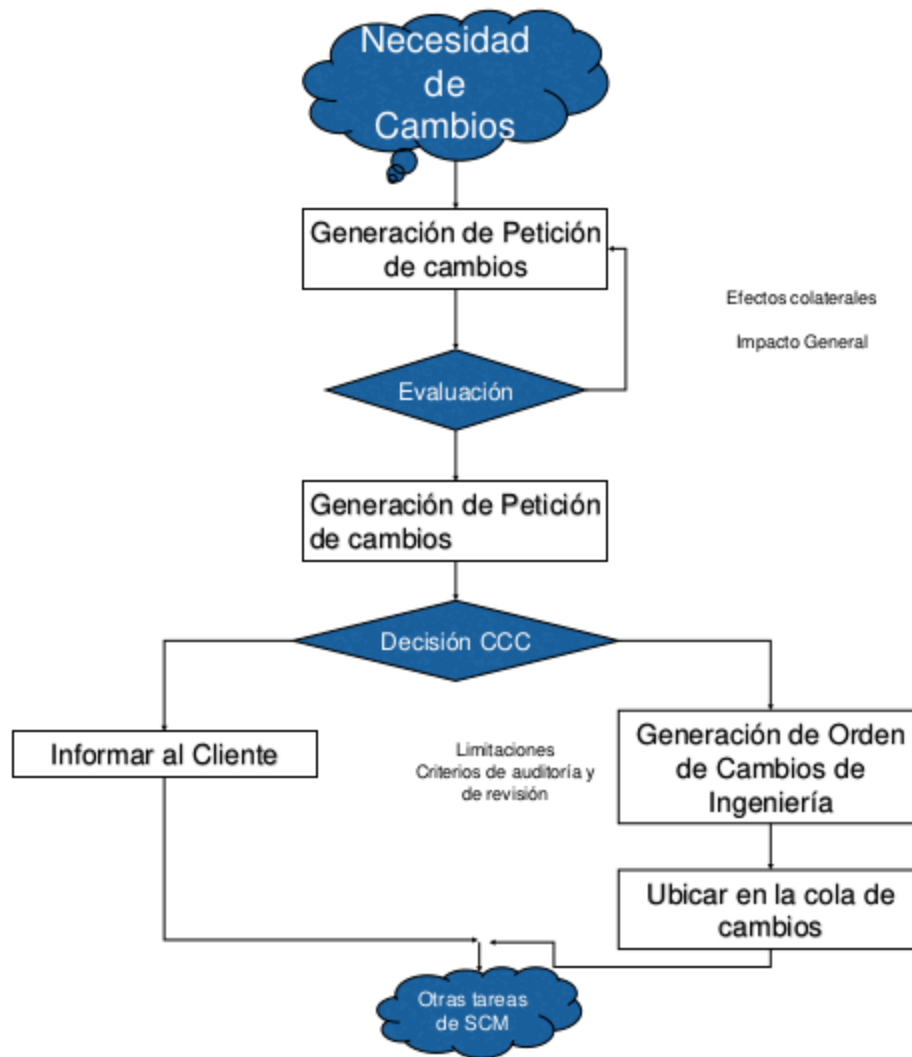
El proceso de control de cambios sería algo como esto:



El control de cambios tiene su origen en un requerimiento de cambio a uno o varios ítems de configuración que se encuentran en una **línea base**

Es un procedimiento formal que involucra diferentes actores y una evaluación del **impacto** del cambio

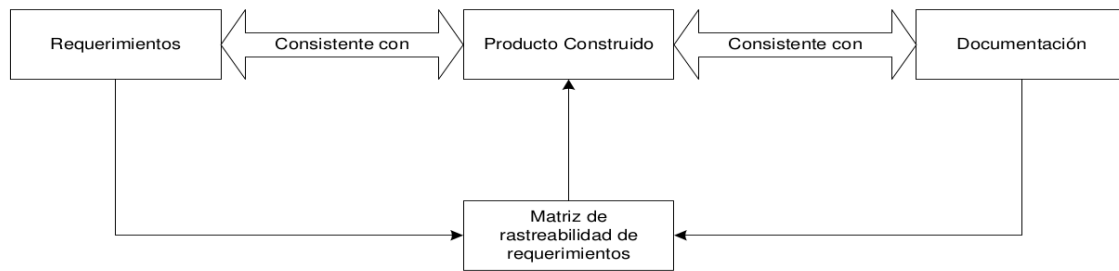
A continuación un diagrama para entender mejor:



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:



Auditorías de Configuración:

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

Consistencia entre lo que definimos en el plan de configuración para cada IC, con lo que tengo en el repositorio

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):** 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. 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

Informes de estado:

El registro e Informe de Estado:

- Se ocupa de mantener los registros de la evolución del sistema
- Maneja mucha información y salidas por lo que se suele implementar dentro de procesos automáticos
- Incluye reportes de rastreabilidad de todos los cambios realizados a las l pinas base durante el ciclo de vida

ALgunas **preguntas** que podr a responder son:

-  Cu al es el estado del  tem?
-  Un requerimiento de cambio ha sido aprobado o rechazado por el CCB?
-  Qu  vers on de  tem implementa un requerimiento de cambio aprobado (saber cu al es el componente que contiene la mejora)?
-  Cu al es la diferencia entre una vers on y otra dada?

Plan de gesti n de Configuraci n:

Lo que deber amos incluir en el plan son:

- Reglas de nombrado de los CI
- Herramientas a utilizar para SCM
- Roles e integrantes del comit 
- Procedimiento formal de cambios
- PLantillas de formularios
- Procesos de auditor a

SCM en ambientes ágiles:

- Sirve a los practicantes (equipo de desarrollo) y no viceversa
- Hace seguimiento y coordina el desarrollo en lugar de controlar a los desarrolladores
- Responde a los cambios en lugar de tratar 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

Algunos tipos para SCM en Agile:

- Es responsabilidad de todo el equipo
- Automatizar lo más posible
- Educar la equipo
- Tareas de SCM embebidas en las demás tareas requeridas para alcanzar el objetivo del Sprint.

Prácticas continuas:

- **Integración continua:** Se refiere a la práctica de combinar frecuentemente y de manera automatizada los cambios de código realizados por diferentes miembros del equipo en un repositorio compartido.

Esto permite detectar y solucionar errores de integración tempranamente, lo que ayuda a garantizar la calidad del código

- **Entrega continua:** es una extensión de la integración continua, que implica la automatización del proceso de construcción, prueba y empaquetado del software en cada cambio de código, y la entrega de estos paquetes a un entorno de pruebas o de producción para su evaluación. La entrega continua permite a los equipos de desarrollo detectar y corregir errores en etapas tempranas del ciclo de vida del software.
- **Despliegue continuo:** es una extensión de la entrega continua, que implica la automatización de todo el proceso de despliegue del software en un entorno de producción. Esto permite que los cambios de código sean entregados a los usuarios finales con mayor rapidez y frecuencia, lo que reduce el tiempo de lanzamiento de nuevas funcionalidades y mejora la experiencia de los usuarios.