Ingeniería de Software.

**Software:**

* Set de programas, y la documentación que lo acompaña. Existen 3 tipos básicos de software: System Software, Utilitarios y Software de Aplicación.
* Cualquier cosa que se genere a partir del procesamiento de información.
* Conocimiento empaquetado en distintos formatos.
* Intangible 🡪 Alto nivel de complejidad.
* Actividad humano-intensiva 🡪 Materia Prima 🡪 Nosotros como personas que pensamos.
* Un cambio en los requerimientos implica un cambio en el software.

**Proceso de Software:**

* Conjunto estructurado de actividades para desarrollar un sistema de software.
* Estas actividades varían dependiendo de la organización y el tipo de sistema que debe desarrollarse.
* Debe ser explícitamente modelado si va a ser administrado.



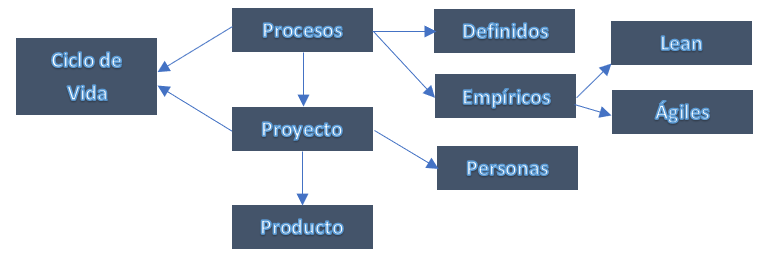
5 razones para **NO** comparar software con manufactura:

1. El software es menos predecible.
2. No hay producción en masa, casi ningún producto de software es igual a otro.
3. No todas las fallas son errores.
4. El software no se gasta.
5. El software no está gobernado por las leyes de la física.

Algunos problemas con el desarrollo de software:

* La versión final del producto no satisface las necesidades del cliente.
* No es fácil extenderlo y/o adaptarlo. Agregar más funcionalidad en otra versión es casi una misión imposible.
* Mala documentación.
* Mala calidad.
* Más tiempos y costos que los presupuestados.

Ingeniería de Software.

1. Disciplinas Técnicas 🡪 Producto (Construcción).
   1. Requerimientos.
   2. Análisis y Diseño.
   3. Construcción.
   4. Prueba.
   5. Despliegue.
2. Disciplinas de Gestión 🡪 Proyectos (se hace software a través de éstos).
   1. Planificación de Proyecto.
   2. Monitoreo y Control de Proyectos.
3. Disciplinas de Soporte 🡪 Complementarias (se ajustan al proyecto y al producto).
   1. Gestión de Configuración de Software.
   2. Aseguramiento de Calidad.
   3. Métricas.

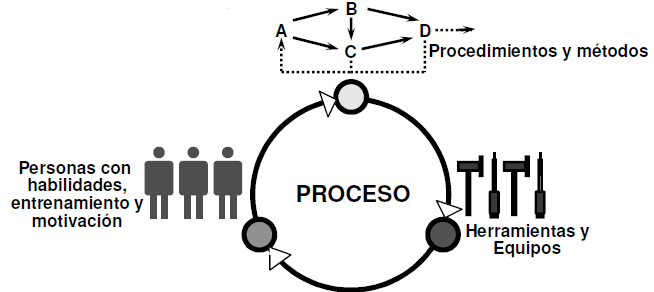
Se adapta

Obtiene como resultado

Se incorpora

**Proceso:** Conjunto de tareas interrelacionadas. Conjunto de pasos ejecutados para un propósito dado (IEEE). El mismo se adapta al proyecto, y se adhiere a un ciclo de vida determinado. Se tiene que definir cuál proceso se va a usar, para ver qué necesito del proyecto, y qué no.

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



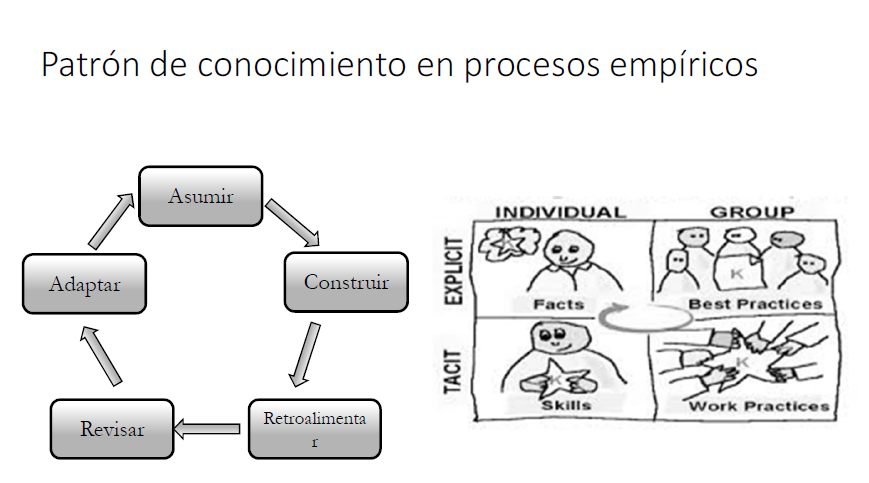
**Tipos de Proyectos:** **Definidos** (ejemplo: Cascada, PUD, RUD, Iconix, Métricas – solo los lineamientos, es decir, hay cosas todavía sin definir -). Inspirados en las líneas de producción:

* 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.
* Estandarizados.
* Siempre se aplica el mismo procedimiento (se definen entradas, salidas, procedimientos, roles, actividades y etapas).
* Ante iguales entradas, se obtienen iguales salidas mediante la aplicación de un proceso definido.



* El proceso implementa un ciclo de vida.
* Calidad del proceso 🡪 Calidad del producto.

**Empíricos** (Inspección y Adaptación):

* Asume procesos complicados con variables cambiantes. Cuando se repite el proceso, se pueden llegar a obtener resultados diferentes.
* La administración y control es a través de inspecciones frecuentes y adaptaciones.
* Son procesos que trabajan bien con procesos creativos y complejos.
* Dependen de la experiencia de sus miembros.
* Tienen un Feedback (permite a un proceso aplicar una mejora, la cual se aplica sobre las personas).

Proceso Ciclo de Vida 🡪 El **Ciclo de Vida** se refiere a cómo encaro la ejecución del proyecto (cuántas tareas se hacen, y en qué momento); serie de pasos a través de los cuales el producto o proyecto progresa. Un ciclo de vida en un proyecto de software es una representación del proceso. Grafica una descripción del proceso desde una perspectiva particular. Los modelos especifican:

* Las fases del proceso (por ejemplo, requerimientos, especificación, diseño, etc.).
* El orden en el cual se llevan a cabo.

Este Ciclo también es llamado **Modelo de Procesos**, y existen 3 tipos:

1. Secuencial (ejemplo: cascada).
2. Iterativo (ejemplo: iterativo-incremental).
3. Recursivo (ejemplo: espiral).

**Proyecto:** Medio que está orientado a objetivos (los cuales son Claros y Alcanzables – factibles; así como también son los que guían al proyecto; y no deben ser ambiguos), y por lo general dicho objetivo es obtener un **Producto** con alcance gradual (ya que implica tareas interrelacionadas basadas en esfuerzo y recursos). Un proyecto tiene una duración limitada en el tiempo (tiene principio y fin), y a su vez cada proyecto es ÚNICO. El desarrollo de un proyecto conlleva la utilización (afectación) de recursos, como se mencionó anteriormente, que, a su vez, cuando el mismo termina, éstos últimos quedan libres (liberados).

Aclaraciones:

* Una línea de producción NO ES UN PROYECTO.
* Todos los proyectos por similares que sean, tienen características que los hacen únicos.

Otra definición: Unidad de gestión u organización que administra recursos para obtener un resultado (producto o servicio de características únicas). Tiene que administrarse SÍ O SÍ (debe tener un plan, y para eso debe tener **Seguimiento** (o monitoreo), el cual es permanente, y **Supervisión** (o control), el cual es en base a hitos determinados (puntos de control – me detengo para ver cómo me está yendo). Estos hitos se asocian a fines de etapas o con entregables.

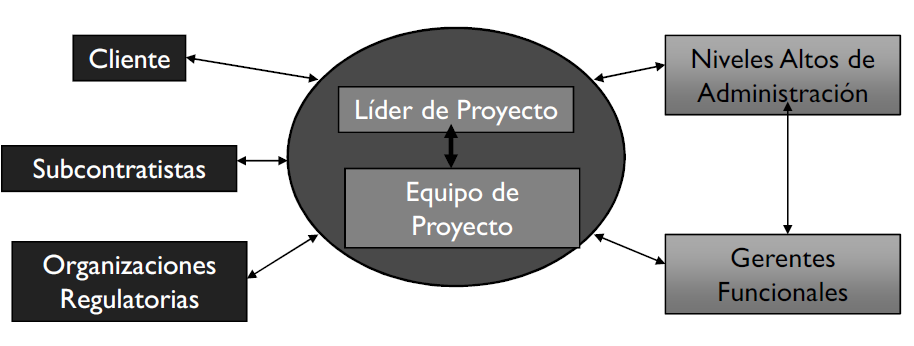
**Administración de proyectos:** “… tener el trabajo hecho …” en tiempo, con el presupuesto acordado y habiendo satisfecho las especificaciones o requerimientos; aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades del proyecto para satisfacer los requerimientos del proyecto. Para administrar un proyecto, se tiene que:

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

Considerando esta administración, se tiene que tener en cuenta el problema de “**La Restricción Triple**”:

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

El balance de estos tres factores afecta directamente 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 del Líder de Proyecto balancear estos tres objetivos (que a menudo compiten entre ellos), y cuyo rol se muestra a continuación:



**Producto Software:** Cada nueva versión es desarrollada **incrementalmente** en una serie de pasos.

**Equipo de Proyecto:** Un grupo de personas comprometidos en alcanzar un conjunto de objetivos de los cuales se sienten mutuamente responsables. Algunas características son:

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

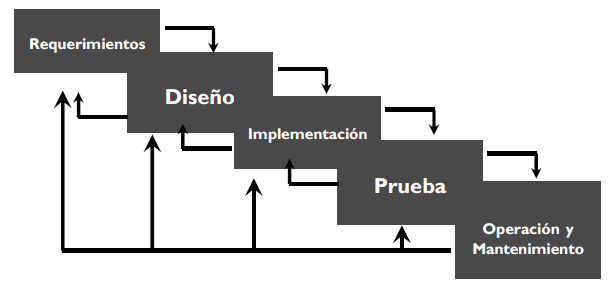
Gestión Tradicional de Proyectos.

Para planificar un proyecto de software se requiere de un **Plan de Proyecto** (Plan de Desarrollo de Software – Un plan es a un proyecto lo que una hoja de ruta a un viaje), el cual tiene varias partes:

* **Definición de un objetivo del proyecto:** Lo que va a hacer.
* **Alcance del proyecto:** Relevamiento de requerimientos, análisis, diseño, etc. Hace referencia al trabajo para cumplir con el objetivo (tareas). En caso del PRODUCTO, el alcance serían todas las características que pueden incluirse en un producto o servicio. (por ejemplo, objetivos de los casos de uso de diseño). En Caso del PROYECTO, el alcance es todo trabajo y solo trabajo que debe hacerse para entregar el producto o servicio con todas las características y funciones especificadas (por ejemplo, la ERS).

El cumplimiento del alcance del **Proyecto** se mide contra el **Plan de Proyecto**, mientras que el cumplimiento del alcance del **Producto** se mide contra la **Especificación de Requerimientos**.

* **Definir qué proceso se va a usar.**
* **Definir qué ciclo de vida:** (Del proyecto, no del producto).



* **Dimensionamiento del Proyecto:** O estimaciones.
  + **Tamaño:** Por ejemplo, a nivel de módulos, alcance, requerimientos, CU, clases, líneas de códigos (aunque al día de hoy ya no sirve).
  + **Esfuerzo:** Del tamaño se deriva el esfuerzo. Esto se mide en Horas-Persona Lineales (1 cosa a la vez por persona). De esto obtengo un total de cantidad de horas.
  + **Tiempo:** Del esfuerzo deriva el tiempo. Base para la calendarización.
  + **Costo:** Por ejemplo, si requiero que sea más rápida la entrega, tengo que tener en cuenta que voy a tener que requerir de horas extras por parte de los trabajadores, por ende, eso es más gasto porque a esas horas hay que pagarlas.
* **Riesgos:** No es lo mismo que problema, ya que los tratamientos en cada caso son distintos (a partir de un riesgo, se puede producir un problema que podría comprometer el éxito del proyecto; o dicho de otra forma, es un problema esperando para suceder). Por ende, un riesgo tiene una probabilidad asociada (probabilidad de ocurrencia de un factor que genere algún daño o pérdida). A los riesgos se los mide: .

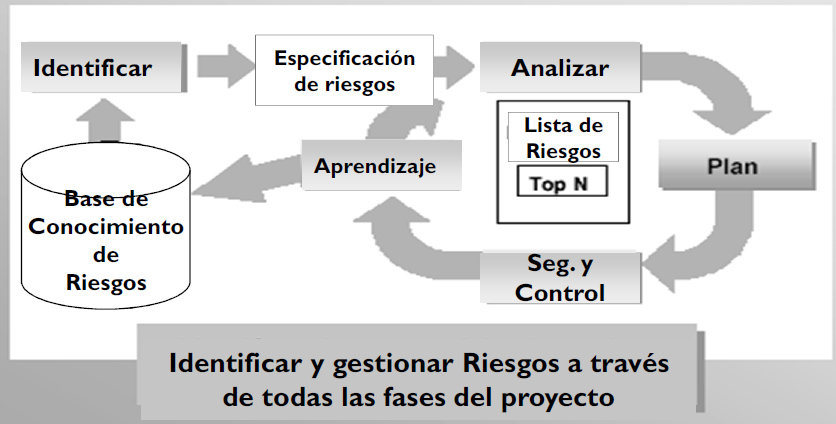
Esta es una fórmula de cálculo (por lo que permite determinar el orden de los riesgos), en donde probabilidad hace referencia a la posibilidad que algo se presente, e impacto al daño que produce dicho acontecimiento posible.

Un ejemplo de riesgo (y en este caso, bastante alto), es el de equivocarse a la hora de la determinación de los requerimientos.

Para los riesgos, hay que tener estrategias **pro**activas (actuar con anticipación para mitigar dichos riesgos), y de ahí un plan de contingencia. En contrapartida, tener medidas **re**activas no es tan eficiente, ya que el suceso ya pasó.

Mitigación 🡪 Plan de Contingencia 🡪 Evento Disparador

Para la gestión de riesgos se tiene:



* **Calendarización o programación del proyecto:** Esto se realiza con las tareas ya definidas.
* **Definición de métricas:** Es un número. Grado de presencia de una característica en el ámbito donde quiero medirla. Se puede dar en 3 dominios (procesos/proyecto/producto).
* **Cómo se hace seguimiento y control del proyecto:** Qué reuniones/reportes hay que hacer; a quién le doy dichos reportes; con qué frecuencia se dan dichas reuniones, etc.

Causas de Fracasos en Proyectos:

* Fallas al definir el problema.
* Planificar basado en datos insuficientes.
* La planificación la hizo el grupo de planificaciones.
* No hay seguimiento del plan de proyecto.
* Plan de proyecto pobre en detalles.
* Planificación de recursos inadecuada.
* Las estimaciones se basaron en “supuestos” sin consultar datos históricos.
* Nadie estaba a cargo.

Metodologías Ágiles.

**Manifiesto Ágil** – Declaración de principios ágiles, los cuatro valores:

1. **Individuos e interacciones** por sobre procesos y herramientas: Se valora los individuos e interacciones **POR SOBRE** (ambos son iguales de importantes, pero se presta más atención en uno) procesos y herramientas. Se les da mucha importancia a los stakeholders.
2. **Software funcionando** por sobre documentación detallada: Se eliminan las “recetas” de un proyecto a la hora de hacer software. Si hay documentación con nivel de detalle en base al criterio de elegir qué documentos y cuáles no. La documentación debe tener valor para el cliente (es decir, en el sentido que el software le satisfaga sus necesidades).
3. **Colaboración** por sobre negociación con el cliente: Antes de tener un contrato prefijado, se valor más una colaboración fluida entre desarrolladores y los clientes.
4. **Responder a cambios** por sobre seguir un plan.

De estos 4 valores se desprenden 12 principios:

1. La prioridad es satisfacer al cliente a través de releases tempranos y frecuentes (2 semanas a un mes): Pero siempre software con valor (que le sirva al cliente, y cumpla con sus expectativas). Con respecto a las entregas, sirven para poder tener un Feedback más rápido, lo que nos evita tener que hacer grandes cambios en el futuro que modifiquen todo el proyecto, y sean difíciles de desarrollar. Como conclusión, con entregas tempranas y frecuentes, se solucionan los problemas mucho más rápido y antes.
2. Recibir cambios de requerimientos, aún en etapas finales: Con análisis de impacto previo, pero siempre con una postura de estar abiertos antes cambios.
3. Releases frecuentes (2 semanas a un mes).
4. Técnicos y no técnicos trabajando juntos TODO el proyecto: Esto para evitar hasta tener una release para obtener un Feedback, por ende, el proceso se hace más rápido.
5. Hacer proyectos con individuo motivados: Personas motivadas 🡪 Éxito en el proyecto (porque se les da importancia a los propios individuos).
6. El medio de comunicación por excelencia es cara a cara: Parte de la comunicación no es verbal (a nivel de gestos, o lo que no se dice al decir tal cosa, por ejemplo).
7. La mejor métrica de progreso es la cantidad de software funcionando: Metodologías ágiles reducen las métricas a usar 🡪 Se reduce a la propia cantidad de software que funciona.
8. El ritmo de desarrollo es sostenible en el tiempo.
9. Atención continua a la excelencia técnica: Que cada miembro del equipo tenga excelencia técnica (capacidad para producir software – que tenga calidad).
10. Simplicidad – Maximización del trabajo no hecho: Evitar el “ya que estamos, agrego esto…”, ya que esto termina en no agregar nada de valor. Se deben plantear soluciones de la forma más simple posible, pero que entregue valor.
11. Las mejores arquitecturas, diseños y requerimientos emergen de equipos auto organizados: Miembros con expertise suficiente para construir productos de calidad.
12. A intervalos regulares, el equipo evalúa su desempeño y ajusta la manera de trabajar.

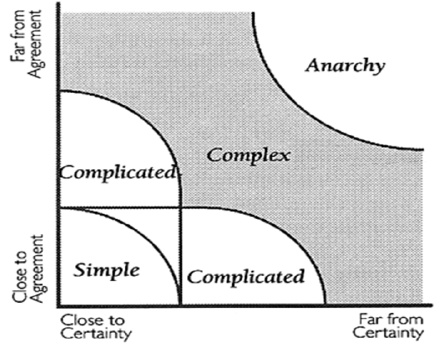
¿Qué significa Ágil?

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.

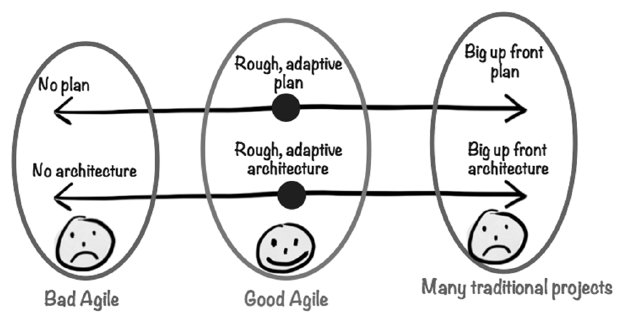
Las metodologías ágiles le dan más foco al producto, **priorizando** al cliente.

Por otro, lado. ¿Cuándo se aplica Agile?

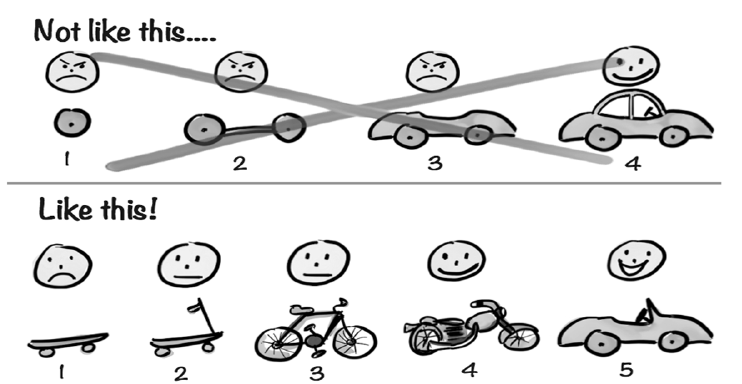


* Agile da mejores resultados cuando los problemas a ser resueltos caen dentro del espacio “Complex”.
* El desarrollo de nuevos productos y Knowledge Work tienden a estar en el espacio Complex.
* Investigación está dentro de Anarchy.
* Mantenimiento por lo general cae en Simple.

Hay que considerar que ser ágil no es ser indisciplinado:



Pero tampoco es que hay que hacer todo por partes pequeñas:



Filosofía Lean.

Es una filosofía que se basa en flujos, y ya no en iteraciones. Con flujo se refiere a conjunto de paradas, o de unidades, que no son más que distintas funcionalidades que van a ir fluyendo a medida que se crean. Estas unidades pueden tener una prioridad, y se ubican en una cola de unidades de trabajo. Cuando termino una de estas unidades, libero recursos, y de esta forma paso al siguiente (Just in Time).

Se describe lo mínimo necesario para arrancar con el flujo (voy agregando requerimientos en el proceso). Consta de una serie de 7 principios:

1. Eliminar el desperdicio: Evitar que las cosas se pongan viejas antes de terminarlas, o evitar re-trabajo. Tiene que ver con el principio ágil de software funcionando y el de simplicidad (arte de maximizar lo que no hacemos). Algunos desperdicios, dependiendo el caso y la medida en que se toma, pueden ser prototipos, estimaciones, documentación, etc. Otra cosa que se considera desperdicio, es lo mencionado anteriormente de “ya que estamos, agrego esto…”.

Se tiene como objetivo 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: el inventario.
  + En software: es el trabajo parcialmente hecho, y las características extra.
  + El 20% del software que entregamos contiene el 80% del valor (regla de Pareto).

Desperdicios en manufactura:

* + - Producción en exceso.
    - Stock.
    - Pasos extra en el proceso.
    - Búsqueda de información.
    - Defectos.
    - Esperas.
    - Transportes.

Desperdicios en software:

* + - Características extras.
    - Trabajos a medias.
    - Proceso extra.
    - Movimiento.
    - Defectos.
    - Esperas.
    - Task Switching.

1. 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.
   * Se debe generar nuevo conocimiento y codificarlo de manera tal que sea accesible a toda la organización.
   * Muchas veces los procesos “estándares” hacen difícil introducir en ellos mejoras.
2. Embeber la integridad conceptual: Encastrar todas las partes del producto o servicio, que tenga coherencia y consistencia (tiene que ver con los requerimientos no funcionales). La integración entre las personas hace el producto más íntegro. Se necesita más disciplina, no menos.
   * **Integridad Percibida:** El producto total tiene un balance entre función, uso, confiabilidad y economía que le gusta a la gente.
   * **Integridad Conceptual:** Todos los componentes del sistema trabajan en forma coherente en conjunto.
   * El objetivo es construir con calidad desde el principio, no probar después.
   * Dos clases de inspecciones:
     + Inspecciones luego de que los defectos ocurren.
     + Inspecciones para prevenir defectos.
   * Si se quiere calidad, no inspeccione después de los hechos.
   * Si no es posible, inspecciones luego de pasos pequeños.
3. Diferir compromisos: El último momento responsable para tomar decisiones (en el cual todavía estamos a tiempo). Si nos anticipamos, tenemos información parcial.
   * Se relaciona con el principio ágil: decidir lo más tarde posible pero responsablemente. No hacer trabajo que no va a ser utilizado. Enlaza con el principio anterior de aprendizaje continuo, mientras más tarde decidimos más conocimientos (máxima información disponible para tomar una decisión).

Las decisiones deben tomarse en el último momento que sea posible.

* + No significa que todas las decisiones deben diferirse.
  + Se debe tratar de tomar decisiones reversibles, de forma tal que pueda ser fácilmente modificable.
  + Vencer la “parálisis de análisis” para obtener algo concreto terminado.
  + Las mejores estrategias de diseño de software están basadas en dejar abiertas opciones de forma tal que las decisiones irreversibles se tomen lo más tarde posible.

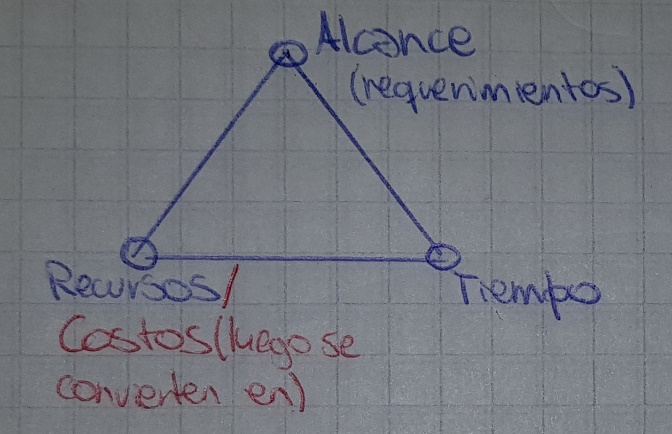
1. Dar poder al equipo: Ejemplo, vamos a comer a un restaurante, y no nos metemos en la cocina del restaurante. Nos fijamos en el precio, pedimos y esperamos. Hay mucho micro management, el dueño no decide cuánta sal poner a la comida.
   * Respetar a la gente.
     + Entrenar líderes.
     + Fomentar buena ética laboral.
     + Delegar decisiones y responsabilidades del producto en desarrollo al nivel más bajo posible.
   * Ágil: El propio equipo pueda estimar el trabajo.
2. Ver el todo – Optimizar el total: Tener una visión holística, de conjunto (el producto, el valor agregado que hay detrás, el servicio que tiene los productos como complemento).
3. Entregar lo antes posible:
   * Entregar rápido: Estabilizar ambientes de trabajo a su capacidad más eficiente y acotar los ciclos de desarrollo.
   * Entregar rápidamente: Esto hace que se vayan transformando “n” veces en cada iteración. Incrementos pequeños de valor. Llegar al producto mínimo que sea valioso. Salir pronto del mercado – relacionado con el principio ágil de entrega frecuente.

User Stories.

La parte más difícil de construir un sistema de software es decidir precisamente qué construir. Ninguna otra parte del trabajo conceptual es tan difícil como establecer los requerimientos técnicos detallados. Ninguna otra parte del trabajo afecta tanto el sistema resultante si se hace incorrectamente. Ninguna otra parte es tan difícil de rectificar más adelante.

**User Stories**: Técnica (manifestación concreta de un enfoque filosófico) para trabajar requerimientos en desarrollos ágiles.

Por esto se plantean 3 principios ágiles que están relacionados entre sí:



El **enfoque** plantea que de acuerdo al alcance (**FIJO**, y definido por el cliente, ya que éste es el que sabe lo que quiere y decide), se define lo demás. Una vez definido el alcance, se determina el tiempo y recursos (**VARIABLES**) para poder llevar adelante el proyecto. Puede ocurrir que se incorporen cambios en los requerimientos, lo cual es algo muy costoso, ya que crece el alcance, y por ende, las variables tiempo y recursos vuelven a cambiarse y tienen que adaptarse. Por eso es que se presenta cierta resistencia a los cambios, aunque el cliente no entienda el trabajo que hay por detrás ante un cambio en el software que nos pide.

Entonces, Ágil lo cambia todo, ya que los requerimientos CAMBIAN (**VARIABLES**), mientras que el tiempo y los recursos ya no (**FIJOS**). El objetivo de Ágil es el requerimiento “Just in Time” (que se relaciona con los principios Lean 1 y 4, así como con el principio Agile 11). Esto me permite empezar con una visión del producto, y con algunos requerimientos identificados, lo que es suficiente para arrancar a trabajar con el ciclo ágil ya que nos da un Feedback que permite las correcciones para la siguiente vuelta.



Ágil plantea un mecanismo de compensación, en donde ya no se ocupe tanto tiempo en la especificación de requerimientos, pero que el cliente esté “on-side”, es decir, cerca nuestro para que el trabajo sea colaborativo, y de esta forma los cambios se hagan de acuerdo a sus preferencias, que él tome las decisiones con nosotros para que luego, al presentarle una versión del software, éste se esté satisfecho. En contrapartida, si la comunicación no es face-to-face, el cliente, al no entender el trabajo de desarrollo, nos pide algo, nosotros como profesionales lo entendemos de una forma, lo desarrollamos, lo codificamos, probamos y todo el trabajo que conlleva, pero a la hora de presentárselo al cliente, éste nos dice que no es lo que él quería (no se hace cargo de sus decisiones).

Como conclusión a esta idea, lo que se quiere es que haya lo que se denomina “clientes disponibles”, para que de esta forma se baje la carga de documentación formal, pero siempre dejando en claro que en ÁGIL SI SE DOCUMENTA.

Partes de una User Story.

Técnica desarrollada por Mike Cohn. Las partes son 3: Tarjeta (Card), Conversation y Confirmation. Es una técnica de requerimientos a nivel de usuario/negocio (NO DE SOFTARE).

Una User Story en la tarjeta describe una funcionalidad esperada, y un recordatorio de que el equipo de desarrollo tiene que hablar con el **Product Owner**, el cual es alguien que sabe qué quiere, y que tiene el poder de decisión necesario para poder priorizar los requerimientos.

Siguiendo con el desarrollo de las partes de una User Story:

* **La parte visible – Tarjeta (Card):** Tiene frente y dorso. En el frente no se determina una especificación de requerimiento, ya que no hay detalle, y esto se compensa con la otra parte (la conversación). Al dorso tiene expresiones de las pruebas de aceptación para garantizar que la User Story se implementó bien. El que escribe estas tarjetas es el Product Owner.
* **Conversation (parte más importante):** Es el cara a cara. La técnica dice que esta conversación no queda asentada en ningún lado, aunque por decisión y con acuerdo previo entre todas las partes, ésta puede ser documentada, por ejemplo, grabada. Se entiende que “no hace falta” por el concepto de “Just in Time” descripto anteriormente. Esta parte tiene la ventaja de evitar la situación que también se describió antes, en la que el cliente nos pide algo, nosotros lo desarrollamos, y cuando se lo presentamos, nos dice que no fue lo que él nos pidió porque no se hace cargo de sus decisiones, ya que la conversación no quedó asentada en ningún lado para poder corroborarlo. Una parte de la conversación se puede guardar en las tarjetas (en el frente), y en el dorso las pruebas de aceptación.
* **Confirmation:** Definición de un acuerdo para hacer, para de esta forma poder mostrar algo que se creó 🡪 Pruebas de aceptación, y si el cliente las acepta, se sigue adelante, pero siempre considerando que lo que se le muestra al cliente, tiene que tener valor de negocio, si no, al Product Owner no le sirve.

Forma de expresar las User Stories.

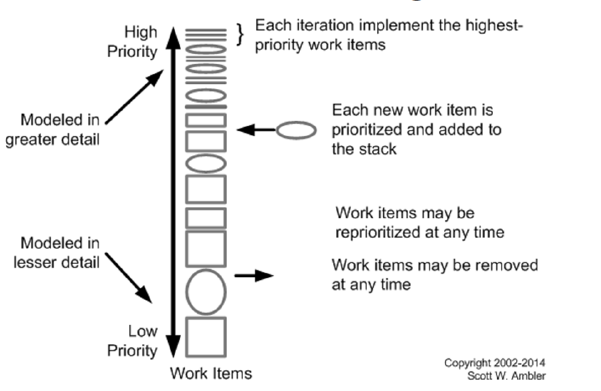
Como **<nombre del rol>** yo puedo **<actividad>** de forma tal que **<valor de negocio que recibo>**.

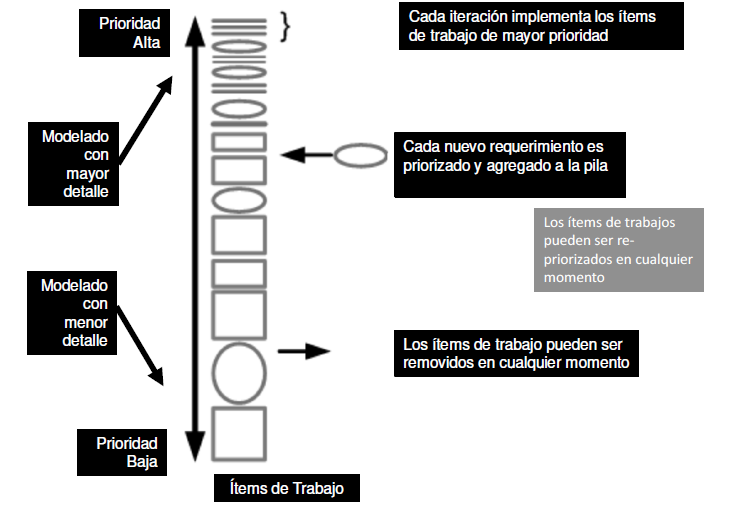
Donde **nombre de rol** representa quién está realizando la acción o quién recibe el valor de la actividad; **actividad** representa la acción que realizará el sistema; y **valor de negocio que recibo** comunica porqué es necesaria la actividad.

Las User Stories son **Multipropósito**: Las historias son:

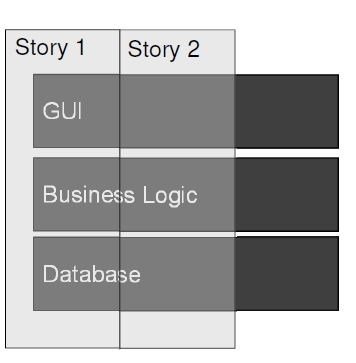
* Una necesidad del usuario.
* Una descripción del producto.
* Un ítem de planificación.
* Token para una conversación.
* Mecanismo para diferir una conversación.

**Product Bachelor:** El Product Owner prioriza las historias en este bachelor. Cada una tiene una granularidad (inversamente proporcional al detalle). Si la historia ya cuenta con una granularidad fina, entonces se puede arrancar a desarrollar.





A las User Stories se las debe considerar como porciones. Esto se demuestra con la siguiente imagen:



Es decir, no tiene sentido que primer se desarrolle horizontalmente TODA la Database, o el Business Logic, ya que, si hacemos esto, no tenemos nada para presentarle al cliente con **valor de negocio**, a pesar que a nivel de programación esté bueno el hecho de contar con algo tan importante ya desarrollado. Por esto, se tiene que trabajar a las User Stories como porciones **verticales**.

Para todo esto, se tiene que tener en cuenta quién va a usar el producto 🡪 Se pueden describir roles de usuario (es decir, una clase con distintas instancias de esta clase – objetos, lo cual tiene una abstracción muy alta). En contrapartida, se puede usar la técnica denominada “**Personas**” (es un objeto). A partir de una Persona se obtiene una experiencia del usuario al que representa a esta Persona, y al hacer análisis de varias, se puede desarrollar una UX agradable que satisfaga los requerimientos al nivel más global posible para todos los usuarios.

**Técnicas Adicionales:**

* Personajes Extremos: Personas en términos extremos que podrían usar el producto, para tenerlos en cuenta a nivel general en cuanto a características. Por ejemplo, considerar el que el producto va a ser utilizado desde el Papa, hasta un narcotraficante.
* Usuarios Representantes (Proxies): Cuando el Product Owner no tiene tiempo, por lo que designa este cargo a otra persona de la empresa (por lo general son personajes del lado del negocio – que no sirven – porque no tienen mucha idea desde el punto de vista de sistemas, o porque pueden verse sesgados por priorizar el área de la empresa donde ellos trabajan y, por ende, el proyecto fracasará.

Criterios de Aceptación de las User Stories.

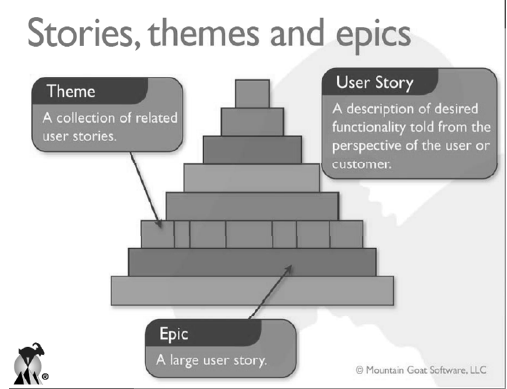
Acuerdo respecto a cómo se tiene que comportar el software. Se desarrollan estos criterios para que el Product Owner lo acepte (definido objetivamente). Para comprobar estos criterios es que se hacen las pruebas de aceptación (que contienen el conocimiento del negocio). Aunque algo a tener en cuenta, es que la mayoría de las veces se desea achicar el número de pruebas, y muchas veces es por motivos económicos para realizarlas. Un ejemplo de criterio puede ser: La nota debe ser un número entero entre 1 y 10 para un examen. Estos criterios se pueden agregar como pie en las tarjetas, y pueden representar tanto requerimientos funcionales, como no funcionales.

**Definition of Ready:** La User Story está apta, por lo que del Product Bachelor pasa a iteración de desarrollo.

**Definition of Done:** Si la User está terminada (está en condiciones de mostrarse al Product Owner – con testing, documentación incluida, y todo lo demás). Esta definición de Done es un checklist, y es una construcción colectiva, es decir, de todo el equipo.

Aspectos adicionales de las User Stories:

* No son especificaciones detalladas de requerimientos (como los casos de uso).
* Son expresiones de intención, “e 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.
* Necesita poco o nulo mantenimiento, y puede descartarse después de la implementación.
* Junto con el código, sirven de entrada a la documentación que se desarrolla incrementalmente después.
* Tiene diferentes niveles de abstracción:



Estos niveles de abstracción dependen de la granularidad de las User Stories (de acuerdo al tamaño, para ver si entra a una iteración o no, para ver si está terminada o no – 0 o 1 (gestión binaria -).

Si una User Story cumple con la Definition of Ready, se desarrolla, y a su vez, si es muy grande para una iteración sola, se la cataloga como **Epic** (aunque siempre considerando que las estimaciones son relativas en cuanto a tamaño/tiempo).

En caso de ser una colección de User Stories relacionadas, se habla de **Theme**.

INVEST Model.

El INVEST Model se utiliza para poder determinar el estado del Definition of Ready. Si la Story cumple con este modelo, entonces se agrega a la iteración para la producción.

* **Independent:** Calendarizables e implementables en cualquier orden. Independiente desde el punto de vista que cada User Story elegida por el Product Owner no tenga dependencia con otra.
* **Negotiable:** El “qué”, no el “cómo”, es decir, el cliente expresa lo que necesita, pero no cómo se implementa, esto es trabajo nuestro, aunque obviamente debemos ponernos de acuerdo previamente con el usuario.
* **Valuable:** Que aporte valor de negocio, debe tener valor para el cliente.
* **Estimatable:** Para ayudar al cliente a armar un ranking basado en costos. Cantidad de información e la User Story que tengo para imaginar el tamaño (para estimarlo).

SPIKE: Tanta incertidumbre que la tengo que investigar y transformar en una User Story. Es una técnica funcional (cómo lo implemento – información del negocio que no tengo, o que es incompleta).

* **Small:** Si es muy pequeña, no da valor al cliente, además si la agrego a la iteración, tengo que esperar a la próxima para que realmente se vea el valor de esta Story, y esta demora no cumple con el principio ágil de releases tempranos y frecuentes.
* **Testable:** Si puedo demostrar que cumplí con expectativas del cliente (con las pruebas de aceptación ya escritas). Demostrar que fueron implementadas.

**Spike:** Tipo especial de historia, utilizado para quitar riesgo e incertidumbre de una User Story u otra faceta del proyecto. Se clasifican en:

1. Técnicas: Utilizadas para investigar enfoques técnicos en el dominio de la solución.
   * Evaluar performance potencial.
   * Decisión hacer o comprar.
   * Evaluar la implementación de cierta tecnología.

Cualquier situación en la que el equipo necesite una comprensión más fiable antes de comprometerse a una nueva funcionalidad en un tiempo fijo.

1. Funcionales: Utilizadas cuando hay cierta incertidumbre respecto de cómo el usuario interactuará con el sistema.

Usualmente son mejor evaluadas con prototipos para obtener realimentación de los usuarios o involucrados.

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 cómo el sistema debe resolver la interacción con el usuario para alcanzar el beneficio esperado.

Algunas User Stories requieren de ambos tipos de spikes (técnicas y funcionales). 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:

1. **Spike Técnico:** 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.
2. **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 spikes:

* Estimables, demostrables, y aceptables.
* La excepción, no la regla:
  + Toda historia tiene incertidumbre y riesgos.
  + El objetivo del equipo es aprender y resolver cierta incertidumbre en cada iteración.
  + Los spikes deben dejarse para incógnitas más críticas y grandes.
  + Utilizar spikes como última opción.
* 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.

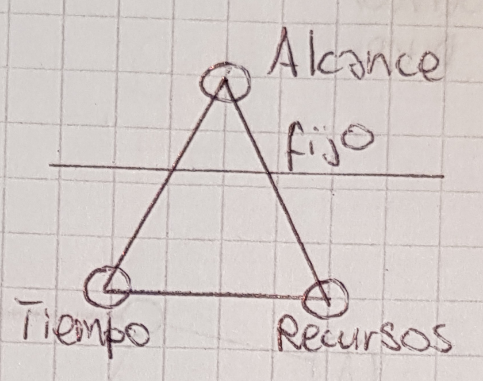
Algunas cosas para dejar en claro.

* Diferir el análisis detallado tan tarde como sea posible, lo que es justo antes de que el trabajo comience.
* Hasta entonces, se captura requerimientos en la forma de “User Story”, que son descripciones breves de funcionalidad relevante para el cliente.
* Las User Stories no son requerimientos; son marcadores para conversaciones más detalladas y análisis que deberán ocurrir conforma esas historias vayan implementándose.
* Por lo tanto, no necesitan ser descripciones exhaustivas de la funcionalidad del sistema, solo la suficiente información para que los desarrolladores y los clientes tengan una comprensión común.

Estimaciones Ágiles.

Primeramente, recordando el tema de estimación, pero a nivel tradicional, y que este tipo de estimación hace referencia a estimaciones **absolutas**, se dividían en 4 puntos, con el siguiente orden:

1. **Tamaño (¿Qué?)**.
2. **Esfuerzo** 🡪 Medido en horas/persona lineales **(¿Cómo?)**. Esto es algo muy personal (individual).
3. **Tiempo (¿Cuándo?)**. Ya en esta instancia se trabaja en equipo, y se definen cosas tales como por ejemplo cuántas personas van a trabajar, qué tareas se pueden desarrollar en paralelo, el propio tiempo para el desarrollo, etc.
4. **Costo (¿Cuánto?)**.

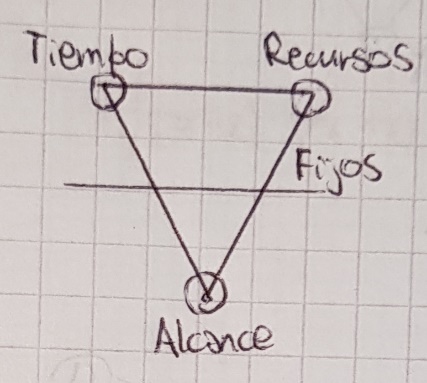


Pasando ahora a Agile:

* En los equipos Agile, las Features/Stories son estimadas usando una medida de tamaño relativo, conocida como **Story Points (SP)**, que hace referencia a qué tan compleja es la característica para poder construirla, y también la incertidumbre que tiene la misma (la falta de información asociada).

Otra definición de Story Point: Unidad de medida específica (del equipo) de complejidad, riesgo y esfuerzo, es lo que “el kilo” a la unidad de nuestro sistema de medición de peso. Da una idea del “peso de cada Story, y decide cuán grande (compleja) es. La complejidad de una Feature/Story tiende a incrementarse exponencialmente.

* Las medidas son **relativas**, no son absolutas, esto quiere decir que se hacen comparaciones (comparo complejidades de las distintas User Stories, las cuales determinan las propias características del producto).
  + Las personas no saben estimar en términos absolutos.
  + Somos buenos comparando cosas.
  + Compara es, generalmente, más rápido.
  + Se obtiene una mejor dinámica grupal y pensamiento de equipo más que individual.
  + Se emplea mejor el tiempo de análisis de las Stories.
* Story Points no es una medida basada en tiempo.



En Agile se hace referencia a cuánto me puedo comprometer (ya que el alcance es lo variable, mientras que el tiempo y los recursos están fijos).

De esta forma, quedan definidas tres variables/dimensiones para determinar el tamaño de una User Story a partir de los Story Points: **Complejidad, Duda y Esfuerzo**.

Se trata que las estimaciones sean un tema tratado por el propio equipo que se va a encargar de desarrollar la misma, ya que, por ejemplo, como se suele hacer en algunos casos, si se deja esta tarea a un “experto”, es difícil que lo que éste “experto” diga, sea lo que en realidad termina sucediendo a la hora de desarrollar la característica del producto, ya que las variables son demasiadas, y una sola persona no puede ser experta en tantos ámbitos, por lo que se generaría desviaciones.

Haciendo una acotación sobre las spikes, mencionadas anteriormente, una User Storie se transforma en una Spike cuando la incertidumbre es mucha, y ya en este punto significa que no se puede estimar, por lo que, por el momento, no puede cumplir con el criterio de listo (Definition of Done).

Siguiendo con las definidas “3 dimensiones”, éstas se pueden combinar de distintas maneras, y eso va a depender del tamaño de la User Story en cuestión, así como también de la escala que se utilice, citando, por ejemplo, la escala de los talles de remera (escala previamente acordada por el equipo, para que, luego, a partir de la misma, ya se puedan comparar las demás User Stories y determinar el tamaño de cada una).

Otras escalas para medir el tamaño pueden ser:

* Tamaño por números: 1 a 10.
* Talles de remeras: S, M, L, XL, XXL.
* Serie : 1, 2, 4, 8, 16, 32, 64, etc.
* Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, etc.

Y una vez elegida la escala, ésta **no se cambia**, porque si se hace, se cambia el metro patrón.

Considerando que el tamaño es una medida de la cantidad de trabajo necesaria para producir una Feature/Story, y que el mismo indica:

* Cuán compleja es la Feature/Story.
* Cuánto trabajo es requerido para hacer o completar una Feature/Story.
* 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.**

Velocidad.

La **velocidad** se la define como una medida (métrica) del progreso de un equipo. Se calcula sumando el número de Story Points (asignados a cada User Story) que el equipo **completa** durante la **iteración**.

Por esta razón, al ver cuántas User Stories meto en una iteración, tengo que ver cuánto pude implementar (software funcionando que le entrega valor de negocio al Product Owner). De esto surge que esto es un asunto binario (o el software está completo, o no está completo), ya que, si no está terminado, no cuenta, por lo que vuelve al Product Backlog; en caso que lo esté, se tiene en cuenta para los Story Points de la iteración.

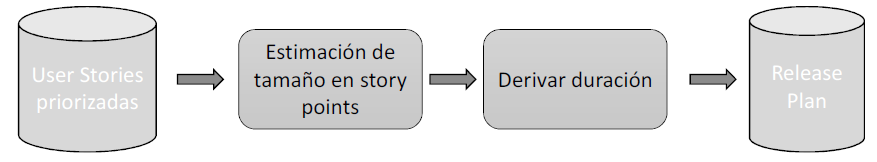
La velocidad corrige los errores de estimación.

Esta medida sirve para estimar de acuerdo a las velocidades de las iteraciones ya hechas. Así, puedo estimar, aproximadamente, cuántas características de producto puedo llegar a implementar en una iteración a futuro, pero SIEMPRE hablando del mismo equipo, en el mismo proyecto, ya que la experiencia no es extrapolable a otros proyectos.

¿Y cómo hago un proyecto?

Si se estima User Stories. ¿Cómo se hace para estimar un proyecto?

* La duración de un proyecto no se “estima”, se deriva tomando el número total de Story Points de las User Stories, y dividiéndolo por la velocidad del equipo.



* La velocidad ayuda a determinar un horizonte de planificación apropiado.
* La estimación en Story Points separa completamente la estimación de esfuerzo de la estimación de duración.

**Estimación de Duración:** Cuántos sprints necesito para un release. Así se estima un proyecto, y lo que mido es el **nivel de certeza**, y **NO el nivel de precisión**.

Poker Estimation.

* Es un método de estimación popular entre los Agile Practicioners, publicado por Mike Cohn.
* Combina opinión de experto, analogía y desegregación.
* Participantes en “Planning Poker” son desarrolladores.
  + “Las personas más competentes en resolver una tarea, deben ser quienes la estiman”.
* Mejor grupal que individual (son muchos factores, y eso hace difícil que una sola persona sea experta en tantos ámbitos).
* “Metro” usado es Fibonacci. Las User Stories de 13 ya son algo complicado para tratar en una sola iteración, por lo que se las trata de partir.
* Se elige una User Story **canónica** (para poder comparar con el resto). Por lo general, se elije una que tenga un peso = 1. Estas comparaciones son relativas, y no se debería cambiar la canónica, porque cambia todo lo ya hecho.
* Time Boxing: Tiempo fijo para todas las actividades (el tiempo es una variable fija en Agile).
* Uno estima por vez, la cantidad de User Story que considera que se van a terminar para el sprint. (por lo general, la menor cantidad posible).

El método consiste en que cada persona, de forma individual y privada, estima cada una de las User Stories, y luego se hace una puesta en común entre todos los miembros del equipo (escuchando justificaciones) para tratar de llegar a una situación en la que estén todos de acuerdo, para poder definir el State of Ready de las mismas.

¿Cómo “decodificar” las estimaciones?

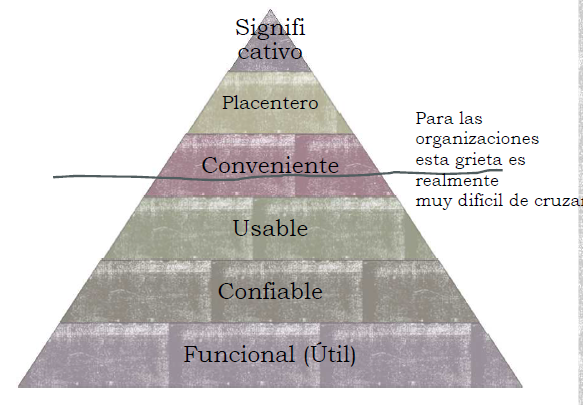
* **0:** Quizás usted no tenga idea de su producto o funcionalidad en este punto.
* **½, 1:** Funcionalidad pequeña (usualmente cosmética).
* **2-3:** Funcionalidad pequeña a mediana. Es lo que se quiere.
* **5:** Funcionalidad media. Es lo que se quiere.
* **8:** Funcionalidad grande, aunque de todas formas se la puede hacer, pero hay que preguntarse si no se puede partir o dividir en algo más pequeño No es lo mejor, pero sigue estand Ok.
* **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:** Confirmación de que algo está muy mal. Mejor ni arrancar.

Gestión de Proyectos.

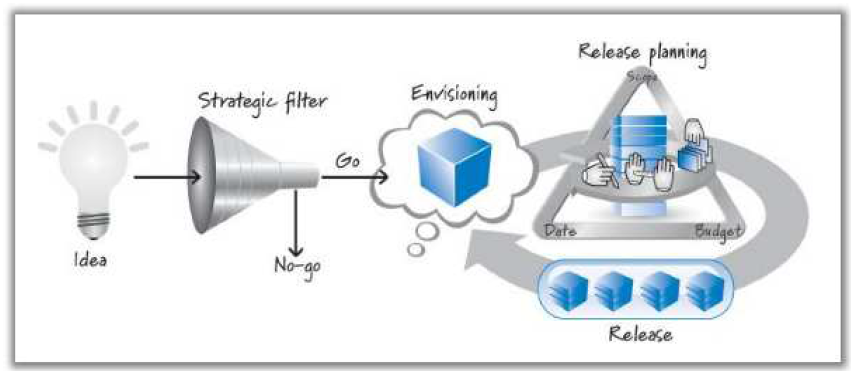
¿Por qué se crean productos?: Hay que entregar valor de negocio al cliente, así como satisfacer las necesidades del mismo, tener muchos usuarios logueados, obtener dinero, realizar una gran visión y cambiar el mundo, entre muchos otros objetivos.

* **Funcionalidad:** Que el producto haga lo que tiene que hacer.
* **Confiabilidad:** El software me da resultados, a partir de los cuales me puedo basar en el futuro.
* **Usabilidad:** Sentirse bien usando el producto. Que me ayude a hacer lo que debería ayudarme a hacer el software.

**Evolución de los productos de software** (focalizado en experiencias – gente, actividades, contexto).

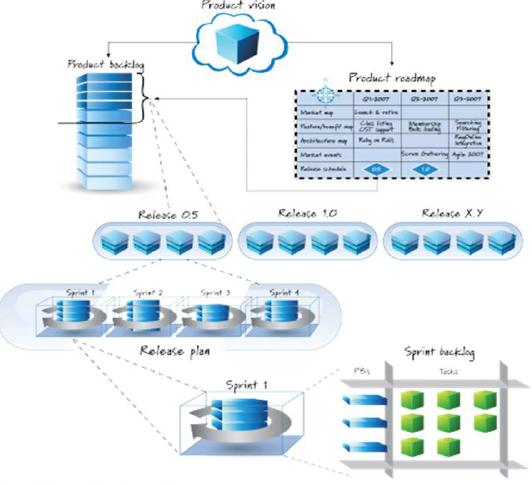
****

**Creación de Productos:** Arranca por 1 idea. De ahí se pasa a un Filtro Estratégico para ver si el producto va a ser viable o no (filtro no tecnológico al día de hoy).



Luego se pasa a una visión (a dónde quiero llegar con el producto), lo cual es el disparador para la creación – **Road Map** – Cómo pienso alcanzar el producto que quiero hacer 🡪 De ahí se deriva el Product Backlog.

Para el Road Map se tienen distintos “quarters”, los cuales conforman un release en un tiempo máximo de 1 mes.



**Release Plan:** Cuántas User Stories en cada sprint para el release.

**Sprint Backlog:** Con tareas (en horas ideales – considerando distracciones) para cada User Story, para cumplir con el Definition of Done, para sacarla del tablero, y pasarla a Done.

Tengo que elegir qué da valor (qué características), ya que, si no lo hace, se lo considera Desperdicio. Tengo que averiguar cuál es la cosa correcta a construir 🡪 **Productividad**. Sobre todo esto se aplica para el MVP (Producto Mínimo Viable – primera release).

**MVP (Minimal Viable Product):**

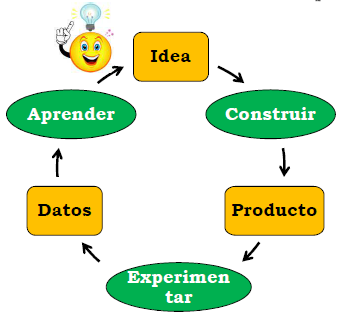
* Es un concepto de Lean Startup que enfatiza el impacto del aprendizaje en el desarrollo de nuevos productos.
* Una premisa clave detrás de la idea de MVP es que se produce un producto real que se puede ofrecer a clientes, y así observar su comportamiento real con el producto/servicio.
* Otra definición (según 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". Este es un aprendizaje validado que viene en forma de si los clientes realmente van a comprar el producto en cuestión.
* Se utiliza para ver lo que la gente realmente hace con respecto a un producto, lo cual es mucho más confiable que directamente preguntarle a la gente qué harían. Tiene como propósito la realimentación, es decir, ver si el producto es correcto o no, lo más rápido posible, ver si el producto tiene valor, en contrapartida al propósito de la ganancia de dinero.
* El desafío en esto es que hay que escuchar al cliente para derivar sus necesidades, a través del MVP.
* 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.

**MVP vs MMF o MMP (Errores Comunes):**



Teniendo como premisa que lo que se quiere lograr es siempre la creación de valor, hay que tener en cuenta que el éxito no es entregar un producto, si no que 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 éstos últimos.

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



La fase **Construir** – 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.



* En esta fase hay que decidir qué tan complejo va a ser el MVP de manera formulada:
  + Requiere juicio.
  + En caso de duda, simplificar.
  + Evitar la construcción excesiva y la promesa excesiva.
  + Cualquier trabajo adicional más allá de lo que se necesita para comenzar el ciclo, podría ser un desperdicio.
* Un MVP no solo habla sobre el diseño del producto y las preguntas técnicas, sino que también sirve para probar hipótesis comerciales fundamentales.
* Por lo tanto, sirve para proporcionar una dosis necesaria de realidad.

**Dilema – La Audacia de Cero:** A menudo es más fácil recaudar dinero cuando se tienen cero ingresos, cero clientes y tracción cero, que cuando se tiene una pequeña cantidad de cada uno de estos elementos.

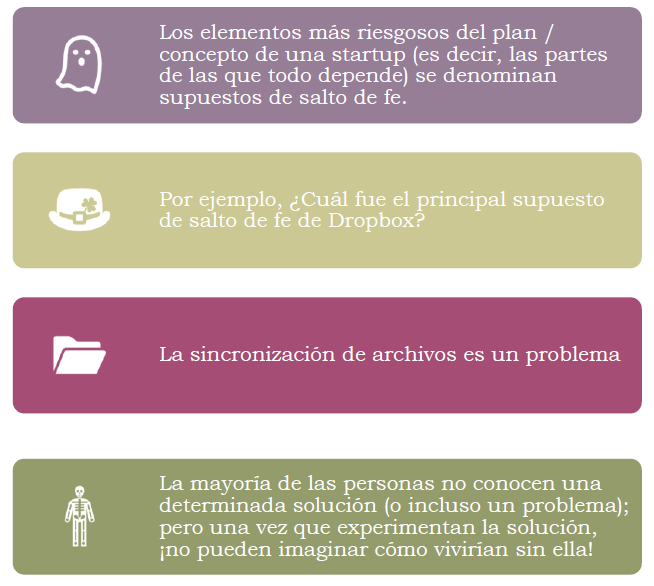
Cero invita a la imaginación, pero los números pequeños hacen preguntas sobre si los números grandes alguna vez se materializarán. Este fenómeno (audacia de cero) crea un incentivo brutal: Aplazar el lanzamiento de cualquier versión de un producto hasta que se esté seguro del éxito.

Si se pospone experimentar con un MVP, van a surgir algunos resultados desafortunados como:

* La cantidad de trabajo desperdiciado puede aumentar.
* Se van a perder los comentarios esenciales.
* El riesgo de que el startup construya algo que nadie quiera puede aumentar.

En cuanto a las compensaciones. ¿Se preferiría atraer capital de riesgo y potencialmente derrocharlo? ¿O se preferiría atraer capital de riesgo y utilizarlo sabiamente? Por ello, se debe usar un MVP para experimentar (inicialmente, en silencio) con los primeros usuarios en el mercado. Luego se verifica el concepto probando **todos** los elementos, comenzando por los más riesgosos.

**Supuestos de “Saltos de Fe”:**



Para estos supuestos se tiene:

* **Hipótesis de Valor:**
  + Prueba si el producto realmente está entregando valor a los clientes después de que comienzan a usarlo.
  + Una métrica de prueba: tasa de retención.
* **Hipótesis de Crecimiento:**
  + Prueba cómo nuevos clientes descubrirán el producto.
  + Una métrica de prueba: tasa de referencia o Net Promoter Score (NPS).

Por último, referido a un MVP, para prepararlo:



**Trazabilidad:** Permite rastrear errores generando más consistencia. Si modifico un archivo en un lugar, puedo ver qué “rompería” o afectaría dicho cambio.

**Gestión de Configuración:** Gestión que hace frente a cambios para establecer y mantener la integridad de un producto de software. Tiene 4 actividades fundamentales:

1. **Identificación de los Ítems de Configuración:** Actividad operativa. Se hace para garantizar la integridad de un ítem, y para poder rastrearlo (para lo cual, primero debo identificarlo – a partir de la definición de su nomenclatura).
2. **Control de Cambios:** Actividad operativa. En este punto entra el concepto de **Línea Base**, el cual se desarrolla más adelante, pero a modo de resumen, es como una “foto” de 1 o más ítems de configuración en un momento del tiempo que han sido revisados, evaluados y aprobados, y que, para ser modificados, tienen que pasar por un procedimiento de control de cambios. Esta línea se la considera estable, es decir que cuenta con el visto bueno de todo el equipo. Existen 2 tipos:
   1. Operacionales: Contiene una versión del producto con código ejecutable.
   2. De Especificación: Sin código.

Un concepto relacionado al control de cambios es el de **Comité de Control de Cambios**, el cual, en caso que haya cambios en la línea base, se reúne para su modificación (o creación previa), y cuenta con una cantidad reducida de personas que sea necesario que estén al tanto de los cambios, para que éstos luego informen a las demás.

1. **Informe y Reporte de estados:** Actividad para dar visibilidad. Existen informes de inventario (como mínimo), así como también informes de quién cambió qué y cuándo (árbol de versiones – historial), y de líneas base en sí mismas.
2. **Realizar Auditorías de Configuración:** Para control y para definir desviaciones. Una auditoría es una revisión objetiva (no depende de criterios) e independiente. Cuando se desarrolla, se lo hace para garantizar la calidad e integridad (es decir, prevenir errores lo antes posible para que esto resulte más barato) una vez que el software ya está funcionando. Existen 3 tipos de auditorías:
   1. Auditoría de Configuración Física: Se asegura que los ítems de configuración estén, y que lo hagan donde estableció que iban a estar.
   2. Auditoría de Configuración Funcional: Se lleva a cabo una vez que se desarrollo la auditoría de configuración física, y se realiza para **validación**, es decir, que el software haga lo que tiene que hacer.
   3. Auditoría de Proyecto: Se analiza el proyecto para ver si cumple con los procesos, o no.

Las auditorías se hacen sobre una determinada línea base, y no sobre todo el repositorio.

Las situaciones por las cuales se generan cambios en el software pueden ser variadas, pero algunos ejemplos tienen su origen en:

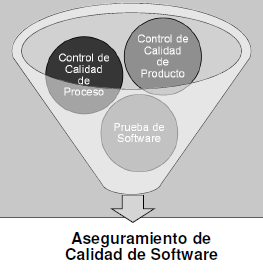
* Cambios del negocio y nuevos requerimientos.
* Soporte de cambios de productos asociados.
* Reorganización de las prioridades de la empresa por crecimiento.
* Cambios en el presupuesto.
* Defectos encontrados al corregir.
* Oportunidades de mejora.

**SCM (Software Configuration Management):** Una de las primeras cosas que se tienen en cuenta cuando se define un proyecto. Es una disciplina transversal (actividad que abarca a todo el proyecto con aplicación de diferentes disciplinas, y no puedo ubicarla en un momento temporal determinado del proyecto) dentro de la ingeniería de software cuyo propósito es mantener la integridad de un producto de software desde que se concibe su idea, hasta que sale del mercado (es decir, durante todo su ciclo de vida).

Administra ítems de configuración (los mantiene para tener una referencia de cada uno), los cuales no son solo piezas de código, sino que es **TODO.**

Otra definición (ANSI/IEEE 828, 1990): 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, y verificar correspondencia con los requerimientos.

**Disciplinas de Soporte del Software:**



**Pero. ¿Por qué se debería gestionar la configuración?** El 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.

**Producto Íntegro:** Un producto íntegro es aquel que cumple con todos los requerimientos (tanto los funcionales como los no funcionales), lo que significa que satisface las necesidades del usuario. Si el mismo es íntegro, se puede mantener la trazabilidad de los elementos (puede ser fácil y completamente rastreado durante su ciclo de vida); y a su vez cumple con las expectativas de costo y criterios de performance.

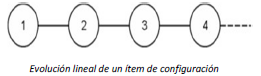
Mantener un producto íntegro no es tarea fácil, ya que hay que trabajar con componentes, y su manejo puede presentar algunos problemas tales como:

* Pérdida de un componente.
* Pérdida de cambios (el componente que se tiene no es el último).
* Sincronía fuente – objeto – ejecutable.
* Regresión de fallas.
* Doble mantenimiento.
* Superposición de cambios.
* Cambios no validados.

**Ítem de Configuración:** Todo y cada uno de los artefactos que forman parte del producto o del proyecto (el cual se necesita mantenerlo identificado), que pueden sufrir cambios (o no), o necesitar ser compartidos entre los miembros del equipo, y sobre los cuales se necesita conocer su estado y evolución.

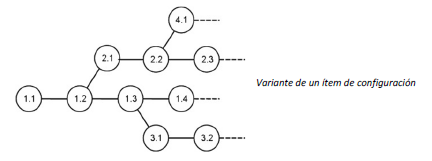
Cada conjunto de cambios sobre un ítem conforma una nueva **versión**:

* 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:



**Variante:**

* Una variante es una versión de un ítem de configuración (o de la configuración) que evoluciona por separado.
* Las variantes representan configuraciones alternativas.
* Un producto de software puede adoptar distintas formas (configuraciones) dependiendo del lugar donde se instale.
* Por ejemplo, dependiendo de la plataforma (máquina + SO) que al soporta, o de las funciones opcionales que haya de realizar o no.



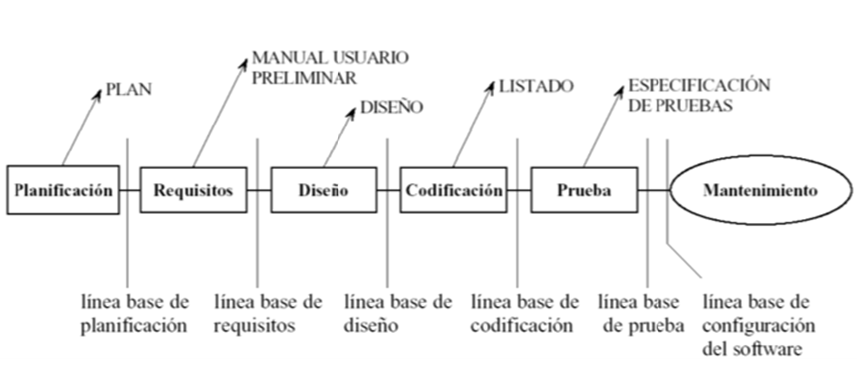
Existen 3 tipos de ítems:

1. **De Producto:** Por ejemplo, un archivo de arquitectura.
2. **De Proyecto:** Por ejemplo, el plan de proyecto.
3. **De Iteración.**

**Configuración:** Conjunto de ítems de configuración que conforman un producto de software en un momento determinado.

**Línea Base:**

* Se utilizan etiquetas para “marcar” las baselines.
* No se debe confundir a este concepto con el de versión del producto.
* Configuración marcada (ha sido revisada formalmente) sobre la que se ha llegado a un acuerdo.
* Sirve como base para desarrollos posteriores y puede cambiarse solo a través de un procedimiento formal de control de cambios.
* Se define por el equipo a partir de una configuración en particular, por tener interés sobre ésta.
* Permiten ir atrás en el tiempo y reproducir el entorno de desarrollo en un momento dado del proyecto.
* Con respecto a su representación, puede ser:
  + **De Especificiación** (Requerimientos, Diseño).
  + **De Productos** que han pasado por un control de calidad definido previamente.



**Calidad:** Conjunto de características subjetivas para satisfacer necesidades y expectativas. En el software es difícil de medirla, ya que el mismo es intangible. Ingeniería de Software.

**Software:** Conjunto de programas con la documentación que lo acompaña. Es todo resultado que se obtenga a partir del procesamiento de información, y también se lo puede definir como conocimiento que puede estar empaquetado en distintos formatos. Existen 3 tipos básicos de software: System Software, Utilitarios y Software de Aplicación.

El software es intangible, lo que le da un alto nivel de complejidad, y a su vez es una actividad humano-intensiva que tiene como principal materia prima a la persona por su capacidad de pensar.

Está conformado por:

* Diversos programas independientes.
* Archivos de configuración que se utilizan para ejecutar estos programas.
* Documentación que describe la estructura del sistema.
* Documentación para el usuario que explica cómo utilizar el sistema.

Se clasifican en:

* **Productos Genéricos:** Son sistemas aislados producidos por una organización de desarrollo y que se venden en el mercado abierto a cualquier cliente. La especificación es controlada por la organización que desarrolla.
* **Productos Personalizados (o hechos a medida):** Son sistemas requeridos por un cliente en particular. La especificación, por lo general, es desarrollada y controlada por la organización que compra el software.

La calidad que alcanza un software recae en la personalidad y el intelecto de los miembros del equipo que lo crean. Es decir, que si se aplicara un proceso que ha entregado excelente resultados para un equipo, en otro equipo, muy posiblemente no se alcance el resultado de calidad logrado por el primer equipo en cuestión.

**Proceso de Software:** Proceso estructurado de actividades para desarrollar un sistema de software, las cuales varían dependiendo de la organización, y del tipo de sistema que se quiere desarrollar. A su vez, si se planea que el mismo sea administrable, previamente tiene que ser explícitamente modelado.



A la hora de desarrollar software, hay que considerar que éste **NO** es una manufactura, y a continuación algunas razones:

1. El software es menos predecible.
2. No hay producción en masa, casi ningún producto de software es igual a otro.
3. No todas las fallas son errores.
4. El software no se gasta.
5. El software no está gobernado por las leyes de la física.

# La Crisis del Software.

Este término tiene origen en 1968 por F. L. Bauer, quien recalcó la dificultad para generar software libre de defectos, fácilmente comprensible, y que sea verificable. Las causas son:

* La introducción de nuevas tecnologías de hardware que generaron la posibilidad de construir software más complejo, y de mayor tamaño.
* La complejidad que supone la tarea de programar.
* Los cambios a los que tiene que ser sometido un software para ser continuamente adaptado a las necesidades de los usuarios.

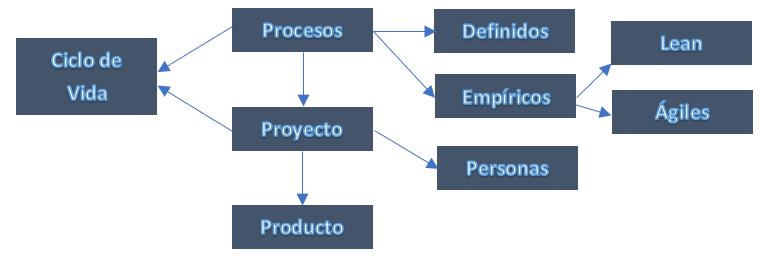
Por otro lado, este proceso puede presentar algunos problemas:

* La versión final del producto no satisface las necesidades del cliente. A su vez, un cambio de requerimientos implica un cambio significativo en el software.
* No es fácil extenderlo y/o adaptarlo. Agregar más funcionalidad en otra versión es casi una misión imposible.
* Mala documentación.
* Mala calidad.
* Más tiempos y costos que los presupuestados.

# Ingeniería de Software.

Es una disciplina de la ingeniería que se preocupa de todos los aspectos de la producción de un software, **desde** las primeras etapas de la especificación, **hasta** el mantenimiento del sistema después que se pone en operación. Parmas [1987] definió a la ingeniería de software como “Multi-person construction of multi-version software”. En esta definición existen dos frases clave:

* La 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 den trabajar con restricciones financieras y organizacionales, por lo que buscan soluciones considerando estas restricciones.
* Disciplinas: Cuenta con 3:

1. Disciplinas Técnicas 🡪 Ayudan a construir el Producto (Construcción).
   1. Requerimientos.
   2. Análisis y Diseño.
   3. Construcción.
   4. Prueba.
   5. Despliegue.
2. Disciplinas de Gestión 🡪 Proyectos (cuyo resultado es un producto).
   1. Planificación de Proyecto.
   2. Monitoreo y Control de Proyectos.
3. Disciplinas de Soporte 🡪 Complementarias (se ajustan al proyecto y al producto).
   1. Gestión de Configuración de Software.
   2. Aseguramiento de Calidad.
   3. Métricas.

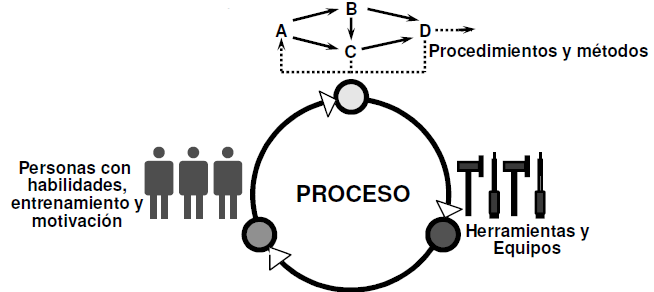
Se adapta

Obtiene como resultado

Se incorpora

**Proceso:** Conjunto de tareas interrelacionadas o pasos ejecutados para un propósito dado. El mismo se adapta al proyecto, y se adhiere a un ciclo de vida determinado. Se tiene que definir cuál proceso se va a usar, para ver qué necesito del proyecto, y qué no.

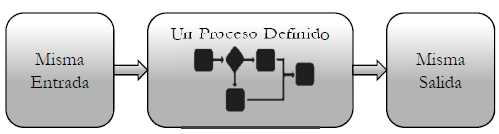
**Proceso de Software:** Un conjunto de actividades, métodos, prácticas, y transformaciones que la gente usa para desarrollar o mantener software y sus productos asociados.



**Tipos de Procesos:**

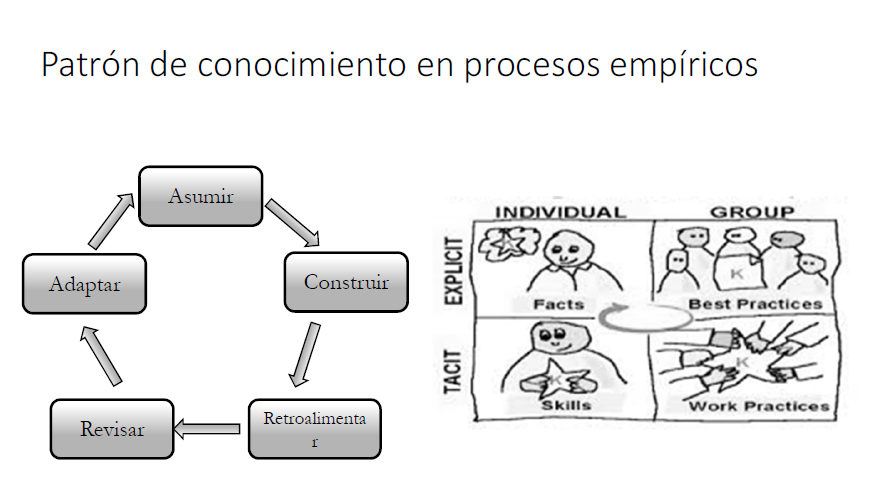
**Definidos:** Son solo los lineamientos, es decir, que todavía quedan cosas sin definir. Ejemplos: Cascada, PUD, RUD, Iconix, Métricas. Están inspirados en las líneas de producción:

* 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.
* Estandarizados.
* Siempre se aplica el mismo procedimiento (se definen entradas, salidas, procedimientos, roles, actividades y etapas).
* Ante iguales entradas, se obtienen iguales salidas mediante la aplicación de un proceso definido.



* El proceso implementa un ciclo de vida.
* Calidad del proceso 🡪 Calidad del producto.

**Empíricos** (Inspección y Adaptación):

* Asume procesos complicados con variables cambiantes. Cuando se repite el proceso, se pueden llegar a obtener resultados diferentes.
* La administración y control es a través de inspecciones frecuentes y adaptaciones.
* Son procesos que trabajan bien con procesos creativos y complejos.
* Dependen de la experiencia de sus miembros.
* Tienen un Feedback (permite a un proceso aplicar una mejora, la cual se aplica sobre las personas).

Proceso Ciclo de Vida 🡪 El **Ciclo de Vida** se refiere a cómo encaro la ejecución del proyecto (cuántas tareas se hacen, y en qué momento); modelo genérico (no una descripción definitiva) de los procesos de software, es decir, es una abstracción del proceso que se usa para explicar los distintos enfoques del desarrollo de software. En definitiva, un ciclo de vida de software es una representación de un proceso, el cual grafica una descripción del mismo desde una perspectiva particular. Algunas características:

* Especifica las fases del proceso, y el orden en el que se llevan a cabo (requerimientos, especificación, diseño, etc.).
* Es una guía para la administración del proyecto, ya que indica el progreso a través de hitos.
* Los modelos de ciclos de vida se han vuelto necesarios debido a que los sistemas son más complejos por el aumento de funcionalidad, y la mayor variedad de usuarios.
* Son independientes de los procedimientos de cada actividad del ciclo de vida.

Este Ciclo también es llamado **Modelo de Procesos**, y existen 3 tipos:

1. **Secuencial** (ejemplo: cascada): Toma las actividades fundamentales del proceso (especificación, desarrollo, validación y evolución), y los representa como fases separadas del proceso (Especificación de requerimientos, diseño de software, implementación, pruebas, etc.).
   1. Desarrollo dirigido por un plan.
   2. Cada etapa genera documentación para realizar un monitoreo constante contra el plan.
   3. Muy útil cuando los requerimientos son claros y es poco probable un cambio drástico durante el desarrollo.
   4. La documentación puede ser burocrática y excesiva.
   5. En etapas finales, puede ser que haya que hacer re-trabajo por cambios de requerimientos o fallas de diseño.
2. **Iterativo** (ejemplo: iterativo-incremental): 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.
   1. La especificación, desarrollo y validación están entrelazadas en lugar de separadas y aisladas.
   2. Rápida retroalimentación a través de las actividades.
   3. Muy útiles para sistemas de requerimientos cambiantes (por ejemplo, e-commerce, empresariales, etc.).
   4. Más fácil y menos costoso implementar cambios.
   5. Cada iteración genera funcionalidad para el cliente.
   6. Los incrementos progresivos tienden a degradar la estructura del sistema.
3. **Recursivo** (ejemplo: espiral): Se inicia con algo en forma completa, como una subrutina que se llama a sí misma, e inicia nuevamente. Se presenta un prototipo que va mejorando con cada vuelta.
   1. Se generan productos independientes de la implementación, que pueden ser reusables en sistemas de características similares.
   2. Puede ser más costos en tiempo y dinero readaptarlos para reutilizarlos para los diferentes proyectos.
   3. La tecnología puede ser obsoleta.
   4. Pueden carecer de mantenimiento o documentación.

**Proyecto:** Medio que está orientado a objetivos (los cuales son Claros y Alcanzables – factibles; así como también son los que guían al proyecto; y no deben ser ambiguos), y por lo general dicho objetivo es obtener un **Producto** con alcance gradual (ya que implica tareas interrelacionadas basadas en esfuerzo y recursos). Un proyecto tiene una duración limitada en el tiempo (tiene principio y fin), y a su vez cada proyecto es ÚNICO. El desarrollo de un proyecto conlleva la utilización (afectación) de recursos, como se mencionó anteriormente, que, a su vez, cuando el mismo termina, éstos últimos quedan libres (liberados). Dicho esto último de otra manera, es un conjunto de tareas interrelacionadas basadas en esfuerzo y recursos.

Aclaraciones:

* Una línea de producción NO ES UN PROYECTO.
* Todos los proyectos por similares que sean, tienen características que los hacen únicos.

Otra definición: Unidad de gestión u organización que administra recursos para obtener un resultado (producto o servicio de características únicas). Tiene que administrarse SÍ O SÍ (debe tener un plan), y para eso debe tener **Seguimiento** (o monitoreo), el cual es permanente, y **Supervisión** (o control), el cual es en base a hitos determinados (puntos de control – me detengo para ver cómo me está yendo). Estos hitos se asocian a fines de etapas o con entregables.

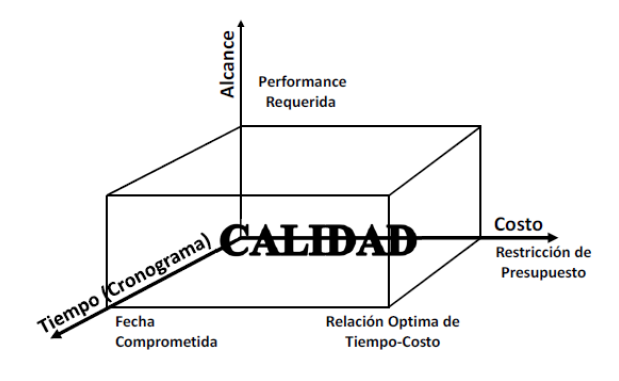
**Administración de proyectos:** “… tener el trabajo hecho …” en tiempo, con el presupuesto acordado y habiendo satisfecho las especificaciones o requerimientos; aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades del proyecto para satisfacer los requerimientos del mismo.

Para administrar un proyecto, se tiene que:

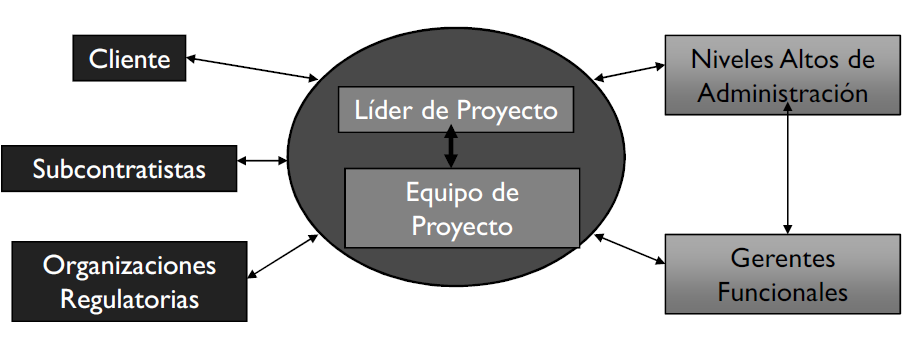
* Identificar los requerimientos.
* Establecer objetivos claros y alcanzables.
* Adaptar las especificaciones, planes, y el enfoque a los diferentes intereses de los involucrados (stakeholders).

Considerando esta administración, se tiene que tener en cuenta el problema de “**La Restricción Triple**”:

* Objetivos de proyecto (**Alcance**): ¿Qué está el proyecto tratando de alcanzar? Son los requerimientos del proyecto, es decir, los límites o el ámbito sobre el cual se va a mover el mismo. Esta variable es la que primero se negocia con el cliente.
* Tiempo: ¿Cuánto tiempo debería llevar completarlo? Hace referencia al calendario, cuáles serán las fechas especificadas para realizar las entregas que determinarán el avance del proyecto.
* Costos: ¿Cuánto debería costar? Son los recursos que se verán implicados en el desarrollo del proyecto. Esto incluye equipamiento, infraestructura, equipos, salarios, entre otros.



El balance de estos tres factores afecta directamente 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 del Líder de Proyecto balancear estos tres objetivos (que a menudo compiten entre ellos), y cuyo rol se muestra a continuación:



Para ser un líder, uno debe sentirse cómodo con los cambios y entender a la organización. El líder debe tener los **Hard Skills** (conocimientos del producto, herramientas y técnicas) y los **Soft Skills** (capacidad de trabajar con gente, y son los más difíciles de conseguir – comunicación, liderazgo, creatividad, organización, motivación, empatía). Es responsabilidad del líder de proyecto:

* Definir el alcance del proyecto.
* Identificar involucrados.
* Detallar tareas, así como estimar tiempos y requerimientos.
* Identificar recursos y presupuestos.
* Identificar y evaluar riesgos.
* Preparar planes de contingencia.
* Controlar hitos.
* Participar en las revisiones de las fases del proyecto.
* Administrar el proceso de control de cambios y producir reporte de estado.

**El Equipo:** Grupo de personas comprometidas en alcanzar un conjunto de objetivos, de los cuales se sienten mutuamente responsables. Tienen diversos conocimientos y habilidades, trabajan juntos desarrollando sinergia y, en general, es un grupo pequeño. Tienen sentido de responsabilidad como una unidad.

**Stakeholders:** Son los interesados del proyecto. Incluye el equipo del proyecto, el equipo de dirección, el líder del proyecto y el patrocinador. Este último debe tener jerarquía en la empresa y garantizar recursos. En metodologías ágiles, el líder del proyecto es el Scrum Master, y el patrocinador el Product Owner.

**Producto Software:** Cada nueva versión es desarrollada **incrementalmente** en una serie de pasos.

**Equipo de Proyecto:** Un grupo de personas comprometidos en alcanzar un conjunto de objetivos de los cuales se sienten mutuamente responsables. Algunas características son:

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

# Gestión Tradicional de Proyectos.

Para planificar un proyecto de software se requiere de un **Plan de Proyecto** (Plan de Desarrollo de Software – Un plan es a un proyecto lo que una hoja de ruta a un viaje), el cual es el documento en el cual se especifica:

* ¿Qué es lo que hacemos? Es decir, se especifica el alcance del proyecto, el cuál guiará el desarrollo del mismo.
* ¿Cuándo lo hacemos? Especifica el calendario, las fechas estipuladas para cubrir el alcance del proyecto.
* ¿Cómo lo hacemos? Es decir, las actividades y tareas que se deberán llevar a cabo para cubrir el alcance del proyecto, en el tiempo especificado.
* ¿Quién lo va a hacer? Definición de responsables, los que van a llevar a cabo cada una de las tareas descriptas anteriormente en el cómo.

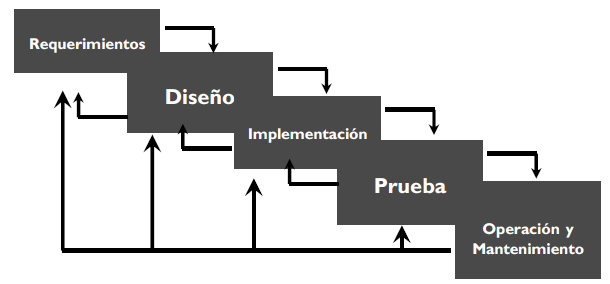
Es de suma importancia que este documento sea correctamente mantenido y actualizado de manera permanente, es decir, “No está escrito en piedra”, y tiene ciclos de cambios. El mismo funciona como un paraguas o caparazón para el equipo. Si el plan de proyecto no se actualiza, es muy probable que el proyecto tienda al fracaso. A su vez, tiene varias partes:

* **Definición de un objetivo del proyecto:** Lo que va a hacer.
* **Alcance del proyecto:** Relevamiento de requerimientos, análisis, diseño, etc. Hace referencia al trabajo para cumplir con el objetivo (tareas). En caso del PRODUCTO, el alcance serían todas las características que pueden incluirse en un producto o servicio. (por ejemplo, objetivos de los casos de uso de diseño). En Caso del PROYECTO, el alcance es todo trabajo y solo trabajo que debe hacerse para entregar el producto o servicio con todas las características y funciones especificadas (por ejemplo, la ERS).

El cumplimiento del alcance del **Proyecto** se mide contra el **Plan de Proyecto**, mientras que el cumplimiento del alcance del **Producto** se mide contra la **Especificación de Requerimientos**.

* **Definir qué proceso se va a usar.**
* **Definir qué ciclo de vida:** (Del proyecto, no del producto). Éste define:
  + Qué trabajo técnico debería realizarse en cada fase.
  + Quién debería estar involucrado en cada fase.
  + Cómo controlar y aprobar cada fase.
  + Cómo deben generarse los entregables.
  + Cómo revisar, verificar y validar el producto.

La mayoría de los ciclos de vida comparten algunas características a saber: Los costos y el personal son bajos al inicio, y más altos hacia el final, cayendo abruptamente cuando el proyecto termina.



* **Dimensionamiento del Proyecto:** O estimaciones.
  + **Tamaño:** Por ejemplo, a nivel de módulos, alcance, requerimientos, CU, clases, líneas de códigos (aunque al día de hoy ya no sirve).
  + **Esfuerzo:** Del tamaño se deriva el esfuerzo. Esto se mide en Horas-Persona Lineales (1 cosa a la vez por persona). De esto obtengo un total de cantidad de horas.
  + **Tiempo:** Del esfuerzo deriva el tiempo. Base para la calendarización.
  + **Costo:** Por ejemplo, si requiero que sea más rápida la entrega, tengo que tener en cuenta que voy a tener que requerir de horas extras por parte de los trabajadores, por ende, eso es más gasto porque a esas horas hay que pagarlas.
* **Riesgos:** Es una probabilidad de que un evento no deseado ocurra en un proyecto, es decir, un evento esperando por suceder, y que podría llegar a comprometer el éxito del proyecto. No es lo mismo que problema, ya que los tratamientos en cada caso son distintos. Por ende, un riesgo tiene una probabilidad asociada (probabilidad de ocurrencia de un factor que genere algún daño o pérdida). A los riesgos se los mide: .

Esta es una fórmula de cálculo (por lo que permite determinar el orden de los riesgos), en donde probabilidad hace referencia a la posibilidad que algo se presente, e impacto al daño que produce dicho acontecimiento posible.

Un ejemplo de riesgo (y en este caso, bastante alto), es el de equivocarse a la hora de la determinación de los requerimientos.

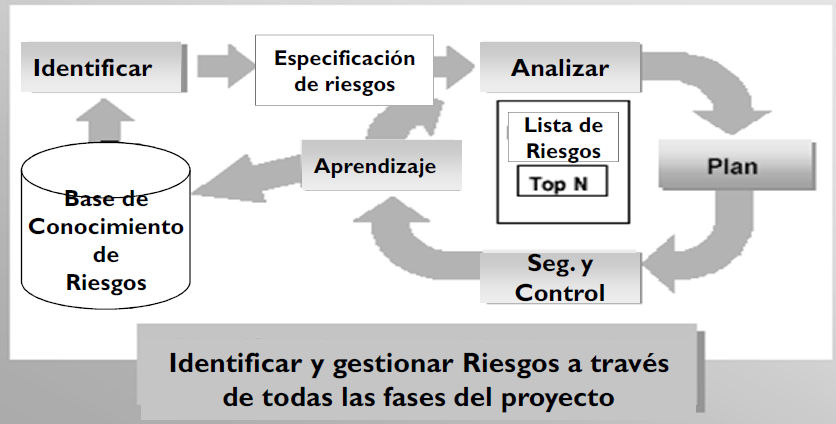
Para los riesgos, hay que tener estrategias **pro**activas (actuar con anticipación para mitigar dichos riesgos), y de ahí un plan de contingencia. En contrapartida, tener medidas **re**activas no es tan eficiente, ya que el suceso ya pasó.

Mitigación 🡪 Plan de Contingencia 🡪 Evento Disparador

Para la gestión de riesgos se tiene:

Consideraciones:

* + No existen proyecto sin riesgos.
  + Los riesgos pueden provocar incrementos en los costos, o desbordamiento del proyecto.
  + Se clasifican en:
    - **Riesgos del Proyecto:** Afectan la calendarización o los recursos del proyecto.
    - **Riesgos del Producto:** Afectan a la calidad o al rendimiento del software que se está desarrollando.
    - **Riesgos del Negocio:** Afectan a la organización que desarrolla o suministra el software.



* **Calendarización o programación del proyecto:** Esto se realiza con las tareas ya definidas.
* **Definición de métricas:** El proyecto incluye las métricas, utilizadas para informar, motivar, comparar, entender, evaluar, predecir y mejorar. **Unidad** es una cantidad particular, definida y adoptada por convención, con la que poder comparar otras cantidades de la misma clase, para expresar sus magnitudes respecto a esa cantidad particular. Se clasifican en:
  + Métrica Directa: Es aquella que se puede obtener sin depender de ninguna otra métrica, y cuya forma de medir es a través de un método de medición, por ejemplo, líneas de código de un módulo.
  + Métrica Indirecta: Es aquella que proviene de una función de cálculo cuyos argumentos son otras métricas directas o indirectas, por ejemplo, total de horas de programación.

Tienen **Validez** (relacionada a la exactitud de la métrica, es decir, la proximidad con respecto al valor verdadero), y **Confiabilidad** (precisión, hace referencia a la repetibilidad o reproductibilidad de la medida. Tienen 3 **Dominios**:

* + Producto.
  + Proceso.
  + Proyecto.
* **Cómo se hace seguimiento y control del proyecto:** Qué reuniones/reportes hay que hacer; a quién le doy dichos reportes; con qué frecuencia se dan dichas reuniones, etc.

Causas de Fracasos en Proyectos:

* Fallas al definir el problema.
* Planificar basado en datos insuficientes.
* La planificación la hizo el grupo de planificaciones.
* No hay seguimiento del plan de proyecto.
* Plan de proyecto pobre en detalles.
* Planificación de recursos inadecuada.
* Las estimaciones se basaron en “supuestos” sin consultar datos históricos.
* Nadie estaba a cargo.

# Metodologías Ágiles.

**Objetivo del Enfoque Ágil:** El objetivo primordial es construir software de forma rápida para aprovechar las actuales oportunidades, y responder ante la amenaza competitiva, convirtiendo la velocidad de entrega en el requerimiento fundamental de los sistemas de software: “software de entregas rápidas en un entorno cambiante”.

Los desarrollos ágiles se utilizan para entornos con gran variabilidad de requerimientos, ya que los clientes encuentran imposible predecir cómo un sistema afectará sus prácticas operacionales, o qué cambios habrá en el entorno (mercado o políticas de negocio) que pueden dejar el sistema completamente obsoleto.

Los procesos de desarrollo de software rápido se diseñan para producir rápidamente un software útil, en cual no se desarrolla como una sola unidad, sino como una serie de incrementos, y cada uno de ellos incluye una nueva funcionalidad del sistema.

**Manifiesto Ágil** – Declaración de principios ágiles, los cuatro valores:

1. **Individuos e interacciones** por sobre procesos y herramientas: Se valora los individuos e interacciones **POR SOBRE** (ambos son iguales de importantes, pero se presta más atención en uno) procesos y herramientas. Se les da mucha importancia a los stakeholders.
2. **Software funcionando** por sobre documentación detallada: Se eliminan las “recetas” de un proyecto a la hora de hacer software. Si hay documentación con nivel de detalle en base al criterio de elegir qué documentos y cuáles no. La documentación debe tener valor para el cliente (es decir, en el sentido que el software le satisfaga sus necesidades).
3. **Colaboración** por sobre negociación con el cliente: Antes de tener un contrato prefijado, se valora más una colaboración fluida entre desarrolladores y los clientes.
4. **Responder a cambios** por sobre seguir un plan.

De estos 4 valores se desprenden 12 principios:

1. La prioridad es satisfacer al cliente a través de releases tempranos y frecuentes (2 semanas a un mes): Pero siempre software con valor (que le sirva al cliente, y cumpla con sus expectativas). Con respecto a las entregas, sirven para poder tener un Feedback más rápido, lo que nos evita tener que hacer grandes cambios en el futuro que modifiquen todo el proyecto, y sean difíciles de desarrollar. Como conclusión, con entregas tempranas y frecuentes, se solucionan los problemas mucho más rápido y antes.
2. Recibir cambios de requerimientos, aún en etapas finales: Con análisis de impacto previo, pero siempre con una postura de estar abiertos antes cambios.
3. Releases frecuentes (2 semanas a un mes).
4. Técnicos y no técnicos trabajando juntos TODO el proyecto: Esto para evitar hasta tener una release para obtener un Feedback, por ende, el proceso se hace más rápido.
5. Hacer proyectos con individuos motivados: Personas motivadas 🡪 Éxito en el proyecto (porque se les da importancia a los propios individuos).
6. El medio de comunicación por excelencia es cara a cara: Parte de la comunicación no es verbal (a nivel de gestos, o lo que no se dice al decir tal cosa, por ejemplo).
7. La mejor métrica de progreso es la cantidad de software funcionando: Metodologías ágiles reducen las métricas a usar 🡪 Se reduce a la propia cantidad de software que funciona.
8. El ritmo de desarrollo es sostenible en el tiempo.
9. Atención continua a la excelencia técnica: Que cada miembro del equipo tenga excelencia técnica (capacidad para producir software – que tenga calidad).
10. Simplicidad – Maximización del trabajo no hecho: Evitar el “ya que estamos, agrego esto…”, ya que esto termina en no agregar nada de valor. Se deben plantear soluciones de la forma más simple posible, pero que entregue valor.
11. Las mejores arquitecturas, diseños y requerimientos emergen de equipos auto organizados: Miembros con expertise suficiente para construir productos de calidad.
12. A intervalos regulares, el equipo evalúa su desempeño y ajusta la manera de trabajar.

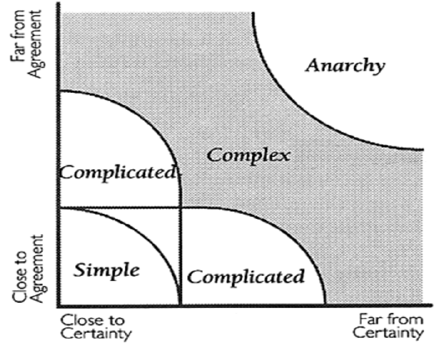
¿Qué significa Ágil?

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.

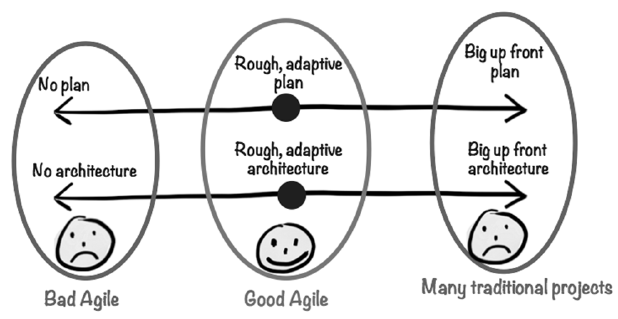
Las metodologías ágiles le dan más foco al producto, **priorizando** al cliente.

Por otro, lado. ¿Cuándo se aplica Agile?

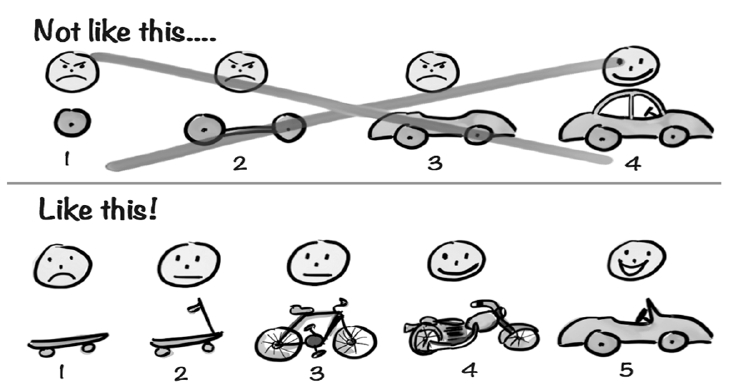


* Agile da mejores resultados cuando los problemas a ser resueltos caen dentro del espacio “Complex”.
* El desarrollo de nuevos productos y Knowledge Work tienden a estar en el espacio Complex.
* Investigación está dentro de Anarchy.
* Mantenimiento por lo general cae en Simple.

Hay que considerar que ser ágil no es ser indisciplinado:



Pero tampoco es que hay que hacer todo por partes pequeñas:



# Filosofía Lean.

Es una filosofía que se basa en flujos, y ya no en iteraciones. Con flujo se refiere a conjunto de paradas, o de unidades, que no son más que distintas funcionalidades que van a ir fluyendo a medida que se crean. Estas unidades pueden tener una prioridad, y se ubican en una cola de unidades de trabajo. Cuando termino una de estas unidades, libero recursos, y de esta forma paso al siguiente (Just in Time).

Se describe lo mínimo necesario para arrancar con el flujo (voy agregando requerimientos en el proceso). Consta de una serie de 7 principios:

1. Eliminar el desperdicio: Evitar que las cosas se pongan viejas antes de terminarlas, o evitar re-trabajo. Tiene que ver con el principio ágil de software funcionando y el de simplicidad (arte de maximizar lo que no hacemos). Algunos desperdicios, dependiendo el caso y la medida en que se toma, pueden ser prototipos, estimaciones, documentación, etc. Otra cosa que se considera desperdicio, es lo mencionado anteriormente de “ya que estamos, agrego esto…”.

Se tiene como objetivo 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: el inventario.
  + En software: es el trabajo parcialmente hecho, y las características extra.
  + El 20% del software que entregamos contiene el 80% del valor (regla de Pareto).

Desperdicios en manufactura:

* + - Producción en exceso.
    - Stock.
    - Pasos extra en el proceso.
    - Búsqueda de información.
    - Defectos.
    - Esperas.
    - Transportes.

Desperdicios en software:

* + - Características extras.
    - Trabajos a medias.
    - Proceso extra.
    - Movimiento.
    - Defectos.
    - Esperas.
    - Task Switching.

1. 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.
   * Se debe generar nuevo conocimiento y codificarlo de manera tal que sea accesible a toda la organización.
   * Muchas veces los procesos “estándares” hacen difícil introducir en ellos mejoras.
2. Embeber la integridad conceptual: Encastrar todas las partes del producto o servicio, que tenga coherencia y consistencia (tiene que ver con los requerimientos no funcionales). La integración entre las personas hace el producto más íntegro. Se necesita más disciplina, no menos.
   * **Integridad Percibida:** El producto total tiene un balance entre función, uso, confiabilidad y economía que le gusta a la gente.
   * **Integridad Conceptual:** Todos los componentes del sistema trabajan en forma coherente en conjunto.
   * El objetivo es construir con calidad desde el principio, no probar después.
   * Dos clases de inspecciones:
     + Inspecciones luego de que los defectos ocurren.
     + Inspecciones para prevenir defectos.
   * Si se quiere calidad, no inspeccione después de los hechos.
   * Si no es posible, inspecciones luego de pasos pequeños.
3. Diferir compromisos: El último momento responsable para tomar decisiones (en el cual todavía estamos a tiempo). Si nos anticipamos, tenemos información parcial.
   * Se relaciona con el principio ágil: decidir lo más tarde posible pero responsablemente. No hacer trabajo que no va a ser utilizado. Enlaza con el principio anterior de aprendizaje continuo, mientras más tarde decidimos más conocimientos (máxima información disponible para tomar una decisión).

Las decisiones deben tomarse en el último momento que sea posible.

* + No significa que todas las decisiones deben diferirse.
  + Se debe tratar de tomar decisiones reversibles, de forma tal que pueda ser fácilmente modificable.
  + Vencer la “parálisis de análisis” para obtener algo concreto terminado.
  + Las mejores estrategias de diseño de software están basadas en dejar abiertas opciones de forma tal que las decisiones irreversibles se tomen lo más tarde posible.

1. Dar poder al equipo: Ejemplo, vamos a comer a un restaurante, y no nos metemos en la cocina del restaurante. Nos fijamos en el precio, pedimos y esperamos. Hay mucho micro management, el dueño no decide cuánta sal poner a la comida.
   * Respetar a la gente.
     + Entrenar líderes.
     + Fomentar buena ética laboral.
     + Delegar decisiones y responsabilidades del producto en desarrollo al nivel más bajo posible.
   * Ágil: El propio equipo pueda estimar el trabajo.
2. Ver el todo – Optimizar el total: Tener una visión holística, de conjunto (el producto, el valor agregado que hay detrás, el servicio que tiene los productos como complemento).
3. Entregar lo antes posible:
   * Entregar rápido: Estabilizar ambientes de trabajo a su capacidad más eficiente y acotar los ciclos de desarrollo.
   * Entregar rápidamente: Esto hace que se vayan transformando “n” veces en cada iteración. Incrementos pequeños de valor. Llegar al producto mínimo que sea valioso. Salir pronto del mercado – relacionado con el principio ágil de entrega frecuente.

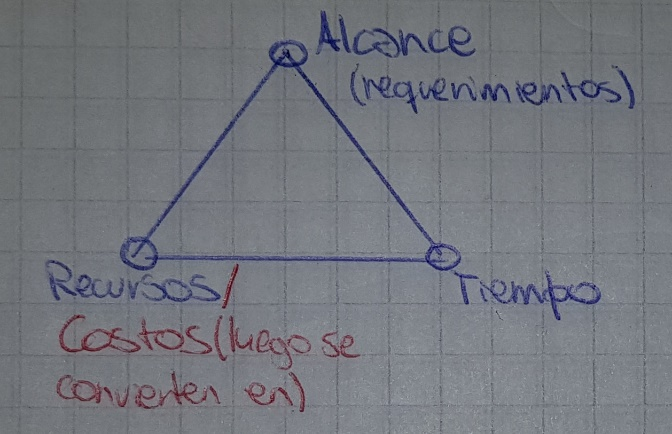
# User Stories.

La parte más difícil de construir un sistema de software es decidir precisamente qué construir. Ninguna otra parte del trabajo conceptual:

* Es tan difícil como establecer los requerimientos técnicos detallados.
* Afecta tanto el sistema resultante si se hace incorrectamente.
* Es tan difícil de rectificar más adelante.

**User Stories**: Técnica (manifestación concreta de un enfoque filosófico) para trabajar requerimientos en desarrollos ágiles.

Por esto se plantean 3 principios ágiles que están relacionados entre sí:



El **enfoque** plantea que de acuerdo al alcance (**FIJO**, y definido por el cliente, ya que éste es el que sabe lo que quiere y decide), se define lo demás. Una vez definido el alcance, se determina el tiempo y recursos (**VARIABLES**) para poder llevar adelante el proyecto. Puede ocurrir que se incorporen cambios en los requerimientos, lo cual es algo muy costoso, ya que crece el alcance y, por ende, las variables tiempo y recursos vuelven a cambiarse y tienen que adaptarse. Por eso es que se presenta cierta resistencia a los cambios, aunque el cliente no entienda el trabajo que hay por detrás ante un cambio en el software que nos pide.

Entonces, Ágil lo cambia todo, ya que los requerimientos CAMBIAN (**VARIABLES**), mientras que el tiempo y los recursos ya no (**FIJOS**). El objetivo de Ágil es el requerimiento “Just in Time” (que se relaciona con los principios Lean 1 y 4, así como con el principio Agile 11). Esto me permite empezar con una visión del producto, y con algunos requerimientos identificados, lo que es suficiente para arrancar a trabajar con el ciclo ágil, ya que nos da un Feedback que permite las correcciones para la siguiente vuelta.



Ágil plantea un mecanismo de compensación, en donde ya no se ocupe tanto tiempo en la especificación de requerimientos, pero que el cliente esté “on-side”, es decir, cerca nuestro para que el trabajo sea colaborativo, y de esta forma los cambios se hagan de acuerdo a sus preferencias, que él tome las decisiones con nosotros para que luego, al presentarle una versión del software, éste esté satisfecho. En contrapartida, si la comunicación no es face-to-face, el cliente, al no entender el trabajo de desarrollo, nos pide algo, nosotros como profesionales lo entendemos de una forma, lo desarrollamos, lo codificamos, probamos y todo el trabajo que conlleva, pero a la hora de presentárselo al cliente, éste nos dice que no es lo que él quería (no se hace cargo de sus decisiones).

Como conclusión a esta idea, lo que se quiere es que haya lo que se denomina “clientes disponibles”, para que de esta forma se baje la carga de documentación formal, pero siempre dejando en claro que en ÁGIL SI SE DOCUMENTA.

Partes de una User Story.

Técnica desarrollada por Mike Cohn. Las partes son 3: Tarjeta (Card), Conversation y Confirmation. Es una técnica de requerimientos a nivel de usuario/negocio (NO DE SOFTWARE).

Una User Story en la tarjeta describe una funcionalidad esperada, y un recordatorio de que el equipo de desarrollo tiene que hablar con el **Product Owner**, el cual es alguien que sabe qué quiere, y que tiene el poder de decisión necesario para poder priorizar los requerimientos.

Siguiendo con el desarrollo de las partes de una User Story:

* **La parte visible – Tarjeta (Card):** Tiene frente y dorso. En el frente no se determina una especificación de requerimiento, ya que no hay detalle, y esto se compensa con la otra parte (la conversación). Al dorso tiene expresiones de las pruebas de aceptación para garantizar que la User Story se implementó bien. El que escribe estas tarjetas es el Product Owner.
* **Conversation (parte más importante):** Es el cara a cara. La técnica dice que esta conversación no queda asentada en ningún lado, aunque por decisión y con acuerdo previo entre todas las partes, ésta puede ser documentada, por ejemplo, grabada. Se entiende que “no hace falta” por el concepto de “Just in Time” descripto anteriormente. Esta parte tiene la ventaja de evitar la situación que también se describió antes, en la que el cliente nos pide algo, nosotros lo desarrollamos, y cuando se lo presentamos, nos dice que no fue lo que él nos pidió porque no se hace cargo de sus decisiones, ya que la conversación no quedó asentada en ningún lado para poder corroborarlo. Una parte de la conversación se puede guardar en las tarjetas (en el frente), y en el dorso las pruebas de aceptación.
* **Confirmation:** Definición de un acuerdo para hacer, para de esta forma poder mostrar algo que se creó 🡪 Pruebas de aceptación, y si el cliente las acepta, se sigue adelante, pero siempre considerando que lo que se le muestra al cliente, tiene que tener valor de negocio, si no, al Product Owner no le sirve.

Forma de expresar las User Stories.

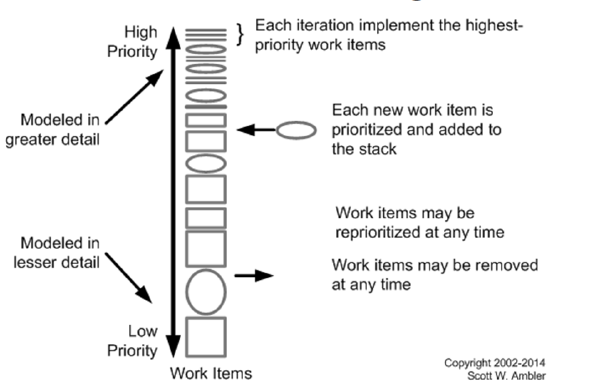
Como **<nombre del rol>** yo puedo **<actividad>** de forma tal que **<valor de negocio que recibo>**.

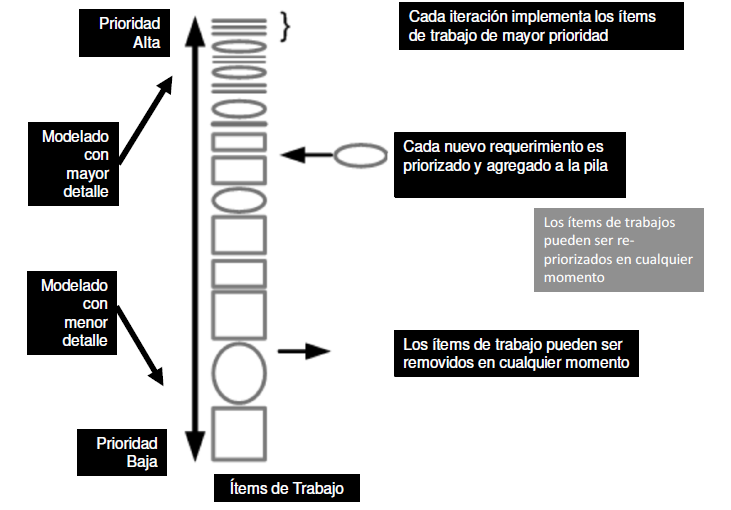
Donde **nombre de rol** representa quién está realizando la acción o quién recibe el valor de la actividad; **actividad** representa la acción que realizará el sistema; y **valor de negocio que recibo** comunica porqué es necesaria la actividad.

Las User Stories son **Multipropósito**: Las historias son:

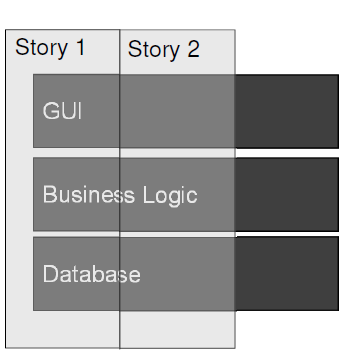
* Una necesidad del usuario.
* Una descripción del producto.
* Un ítem de planificación.
* Token para una conversación.
* Mecanismo para diferir una conversación.

**Product Bachelor:** El Product Owner prioriza las historias en este bachelor. Cada una tiene una granularidad (inversamente proporcional al detalle). Si la historia ya cuenta con una granularidad fina, entonces se puede arrancar a desarrollar.





A las User Stories se las debe considerar como porciones. Esto se demuestra con la siguiente imagen:



Es decir, no tiene sentido que primero se desarrolle horizontalmente TODA la Database, o el Business Logic, ya que, si hacemos esto, no tenemos nada para presentarle al cliente con **valor de negocio**, a pesar que a nivel de programación esté bueno el hecho de contar con algo tan importante ya desarrollado. Por esto, se tiene que trabajar a las User Stories como porciones **verticales**.

Para todo esto, se tiene que tener en cuenta quién va a usar el producto 🡪 Se pueden describir roles de usuario (es decir, una clase con distintas instancias de esta clase – objetos, lo cual tiene una abstracción muy alta). En contrapartida, se puede usar la técnica denominada “**Personas**” (es un objeto). A partir de una Persona se obtiene una experiencia del usuario al que representa a esta Persona, y al hacer análisis de varias, se puede desarrollar una UX agradable que satisfaga los requerimientos al nivel más global posible para todos los usuarios.

**Técnicas Adicionales:**

* Personajes Extremos: Personas en términos extremos que podrían usar el producto, para tenerlos en cuenta a nivel general en cuanto a características. Por ejemplo, considerar el que el producto va a ser utilizado desde el Papa, hasta un narcotraficante.
* Usuarios Representantes (Proxies): Cuando el Product Owner no tiene tiempo, por lo que designa este cargo a otra persona de la empresa (por lo general son personajes del lado del negocio – que no sirven – porque no tienen mucha idea desde el punto de vista de sistemas, o porque pueden verse sesgados por priorizar el área de la empresa donde ellos trabajan y, por ende, el proyecto fracasará).

Criterios de Aceptación de las User Stories.

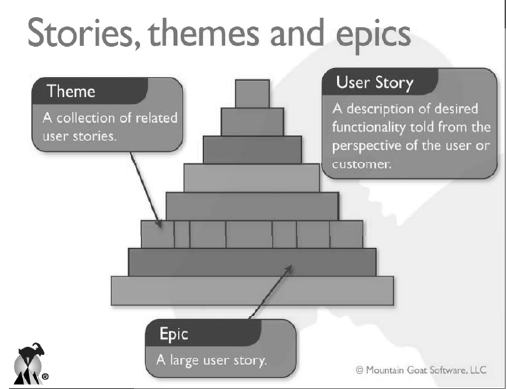
Acuerdo respecto a cómo se tiene que comportar el software. Se desarrollan estos criterios para que el Product Owner lo acepte (definido objetivamente). Para comprobar estos criterios es que se hacen las pruebas de aceptación (que contienen el conocimiento del negocio). Aunque algo a tener en cuenta, es que la mayoría de las veces se desea achicar el número de pruebas, y muchas veces es por motivos económicos para realizarlas. Un ejemplo de criterio puede ser: La nota debe ser un número entero entre 1 y 10 para un examen. Estos criterios se pueden agregar como pie en las tarjetas, y pueden representar tanto requerimientos funcionales, como no funcionales.

**Definition of Ready:** La User Story está apta, por lo que del Product Bachelor pasa a iteración de desarrollo.

**Definition of Done:** Si la User está terminada (está en condiciones de mostrarse al Product Owner – con testing, documentación incluida, y todo lo demás). Esta definición de Done es un checklist, y es una construcción colectiva, es decir, de todo el equipo.

Aspectos adicionales de las User Stories:

* 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.
* Necesita poco o nulo mantenimiento, y puede descartarse después de la implementación.
* Junto con el código, sirven de entrada a la documentación que se desarrolla incrementalmente después.
* Tiene diferentes niveles de abstracción:



Estos niveles de abstracción dependen de la granularidad de las User Stories (de acuerdo al tamaño, para ver si entra a una iteración o no, para ver si está terminada o no – 0 o 1 (gestión binaria -).

Si una User Story cumple con la Definition of Ready, se desarrolla, y a su vez, si es muy grande para una iteración sola, se la cataloga como **Epic** (aunque siempre considerando que las estimaciones son relativas en cuanto a tamaño/tiempo).

En caso de ser una colección de User Stories relacionadas, se habla de **Theme**.

# INVEST Model.

El INVEST Model se utiliza para poder determinar el estado del Definition of Ready. Si la Story cumple con este modelo, entonces se agrega a la iteración para la producción.

* **Independent:** Calendarizables e implementables en cualquier orden. Independiente desde el punto de vista que cada User Story elegida por el Product Owner no tenga dependencia con otra.
* **Negotiable:** El “qué”, no el “cómo”, es decir, el cliente expresa lo que necesita, pero no cómo se implementa, esto es trabajo nuestro, aunque obviamente debemos ponernos de acuerdo previamente con el usuario.
* **Valuable:** Que aporte valor de negocio, debe tener valor para el cliente.
* **Estimatable:** Para ayudar al cliente a armar un ranking basado en costos. Cantidad de información de la User Story que tengo para imaginar el tamaño (para estimarlo).

SPIKE: Tanta incertidumbre que la tengo que investigar y transformar en una User Story. Es una técnica funcional (cómo lo implemento – información del negocio que no tengo, o que es incompleta).

* **Small:** Si es muy pequeña, no da valor al cliente, además si la agrego a la iteración, tengo que esperar a la próxima para que realmente se vea el valor de esta Story, y esta demora no cumple con el principio ágil de releases tempranos y frecuentes.
* **Testable:** Si puedo demostrar que cumplí con expectativas del cliente (con las pruebas de aceptación ya escritas). Demostrar que fueron implementadas.

**Spike:** Tipo especial de historia, utilizado para quitar riesgo e incertidumbre de una User Story u otra faceta del proyecto. Se clasifican en:

1. Técnicas: Utilizadas para investigar enfoques técnicos en el dominio de la solución.
   * Evaluar performance potencial.
   * Decisión hacer o comprar.
   * Evaluar la implementación de cierta tecnología.

Cualquier situación en la que el equipo necesite una comprensión más fiable antes de comprometerse a una nueva funcionalidad en un tiempo fijo.

1. Funcionales: Utilizadas cuando hay cierta incertidumbre respecto de cómo el usuario interactuará con el sistema.

Usualmente son mejor evaluadas con prototipos para obtener realimentación de los usuarios o involucrados.

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 cómo el sistema debe resolver la interacción con el usuario para alcanzar el beneficio esperado.

Algunas User Stories requieren de ambos tipos de spikes (técnicas y funcionales). 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:

1. **Spike Técnico:** 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.
2. **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 spikes:

* Estimables, demostrables, y aceptables.
* La excepción, no la regla:
  + Toda historia tiene incertidumbre y riesgos.
  + El objetivo del equipo es aprender y resolver cierta incertidumbre en cada iteración.
  + Los spikes deben dejarse para incógnitas más críticas y grandes.
  + Utilizar spikes como última opción.
* 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.

Algunas cosas para dejar en claro.

* Diferir el análisis detallado tan tarde como sea posible, lo que es justo antes de que el trabajo comience.
* Hasta entonces, se captura requerimientos en la forma de “User Story”, que son descripciones breves de funcionalidad relevante para el cliente.
* Las User Stories no son requerimientos; son marcadores para conversaciones más detalladas y análisis que deberán ocurrir conforma esas historias vayan implementándose.
* Por lo tanto, no necesitan ser descripciones exhaustivas de la funcionalidad del sistema, solo la suficiente información para que los desarrolladores y los clientes tengan una comprensión común.

# Estimaciones.

Una **estimación** es una predicción que tiene como objetivo predecir la completitud y administrar los riesgos. Se relaciona con los objetivos del negocio, compromisos y control.

Errores en las estimaciones.

* Información imprecisa acerca del software a estimar, o acerca de la capacidad para realizar el proyecto.
* Demasiado caos en el proyecto (mal definido el proyecto).
* Imprecisión generada por el proceso de estimación.
* Una de las fuentes de errores más común es omitir actividades necesarias para la estimación del proyecto tales como, por ejemplo, requerimientos faltantes, licencias, reuniones, revisiones, etc.

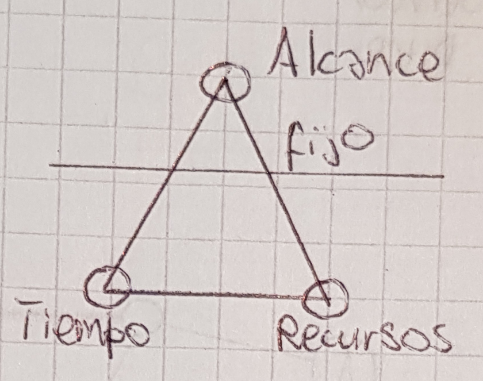
Consideraciones.

* Momentos apropiados para estimar:
  + Al inicio del proyecto.
  + Luego de la especificación de requerimientos.
  + Luego del diseño
* Una de las actividades más complejas en el desarrollo de software, luego de la definición del mismo.
* Por definición, una estimación no es precia. La mayor cantidad de veces se producen equivocaciones al estimar.
* Existe un universo de probabilidades asociado a las estimaciones.
* Estimar no es planear, y planear no es estimar.
* Las estimaciones son al base de los planes, pero los planes no tienen que ser lo mismo que lo estimado.
* A mayor diferencia entre lo estimado y lo planeado, mayor es el riesgo.
* Las estimaciones no son compromisos.

# Estimaciones Ágiles.

Primeramente, recordando el tema de estimación, pero a nivel tradicional, y que este tipo de estimación hace referencia a estimaciones **absolutas**, se dividían en 4 puntos, con el siguiente orden:

1. **Tamaño (¿Qué?)**.
2. **Esfuerzo** 🡪 Medido en horas/persona lineales **(¿Cómo?)**. Esto es algo muy personal (individual).
3. **Tiempo (¿Cuándo?)**. Ya en esta instancia se trabaja en equipo, y se definen cosas tales como por ejemplo cuántas personas van a trabajar, qué tareas se pueden desarrollar en paralelo, el propio tiempo para el desarrollo, etc.
4. **Costo (¿Cuánto?)**.



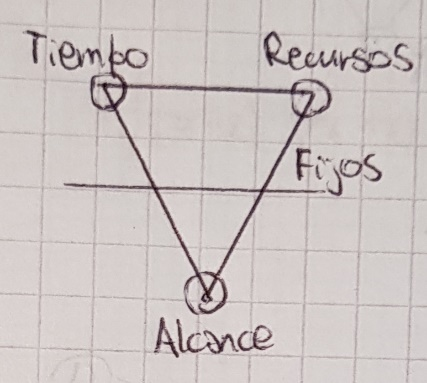
Pasando ahora a Agile:

* En los equipos Agile, las Features/Stories son estimadas usando una medida de tamaño relativo, conocida como **Story Points (SP)**, que hace referencia a qué tan compleja es la característica para poder construirla, y también la incertidumbre que tiene la misma (la falta de información asociada).

Otra definición de Story Point: Unidad de medida específica (del equipo) de complejidad, riesgo y esfuerzo, es lo que “el kilo” a la unidad de nuestro sistema de medición de peso. Da una idea del “peso de cada Story, y decide cuán grande (compleja) es. La complejidad de una Feature/Story tiende a incrementarse exponencialmente.

Las medidas son relativas, no son absolutas, esto quiere decir que se hacen comparaciones (comparo complejidades de las distintas User Stories, las cuales determinan las propias características del producto).

* + Las personas no saben estimar en términos absolutos.
  + Somos buenos comparando cosas.
  + Compara es, generalmente, más rápido.
  + Se obtiene una mejor dinámica grupal y pensamiento de equipo más que individual.
  + Se emplea mejor el tiempo de análisis de las Stories.
* Story Points no es una medida basada en tiempo.



En Agile se hace referencia a cuánto me puedo comprometer (ya que el alcance es lo variable, mientras que el tiempo y los recursos están fijos).

De esta forma, quedan definidas tres variables/dimensiones para determinar el tamaño de una User Story a partir de los Story Points: **Complejidad, Duda y Esfuerzo**.

Se trata que las estimaciones sean un tema tratado por el propio equipo que se va a encargar de desarrollar la misma, ya que, por ejemplo, como se suele hacer en algunos casos, si se deja esta tarea a un “experto”, es difícil que lo que éste “experto” diga, sea lo que en realidad termina sucediendo a la hora de desarrollar la característica del producto, ya que las variables son demasiadas, y una sola persona no puede ser experta en tantos ámbitos, por lo que se generaría desviaciones.

Haciendo una acotación sobre las spikes, mencionadas anteriormente, una User Story se transforma en una Spike cuando la incertidumbre es mucha, y ya en este punto significa que no se puede estimar, por lo que, por el momento, no puede cumplir con el criterio de listo (Definition of Done).

Siguiendo con las definidas “3 dimensiones”, éstas se pueden combinar de distintas maneras, y eso va a depender del tamaño de la User Story en cuestión, así como también de la escala que se utilice, citando, por ejemplo, la escala de los talles de remera (escala previamente acordada por el equipo, para que, luego, a partir de la misma, ya se puedan comparar las demás User Stories y determinar el tamaño de cada una).

Otras escalas para medir el tamaño pueden ser:

* Tamaño por números: 1 a 10.
* Talles de remeras: S, M, L, XL, XXL.
* Serie : 1, 2, 4, 8, 16, 32, 64, etc.
* Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, etc.

Y una vez elegida la escala, ésta **no se cambia**, porque si se hace, se cambia el metro patrón.

Considerando que el tamaño es una medida de la cantidad de trabajo necesaria para producir una Feature/Story, y que el mismo indica:

* Cuán compleja es la Feature/Story.
* Cuánto trabajo es requerido para hacer o completar una Feature/Story.
* 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.**

Velocidad.

La **velocidad** se la define como una medida (métrica) del progreso de un equipo. Se calcula sumando el número de Story Points (asignados a cada User Story) que el equipo **completa** durante la **iteración**.

Por esta razón, al ver cuántas User Stories meto en una iteración, tengo que ver cuánto pude implementar (software funcionando que le entrega valor de negocio al Product Owner). De esto surge que esto es un asunto binario (o el software está completo, o no está completo), ya que, si no está terminado, no cuenta, por lo que vuelve al Product Backlog; en caso que lo esté, se tiene en cuenta para los Story Points de la iteración.

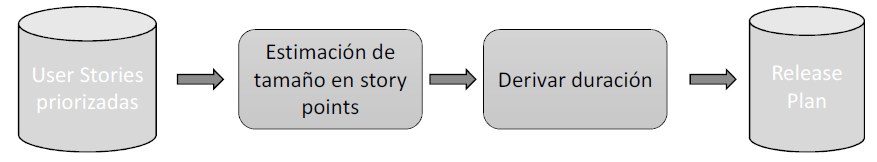
La velocidad corrige los errores de estimación.

Esta medida sirve para estimar de acuerdo a las velocidades de las iteraciones ya hechas. Así, puedo estimar, aproximadamente, cuántas características de producto puedo llegar a implementar en una iteración a futuro, pero SIEMPRE hablando del mismo equipo, en el mismo proyecto, ya que la experiencia no es extrapolable a otros proyectos.

¿Y cómo hago un proyecto?

Si se estima User Stories. ¿Cómo se hace para estimar un proyecto?

* La duración de un proyecto no se “estima”, se deriva tomando el número total de Story Points de las User Stories, y dividiéndolo por la velocidad del equipo.



* La velocidad ayuda a determinar un horizonte de planificación apropiado.
* La estimación en Story Points separa completamente la estimación de esfuerzo de la estimación de duración.

**Estimación de Duración:** Cuántos sprints necesito para un release. Así se estima un proyecto, y lo que mido es el **nivel de certeza**, y **NO el nivel de precisión**.

Poker Estimation.

* Es un método de estimación popular entre los Agile Practicioners, publicado por Mike Cohn.
* Combina opinión de experto, analogía y desegregación.
* Participantes en “Planning Poker” son desarrolladores.
  + “Las personas más competentes en resolver una tarea, deben ser quienes la estiman”.
* Mejor grupal que individual (son muchos factores, y eso hace difícil que una sola persona sea experta en tantos ámbitos).
* “Metro” usado es Fibonacci. Las User Stories de 13 ya son algo complicado para tratar en una sola iteración, por lo que se las trata de partir.
* Se elige una User Story **canónica** (para poder comparar con el resto). Por lo general, se elije una que tenga un peso = 1. Estas comparaciones son relativas, y no se debería cambiar la canónica, porque cambia todo lo ya hecho.
* Time Boxing: Tiempo fijo para todas las actividades (el tiempo es una variable fija en Agile).
* Uno estima por vez, la cantidad de User Story que considera que se van a terminar para el sprint. (por lo general, la menor cantidad posible).

El método consiste en que cada persona, de forma individual y privada, estima cada una de las User Stories, y luego se hace una puesta en común entre todos los miembros del equipo (escuchando justificaciones) para tratar de llegar a una situación en la que estén todos de acuerdo, para poder definir el State of Ready de las mismas.

¿Cómo “decodificar” las estimaciones?

* **0:** Quizás usted no tenga idea de su producto o funcionalidad en este punto.
* **½, 1:** Funcionalidad pequeña (usualmente cosmética).
* **2-3:** Funcionalidad pequeña a mediana. Es lo que se quiere.
* **5:** Funcionalidad media. Es lo que se quiere.
* **8:** Funcionalidad grande, aunque de todas formas se la puede hacer, pero hay que preguntarse si no se puede partir o dividir en algo más pequeño No es lo mejor, pero sigue estand Ok.
* **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:** Confirmación de que algo está muy mal. Mejor ni arrancar.

Gestión de Proyectos.

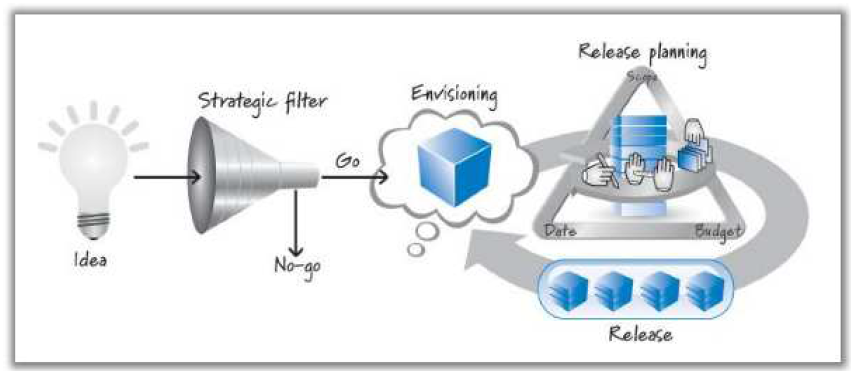
¿Por qué se crean productos?: Hay que entregar valor de negocio al cliente, así como satisfacer las necesidades del mismo, tener muchos usuarios logueados, obtener dinero, realizar una gran visión y cambiar el mundo, entre muchos otros objetivos.

* **Funcionalidad:** Que el producto haga lo que tiene que hacer.
* **Confiabilidad:** El software me da resultados, a partir de los cuales me puedo basar en el futuro.
* **Usabilidad:** Sentirse bien usando el producto. Que me ayude a hacer lo que debería ayudarme a hacer el software.

**Evolución de los productos de software** (focalizado en experiencias – gente, actividades, contexto).

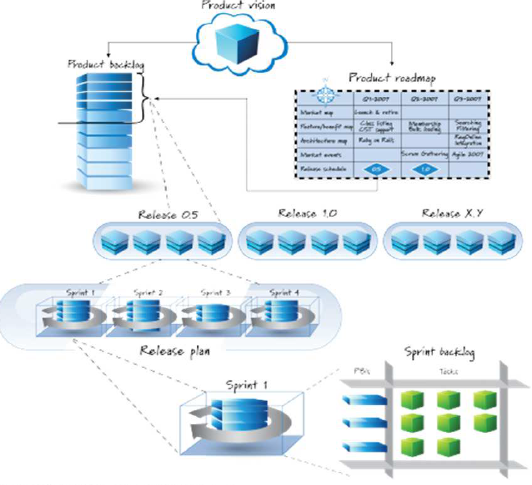
****

**Creación de Productos:** Arranca por 1 idea. De ahí se pasa a un Filtro Estratégico para ver si el producto va a ser viable o no (filtro no tecnológico al día de hoy).



Luego se pasa a una visión (a dónde quiero llegar con el producto), lo cual es el disparador para la creación – **Road Map** – Cómo pienso alcanzar el producto que quiero hacer 🡪 De ahí se deriva el Product Backlog.

Para el Road Map se tienen distintos “quarters”, los cuales conforman un release en un tiempo máximo de 1 mes.



**Release Plan:** Cuántas User Stories en cada sprint para el release.

**Sprint Backlog:** Con tareas (en horas ideales – considerando distracciones) para cada User Story, para cumplir con el Definition of Done, para sacarla del tablero, y pasarla a Done.

Tengo que elegir qué da valor (qué características), ya que, si no lo hace, se lo considera Desperdicio. Tengo que averiguar cuál es la cosa correcta a construir 🡪 **Productividad**. Sobre todo esto se aplica para el MVP (Producto Mínimo Viable – primera release).

**MVP (Minimal Viable Product):**

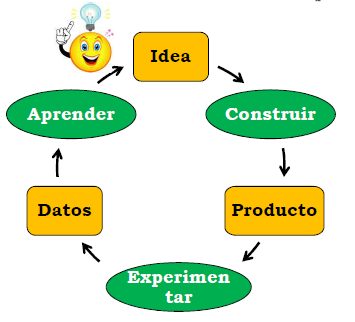
* Es un concepto de Lean Startup que enfatiza el impacto del aprendizaje en el desarrollo de nuevos productos.
* Una premisa clave detrás de la idea de MVP es que se produce un producto real que se puede ofrecer a clientes, y así observar su comportamiento real con el producto/servicio.
* Otra definición (según 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". Este es un aprendizaje validado que viene en forma de si los clientes realmente van a comprar el producto en cuestión.
* Se utiliza para ver lo que la gente realmente hace con respecto a un producto, lo cual es mucho más confiable que directamente preguntarle a la gente qué harían. Tiene como propósito la realimentación, es decir, ver si el producto es correcto o no, lo más rápido posible, ver si el producto tiene valor, en contrapartida al propósito de la ganancia de dinero.
* El desafío en esto es que hay que escuchar al cliente para derivar sus necesidades, a través del MVP.
* 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.

**MVP vs MMF o MMP (Errores Comunes):**



Teniendo como premisa que lo que se quiere lograr es siempre la creación de valor, hay que tener en cuenta que el éxito no es entregar un producto, si no que 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 éstos últimos.

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



La fase **Construir** – 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.



* En esta fase hay que decidir qué tan complejo va a ser el MVP de manera formulada:
  + Requiere juicio.
  + En caso de duda, simplificar.
  + Evitar la construcción excesiva y la promesa excesiva.
  + Cualquier trabajo adicional más allá de lo que se necesita para comenzar el ciclo, podría ser un desperdicio.
* Un MVP no solo habla sobre el diseño del producto y las preguntas técnicas, sino que también sirve para probar hipótesis comerciales fundamentales.
* Por lo tanto, sirve para proporcionar una dosis necesaria de realidad.

**Dilema – La Audacia de Cero:** A menudo es más fácil recaudar dinero cuando se tienen cero ingresos, cero clientes y tracción cero, que cuando se tiene una pequeña cantidad de cada uno de estos elementos.

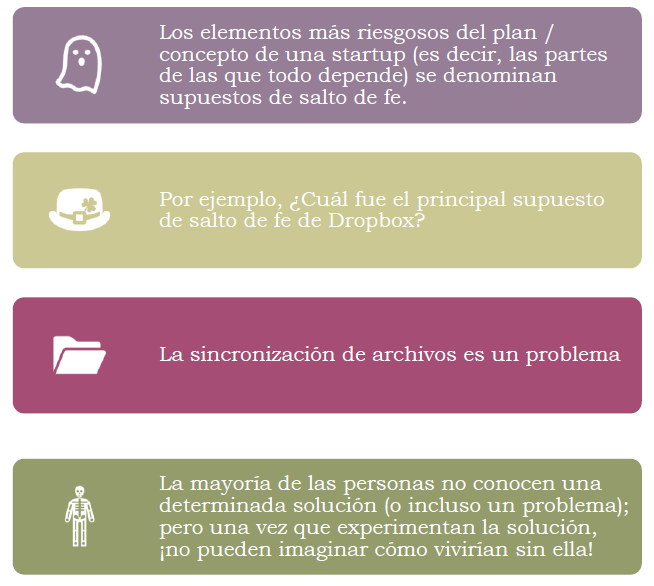
Cero invita a la imaginación, pero los números pequeños hacen preguntas sobre si los números grandes alguna vez se materializarán. Este fenómeno (audacia de cero) crea un incentivo brutal: Aplazar el lanzamiento de cualquier versión de un producto hasta que se esté seguro del éxito.

Si se pospone experimentar con un MVP, van a surgir algunos resultados desafortunados como:

* La cantidad de trabajo desperdiciado puede aumentar.
* Se van a perder los comentarios esenciales.
* El riesgo de que el startup construya algo que nadie quiera puede aumentar.

En cuanto a las compensaciones. ¿Se preferiría atraer capital de riesgo y potencialmente derrocharlo? ¿O se preferiría atraer capital de riesgo y utilizarlo sabiamente? Por ello, se debe usar un MVP para experimentar (inicialmente, en silencio) con los primeros usuarios en el mercado. Luego se verifica el concepto probando **todos** los elementos, comenzando por los más riesgosos.

**Supuestos de “Saltos de Fe”:**



Para estos supuestos se tiene:

* **Hipótesis de Valor:**
  + Prueba si el producto realmente está entregando valor a los clientes después de que comienzan a usarlo.
  + Una métrica de prueba: tasa de retención.
* **Hipótesis de Crecimiento:**
  + Prueba cómo nuevos clientes descubrirán el producto.
  + Una métrica de prueba: tasa de referencia o Net Promoter Score (NPS).

Por último, referido a un MVP, para prepararlo:



**Trazabilidad:** Permite rastrear errores generando más consistencia. Si modifico un archivo en un lugar, puedo ver qué “rompería” o afectaría dicho cambio.

**Gestión de Configuración:** Gestión que hace frente a cambios para establecer y mantener la integridad de un producto de software. Tiene 4 actividades fundamentales:

1. **Identificación de los Ítems de Configuración:** Actividad operativa. Se hace para garantizar la integridad de un ítem, y para poder rastrearlo (para lo cual, primero debo identificarlo – a partir de la definición de su nomenclatura).
2. **Control de Cambios:** Actividad operativa. En este punto entra el concepto de **Línea Base**, el cual se desarrolla más adelante, pero a modo de resumen, es como una “foto” de 1 o más ítems de configuración en un momento del tiempo que han sido revisados, evaluados y aprobados, y que, para ser modificados, tienen que pasar por un procedimiento de control de cambios. Esta línea se la considera estable, es decir que cuenta con el visto bueno de todo el equipo. Existen 2 tipos:
   1. Operacionales: Contiene una versión del producto con código ejecutable.
   2. De Especificación: Sin código.

Un concepto relacionado al control de cambios es el de **Comité de Control de Cambios**, el cual, en caso que haya cambios en la línea base, se reúne para su modificación (o creación previa), y cuenta con una cantidad reducida de personas que sea necesario que estén al tanto de los cambios, para que éstos luego informen a las demás.

1. **Informe y Reporte de estados:** Actividad para dar visibilidad. Existen informes de inventario (como mínimo), así como también informes de quién cambió qué y cuándo (árbol de versiones – historial), y de líneas base en sí mismas.
2. **Realizar Auditorías de Configuración:** Para control y para definir desviaciones. Una auditoría es una revisión objetiva (no depende de criterios) e independiente. Cuando se desarrolla, se lo hace para garantizar la calidad e integridad (es decir, prevenir errores lo antes posible para que esto resulte más barato) una vez que el software ya está funcionando. Existen 3 tipos de auditorías:
   1. Auditoría de Configuración Física: Se asegura que los ítems de configuración estén, y que lo hagan donde estableció que iban a estar.
   2. Auditoría de Configuración Funcional: Se lleva a cabo una vez que se desarrollo la auditoría de configuración física, y se realiza para **validación**, es decir, que el software haga lo que tiene que hacer.
   3. Auditoría de Proyecto: Se analiza el proyecto para ver si cumple con los procesos, o no.

Las auditorías se hacen sobre una determinada línea base, y no sobre todo el repositorio.

Las situaciones por las cuales se generan cambios en el software pueden ser variadas, pero algunos ejemplos tienen su origen en:

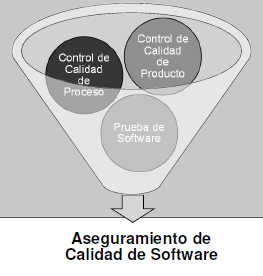
* Cambios del negocio y nuevos requerimientos.
* Soporte de cambios de productos asociados.
* Reorganización de las prioridades de la empresa por crecimiento.
* Cambios en el presupuesto.
* Defectos encontrados al corregir.
* Oportunidades de mejora.

**SCM (Software Configuration Management):** Una de las primeras cosas que se tienen en cuenta cuando se define un proyecto. Es una disciplina transversal (actividad que abarca a todo el proyecto con aplicación de diferentes disciplinas, y no puedo ubicarla en un momento temporal determinado del proyecto) dentro de la ingeniería de software cuyo propósito es mantener la integridad de un producto de software desde que se concibe su idea, hasta que sale del mercado (es decir, durante todo su ciclo de vida).

Administra ítems de configuración (los mantiene para tener una referencia de cada uno), los cuales no son solo piezas de código, sino que es **TODO.**

Otra definición (ANSI/IEEE 828, 1990): 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, y verificar correspondencia con los requerimientos.

**Disciplinas de Soporte del Software:**



**Pero. ¿Por qué se debería gestionar la configuración?** El 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.

**Producto Íntegro:** Un producto íntegro es aquel que cumple con todos los requerimientos (tanto los funcionales como los no funcionales), lo que significa que satisface las necesidades del usuario. Si el mismo es íntegro, se puede mantener la trazabilidad de los elementos (puede ser fácil y completamente rastreado durante su ciclo de vida); y a su vez cumple con las expectativas de costo y criterios de performance.

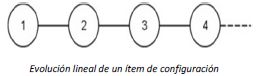
Mantener un producto íntegro no es tarea fácil, ya que hay que trabajar con componentes, y su manejo puede presentar algunos problemas tales como:

* Pérdida de un componente.
* Pérdida de cambios (el componente que se tiene no es el último).
* Sincronía fuente – objeto – ejecutable.
* Regresión de fallas.
* Doble mantenimiento.
* Superposición de cambios.
* Cambios no validados.

**Ítem de Configuración:** Todo y cada uno de los artefactos que forman parte del producto o del proyecto (el cual se necesita mantenerlo identificado), que pueden sufrir cambios (o no), o necesitar ser compartidos entre los miembros del equipo, y sobre los cuales se necesita conocer su estado y evolución.

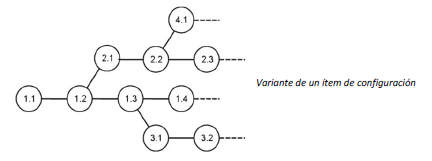
Cada conjunto de cambios sobre un ítem conforma una nueva **versión**:

* 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:



**Variante:**

* Una variante es una versión de un ítem de configuración (o de la configuración) que evoluciona por separado.
* Las variantes representan configuraciones alternativas.
* Un producto de software puede adoptar distintas formas (configuraciones) dependiendo del lugar donde se instale.
* Por ejemplo, dependiendo de la plataforma (máquina + SO) que al soporta, o de las funciones opcionales que haya de realizar o no.



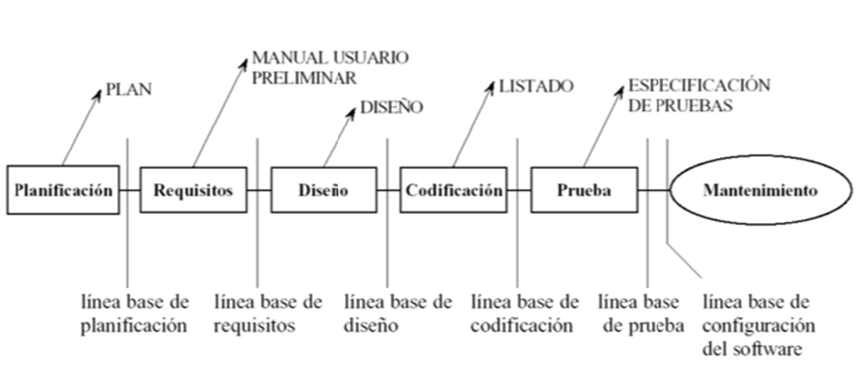
Existen 3 tipos de ítems:

1. **De Producto:** Por ejemplo, un archivo de arquitectura.
2. **De Proyecto:** Por ejemplo, el plan de proyecto.
3. **De Iteración.**

**Configuración:** Conjunto de ítems de configuración que conforman un producto de software en un momento determinado.

**Línea Base:**

* Se utilizan etiquetas para “marcar” las baselines.
* No se debe confundir a este concepto con el de versión del producto.
* Configuración marcada (ha sido revisada formalmente) sobre la que se ha llegado a un acuerdo.
* Sirve como base para desarrollos posteriores y puede cambiarse solo a través de un procedimiento formal de control de cambios.
* Se define por el equipo a partir de una configuración en particular, por tener interés sobre ésta.
* Permiten ir atrás en el tiempo y reproducir el entorno de desarrollo en un momento dado del proyecto.
* Con respecto a su representación, puede ser:
  + **De Especificiación** (Requerimientos, Diseño).
  + **De Productos** que han pasado por un control de calidad definido previamente.



**Calidad:** Conjunto de características subjetivas para satisfacer necesidades y expectativas. En el software es difícil de medirla, ya que el mismo es intangible.