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.

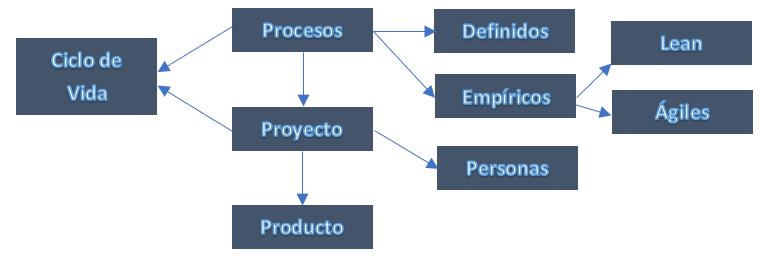
Por otro lado, este proceso puede presentar algunos problemas:

* 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.

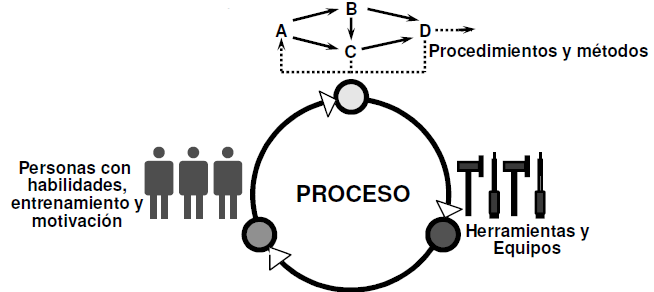
Se adapta

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.

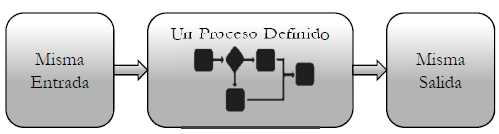
**Proceso de Software:** Un conjunto de actividades, métodos, prácticas, y transformaciones que la gente usa para desarrollar o mantener software y sus productos asociados.



**Tipos de Procesos:**

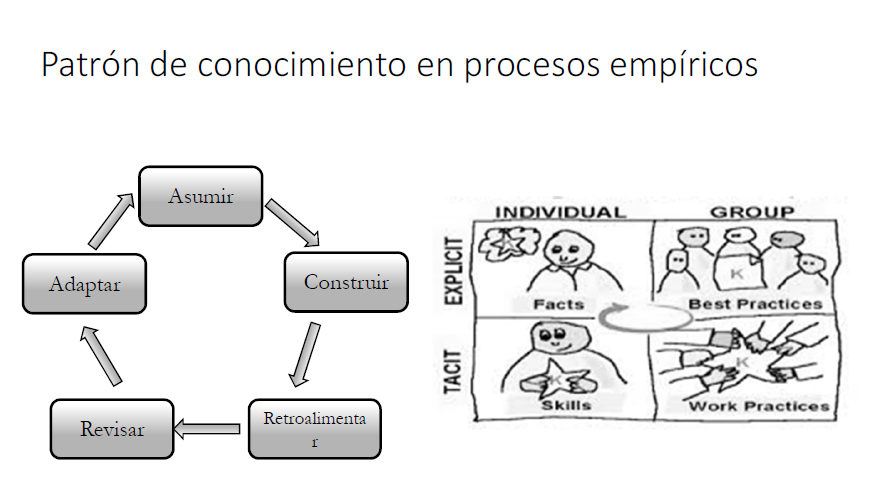
**Definidos:** Son solo los lineamientos, es decir, que todavía quedan cosas sin definir. Ejemplos: Cascada, PUD, RUD, Iconix, Métricas. Están inspirados en las líneas de producción:

* 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.

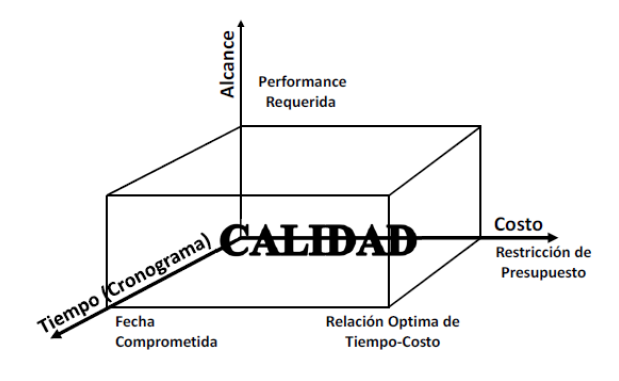
**Administración de proyectos:** “… tener el trabajo hecho …” en tiempo, con el presupuesto acordado y habiendo satisfecho las especificaciones o requerimientos; aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades del proyecto para satisfacer los requerimientos del mismo.

Para administrar un proyecto, se tiene que:

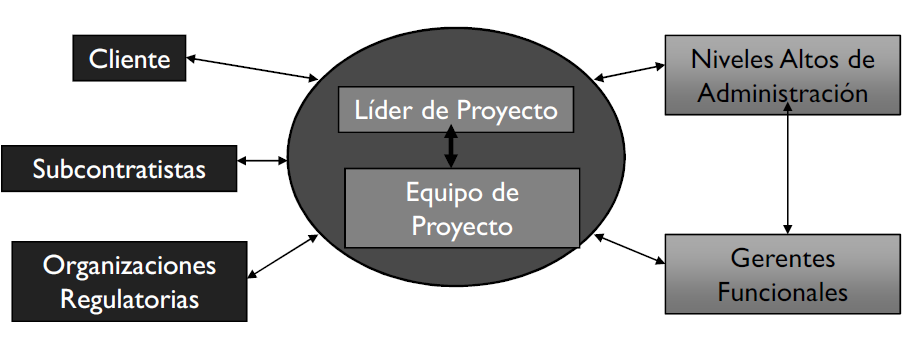
* Identificar los requerimientos.
* Establecer objetivos claros y alcanzables.
* Adaptar las especificaciones, planes, y el enfoque a los diferentes intereses de los involucrados (stakeholders).

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.

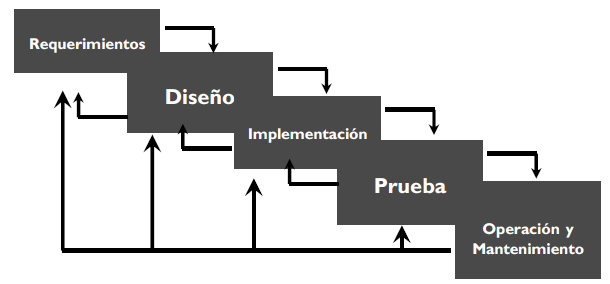
Es de suma importancia que este documento sea correctamente mantenido y actualizado de manera permanente, es decir, “No está escrito en piedra”, y tiene ciclos de cambios. El mismo funciona como un paraguas o caparazón para el equipo. Si el plan de proyecto no se actualiza, es muy probable que el proyecto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

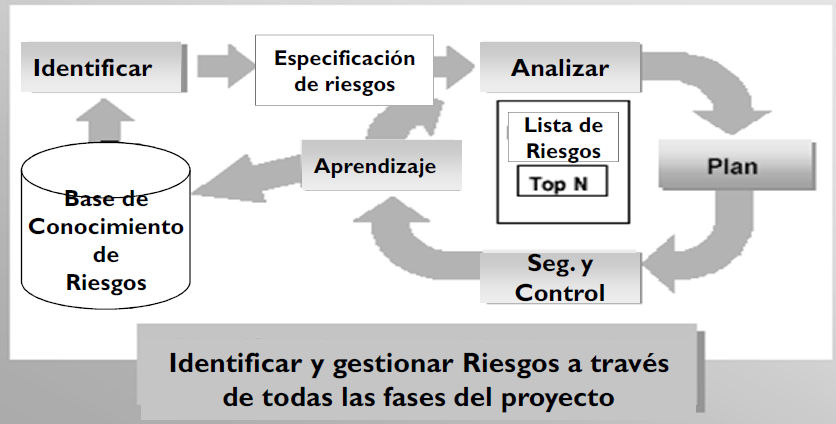
Para los riesgos, hay que tener estrategias **pro**activas (actuar con anticipación para mitigar dichos riesgos), y de ahí un plan de contingencia. En contrapartida, tener medidas **re**activas no es tan eficiente, ya que el suceso ya pasó.

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.

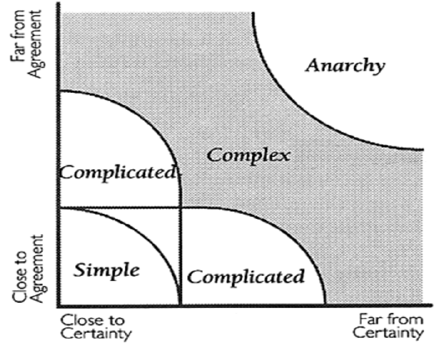
¿Qué significa Ágil?

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.

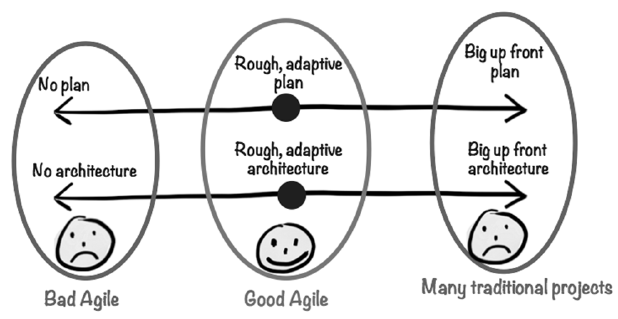
Las metodologías ágiles le dan más foco al producto, **priorizando** al cliente.

Por otro, lado. ¿Cuándo se aplica Agile?

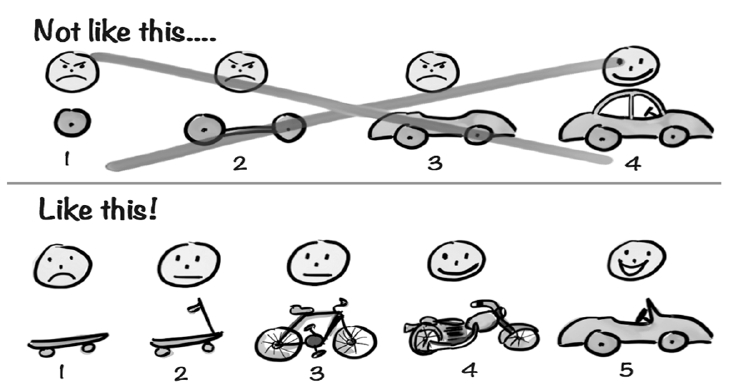


* Agile da mejores resultados cuando los problemas a ser resueltos caen dentro del espacio “Complex”.
* El desarrollo de nuevos productos y Knowledge Work tienden a estar en el espacio Complex.
* Investigación está dentro de Anarchy.
* Mantenimiento por lo general cae en Simple.

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.

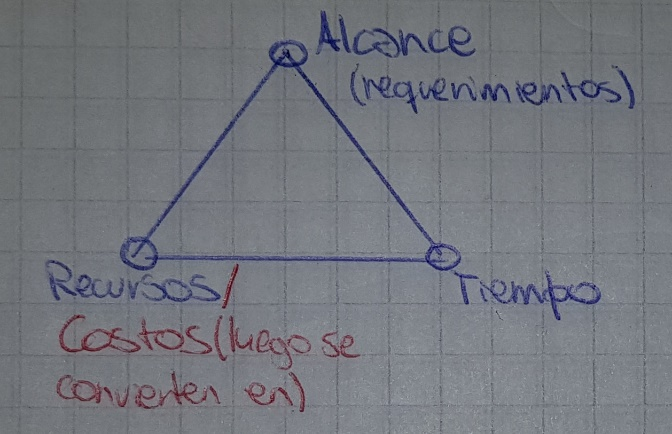
# User Stories.

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.

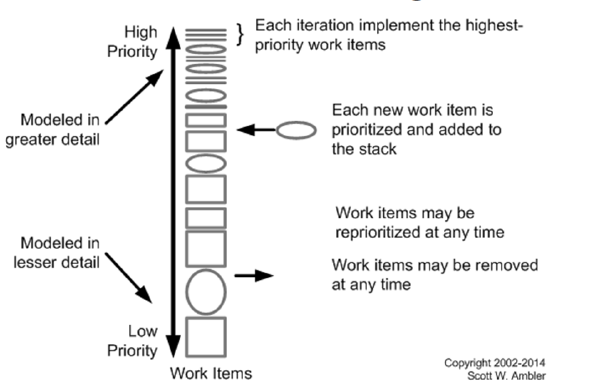
Como **<nombre del rol>** yo puedo **<actividad>** de forma tal que **<valor de negocio que recibo>**.

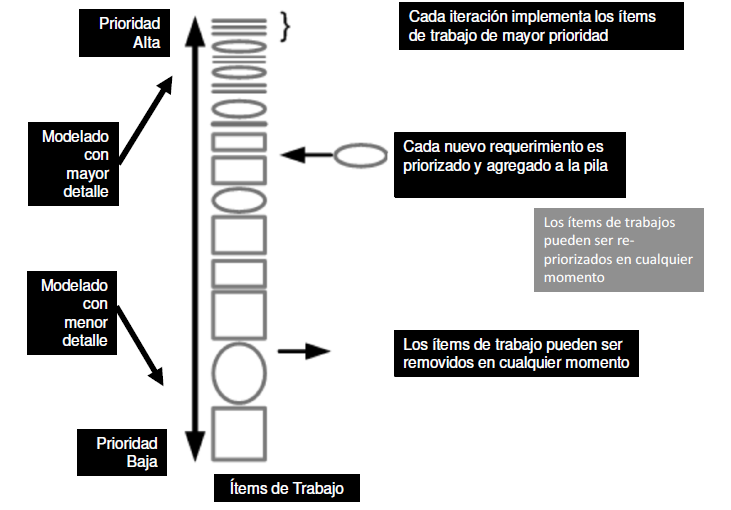
Donde **nombre de rol** representa quién está realizando la acción o quién recibe el valor de la actividad; **actividad** representa la acción que realizará el sistema; y **valor de negocio que recibo** comunica porqué es necesaria la actividad.

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

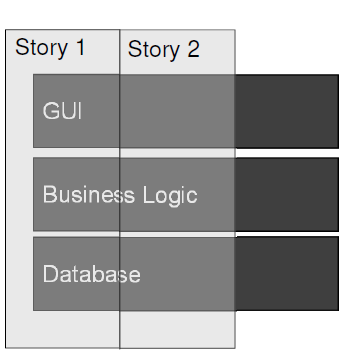
* Una necesidad del usuario.
* Una descripción del producto.
* Un ítem de planificación.
* Token para una conversación.
* Mecanismo para diferir una conversación.

**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.

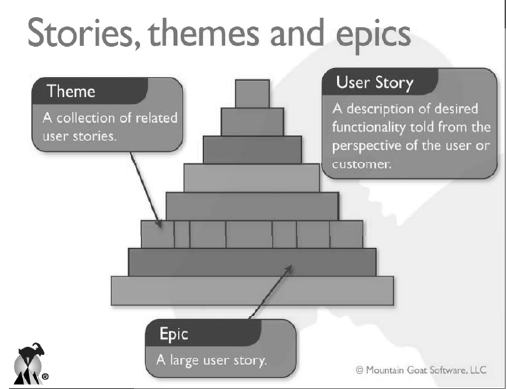
Acuerdo respecto a cómo se tiene que comportar el software. Se desarrollan estos criterios para que el Product Owner lo acepte (definido objetivamente). Para comprobar estos criterios es que se hacen las pruebas de aceptación (que contienen el conocimiento del negocio). Aunque algo a tener en cuenta, 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.

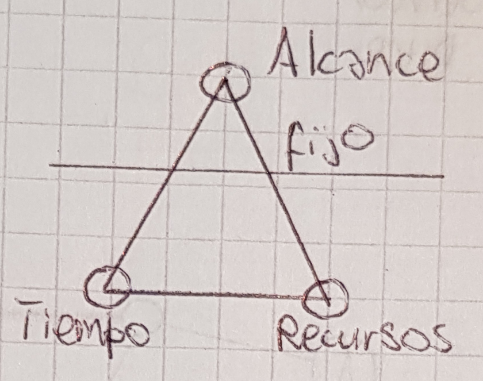
Consideraciones.

* 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?)**.



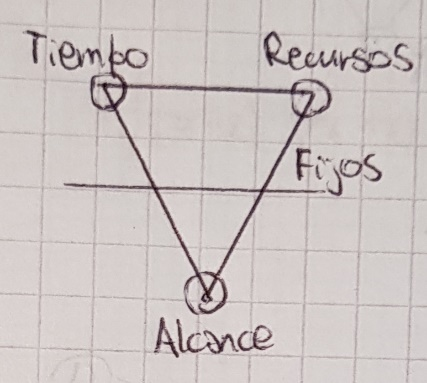
Pasando ahora a Agile:

* 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.

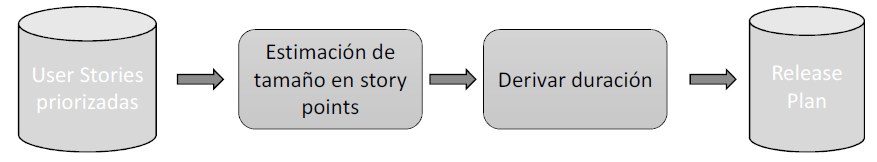
La velocidad corrige los errores de estimación.

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.

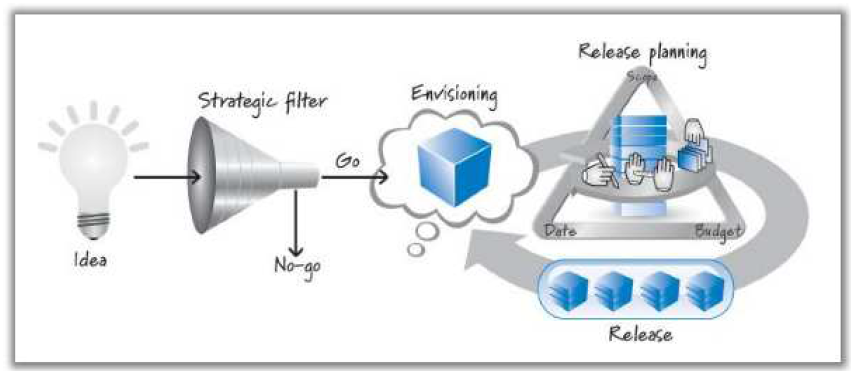
¿Por qué se crean productos?: Hay que entregar valor de negocio al cliente, así como satisfacer las necesidades del mismo, tener muchos usuarios logueados, obtener dinero, realizar una gran visión y cambiar el mundo, entre muchos otros objetivos.

* **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).

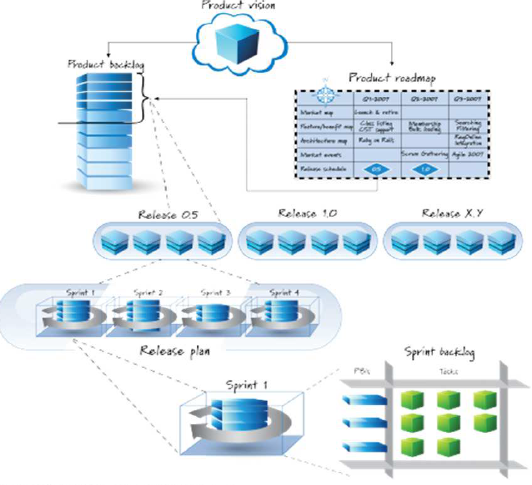
****

**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):**

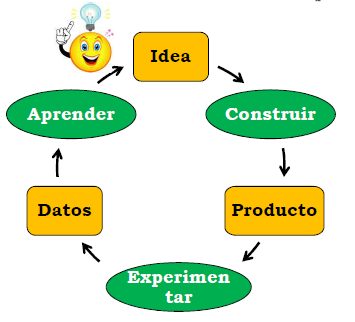
* Es un concepto de Lean Startup que enfatiza el impacto del aprendizaje en el desarrollo de nuevos productos.
* Una premisa clave detrás de la idea de MVP es que se produce un producto real que se puede ofrecer a clientes, y así observar su comportamiento real con el producto/servicio.
* Otra definición (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, sino 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.

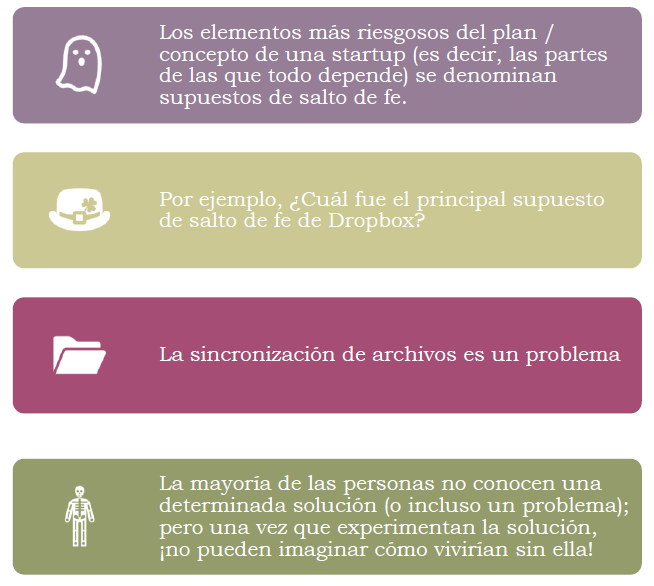
Cero invita a la imaginación, pero los números pequeños hacen preguntas sobre si los números grandes alguna vez se materializarán. Este fenómeno (audacia de cero) crea un incentivo brutal: Aplazar el lanzamiento de cualquier versión de un producto hasta que se esté seguro del éxito.

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 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**: Una vez que la línea base se conformó, no es posible cambiarla sin pasar por un proceso formal, llevado a cabo por este comité. La formalidad del proceso está dada por el hecho de que todos los involucrados se anoticien. Esta autoridad de cambio, al recibir una “propuesta de cambio”, lleva adelante un análisis de impacto de cambio, donde se evalúa el esfuerzo técnico, efectos secundarios, impacto global sobre otras funciones y sobre otros objetos, y se le asigna una prioridad para que posteriormente se realice lo que se conoce como revisión de partes. En base a todo este análisis, el comité acepta o rechaza el cambio, y notifica a todas las partes involucradas.

* + Involucrados directos que están en el proyecto.
  + Si el cambio se origina en el cliente, se lo incluye.
  + La convocatoria se realiza para analizar el cambio sobre una línea base, y no un ítem en particular.

1. **Informes y Reportes de Estados:** Actividad para dar visibilidad. Los informes nos dicen el estado actual de la configuración del software. Existen informes de inventario (como mínimo – contiene una copia del contenido del repositorio), así como también informes de quién cambió qué y cuándo (árbol de versiones – historial), y de líneas base en sí mismas. Los objetivos de los informes son:
   1. Mantener los registros de la evolución del sistema. Incluye reportes de rastreabilidad de todos los cambios realizados a las líneas base durante el ciclo de vida, es decir, dice cuáles son las líneas base, cuándo se modificaron, quién modificó cada cosa.
   2. Manejan mucha información y salidas, por lo que se suele implementar dentro de procesos automáticos.
   3. Este informe es útil desde el punto de vista administrativo para realizar auditorías.
2. **Realizar Auditorías de Configuración:** Para control y para definir desviaciones. Una auditoría es una revisión objetiva (no depende de criterios) e independiente que tiene como objetivo asegurar que lo que está indicado en cada ítem de configuración de software en la línea base, o actualización, se ha alcanzado realmente, y que el software y la documentación son internamente consistentes para entregarlos al cliente. Cuando se desarrolla, se lo hace para:
   * **Validación – Construir el producto correcto:** Asegurar que el problema se ha resuelto de la manera apropiada, de tal manera de permitir que el usuario obtenga el producto correcto.
   * **Verificación – Construir el producto correctamente:** Asegurar que el producto cumple con los objetivos definidos en la documentación de líneas base. Todas las funciones son llevadas a cabo con éxito, y los tests cases tengan status “ok”, o bien consten como “problemas reportados” en la nota de release.

Entre sus funciones se encuentra:

* + Determinar la semejanza entre el estado actual del sistema, y el establecido como línea base.
  + Proveer el mecanismo para establecer una línea base.
  + Transición desde:
    - Línea base a establecer (en etapas formativas).
    - Línea base sancionada.

Existen 3 tipos de auditorías:

* 1. Auditoría de Configuración Física: Se asegura que los ítems de configuración estén, y que lo hagan donde se estableció que iban a estar. Asegura que lo que está indicado para cada ítem de configuración de software en la línea base, o actualización, se ha alcanzado realmente, y que el software y la documentación son internamente consistentes para entregarlos al cliente. Se realiza para **verificar**.
  2. Auditoría de Configuración Funcional: Se lleva a cabo una vez que se desarrolló la auditoría de configuración física, y se realiza para **validación**, es decir, que el software haga lo que tiene que hacer. Es la evaluación independiente de los productos de software, verificando que la funcionalidad y rendimiento reales de cada ítem sean consistentes con la especificación del requerimiento.
  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.

**Beneficios de las auditorías:**

* Aumenta la protección contra cambios innecesarios.
* Mejora de la visibilidad del estado del proyecto y sus componentes.
* Aumenta la auto responsabilidad.
* Disminuye los costos por re-trabajos.
* Disminuye el tiempo de desarrollo.
* Aumenta la calidad.
* Suministra visibilidad y rastreabilidad del ciclo de vida del producto de software.

**Desventajas:**

* Quejas.
  + Es burocrático.
  + Es molesto.
  + Se meten con mi trabajo.
* La transición es difícil.
* No hay compromiso en todos los niveles.
* No hay conciencia del problema.

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.

**Actividades relacionadas con la Gestión de Configuración:**

* Administración del cambio: Hacer seguimiento de las peticiones de cambios al software por parte de los clientes y desarrolladores, estimar costos, y el efecto de realizar dichos cambios.
* Gestión de versiones: Seguimiento de versiones de los distintos componentes del sistema, y garantizar que los cambios hechos no interfieren entre sí.
* Construcción del sistema: Ensamblar componentes del programa, datos y librerías, compilarlos y vincularlos para generar un ejecutable.
* Gestión de entregas (release): Preparar el software para la entrega externa, hacer un seguimiento de las versiones del sistema que se entregaron al cliente.

**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.

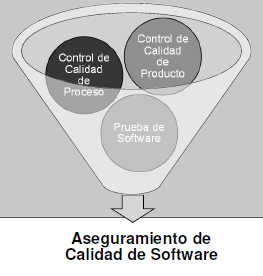
**Planificación de SCM:** Un plan describe los estándares y procedimientos utilizados para la gestión de la configuración. El plan de gestión de configuraciones contiene:

* Tipos de documentos.
* Esquemas de nombrado.
* Estructura de repositorio.
* Línea Base y sus responsables.
* Responsables de creación de procedimientos.
* Registros que deben mantenerse.
* Herramientas y el proceso para usarlas.
* Procesos de Auditoría, ejecución y registro.
* SCM para software externo (opcional).
* Cómo se hará el control de cambio, y miembros del comité de control de cambios.

Consideraciones:

* Debe hacerse de forma temprana.
* Se deben definir los documentos que se administrarán.
* Todos los productos del proceso deben administrase.

**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:

* Satisface las expectativas del usuario: Considera los requerimientos funcionales.
* Puede ser fácilmente rastreado durante su ciclo de vida: Se puede saber cómo llego a cada versión de cada pieza de software, es decir, saber a qué requerimiento está asociado, y saber dónde se analizó, dónde se diseñó, dónde se testeó, etc.
* Satisface criterios de performance: Satisface requerimientos no funcionales (performance, por ejemplo).
* Cumple expectativas de costo: Para que esto sea posible, es necesario planificar, por lo cual, se evalúa la relación costo-beneficio.

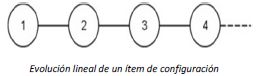
Mantener un producto íntegro no es tarea fácil, ya que hay que trabajar con componentes, y su manejo puede presentar problemas. El software está en cambio constante, y esos cambios pueden ser de tipo internos o externos. Algunos de ellos pueden ser:

* 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**:

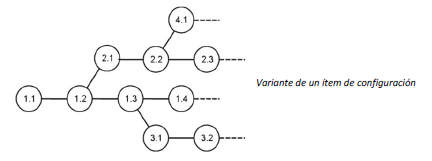
* Instancia de un ítem de configuración que difiere de otras instancias del mismo ítem. El **control de versiones** combina procedimientos y herramientas para gestionar las versiones de los objetos de configuración creados durante el proceso de software.
* Una versión se define, desde el punto de vista 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:



* La gestión de configuración permite a un usuario especificar configuraciones alternativas del sistema de software mediante la selección de las versiones adecuadas. Esto se puede gestionar asociando atributos a cada versión (que pueden ser datos sencillos como un número de versión asociado a cada objeto). Cada versión de software es una colección de elementos de configuración, como ser código fuente, documentos, datos, etc.

**Variante:**

* Si la diferencia que existe entre las instancias de un mismo ítem de configuración es muy pequeña, no se denomina versión, sino que se la conoce como **variante**.
* 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 la soporta, o de las funciones opcionales que haya de realizar o no.



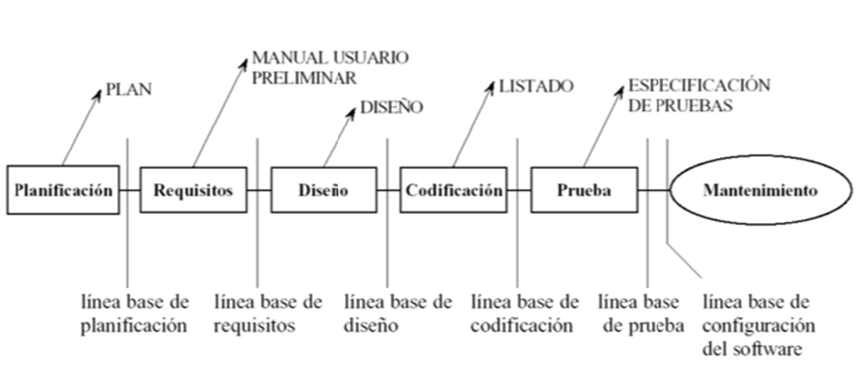
Existen 3 tipos de ítems:

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:** Especificación o producto que se ha revisado formalmente, sobre el cual se ha llegado a un acuerdo y que, de ahí en adelante, sirve como base para un desarrollo posterior, y que puede cambiarse solamente a través de procedimientos formales de control de cambio.

* Se identifican con etiquetas, las que permiten encontrar sus elementos a partir de referencias.
* Se almacenan en un repositorio.
* 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.
* 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.
* Permite repetitividad, permite entregar siempre lo mismo.
* No se modifican ni eliminan, ya que se pierde trazabilidad.
* Se definen:
  + Cuando termina un sprint (ágil).
  + Cuando termina cada fase (tradicional).
* Con respecto a sus tipos, puede ser:
  + **De Especificación** o **Líneas Base de Fin de Fase**: En ellas se definen modelos (Requerimientos, Diseño). Generalmente se definen antes que se tenga código.
  + **Operacionales**: Contiene el producto entregado al cliente, es decir, el producto ya ha pasado por un control de calidad definido.



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

Frameworks para escalar SCRUM.

Un framework es una herramienta de gestión de proyectos (de cualquier cosa) para organizar equipos, de forma ágil. Inicialmente se crearon para el desarrollo de software, pero actualmente este horizonte está ampliado a cualquier ámbito.

Tiene como foco la creación/gestión de productos (y no está tan guiado por los conceptos de línea de producción).

Para trabajar con un framework, hay que considerar como requisito que el Product Owner tiene que venir del negocio (la presencia de éste es el elemento diferenciador con respecto a las metodologías tradicionales), y es el que toma las decisiones (él y sólo él). Tiene que haber **UN** solo Product Owner, que puede representar a todos dentro de una empresa, y es el que toma las decisiones (incluidas las económicas), por lo que es el dueño del MVP.

Por lo general, en cuanto a los equipos de trabajo, se recomienda que tenga un tamaño de integrantes (más de 9 ya no, y también debe considerarse que el PO **NO** está incluido en este cálculo).

El Scrum Master es el que asume el liderazgo de servicio (no es lo mismo que el Líder de Proyecto o que el Analista Funcional), y es el que busca el consenso.

Con respecto a las iteraciones, éstas tienen una **duración** fija dentro de SCRUM (Time-Boxing), por lo que cuando termina el tiempo, termina la iteración, considerando que los requerimientos son variables.

Por otro lado, en caso que no tengan la duración fija, si no que tengan el **alcance** fijo, se está hablando de, por ejemplo, el PUD (iteraciones incrementales), en donde se tiene un conjunto de requerimientos para cada iteración. Al tener el alcance fijo, tengo el tiempo variable, por lo que, si no termino lo establecido, tengo que extender este último.

Ceremonias.

La primera ceremonia es al **Spring Planning**, el cual tiene como resultado al **Spring Backlog**. En esta se analiza el concepto de la Capacidad (Story Points u horas hábiles por Sprint), así como también la Velocidad (la cual se calcula, y no se estima, y se determina por qué aceptó – cuántos Story Points – el PO, lo que se traduce en cuántos SP se quemaron en el Sprint).

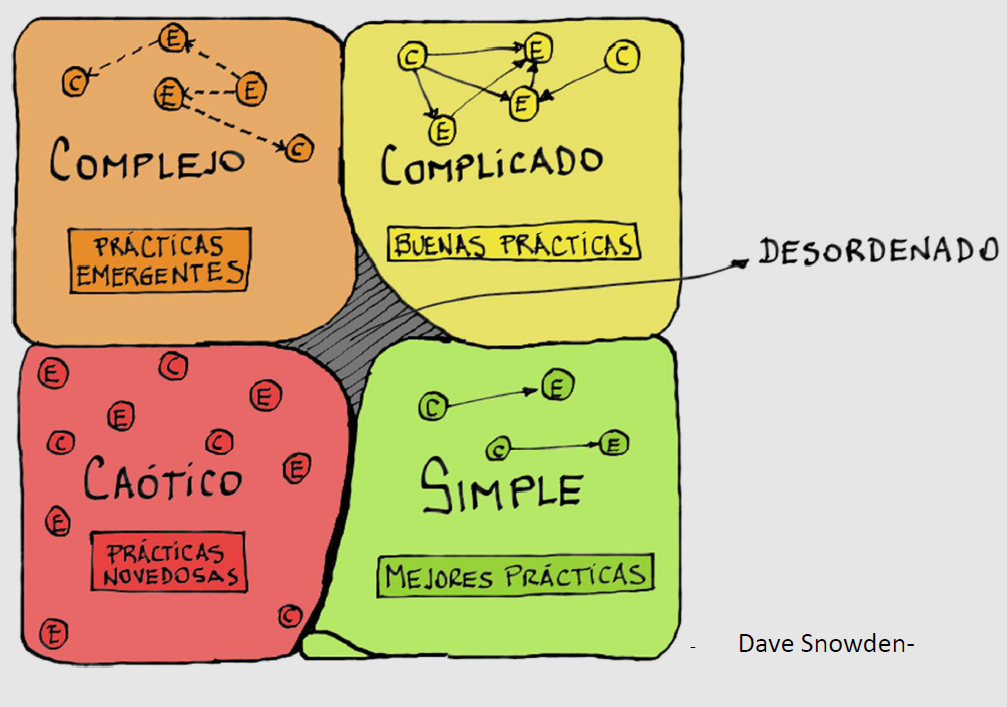
En esta ceremonia se establecen las Tareas (desintegración de cada una de las US medidas en horas ideales) para el **Sprint Backlog** (lo cual no hay en el **Product Backlog**, ya que acá hay US, épicas o themes, es decir, productos), para las cuales, para determinar si las cumplo o no, me fijo en el concepto de **Definition of Done**.

Otra ceremonia es la de **Inspección** del Producto (**Sprint Review**), y la **Inspección** del Proyecto (**Sprint Retrospective**). Para está última, así como en las **Dailys**, el PO **NO** participa, y son actividades en las que se toma como referencia al Product Backlog, el cual sí es exclusivo del PO (considerando que este NO puede cambiar el Sprint Backlog, lo que significa que durante el Sprint no hay cambios, razón por las cuales éstos son cortos).

Por otra parte, el Scrum Master, así como también el Scrum Team, participan en todas las ceremonias.

A la hora de hablar de las problemáticas de complejidad que pueden surgir en el desarrollo de un proyecto, a éstas se les puede hacer frente con el framework denominado **Cynefin**, el que presenta 4 escenarios, y sirve para tomar decisiones en cada uno de ellos (se lee en sentido horario):

* **Simple:** Se pueden distinguir las relaciones Causa/Efecto, es decir, está todo controlado, por lo que se aplican las Mejores Prácticas.
* **Complicado:** Existe 1 Causa, pero no necesariamente 1 solo Efecto asociado a ésta. Aparece el concepto de probabilidad, con el consecuente análisis de riesgo, por lo que se aplican las Buenas Prácticas.
* **Complejo:** Existen “n” Efectos, sin poder relacionarlos con las Causas. En este punto se encuentra el software, y es donde hay que aplicar creatividad. Un dato es que SCRUM se autodefine muy bueno para este tipo de situaciones. En este caso se aplican Prácticas Emergentes.
* **Caótico:** En este escenario ya no hay, o no se pueden, o no se encuentran las relaciones entre las Causas y los Efectos. Se aplican Prácticas Novedosas.



# Nexus.

Entonces. ¿Qué es lo que se hace? ¿Proyecto o Producto? Se hacen Productos, y se utilizan a los Proyectos para hacerlos. Para esto se utilizan Frameworks.

Nexus es un framework integrado que utiliza SCRUM para escalar SCRUM. Sus elementos son:

* **Nexus Integration Team (NIT):** Compuesto por: PO + SM + 10 + NIT Members. Es igual que un equipo SCRUM.

Los miembros del NIT pueden ser miembros de equipos Scrum individuales y, si es así, tiene prioridad su pertenencia al NIT. Agregando la consideración que es un framework con límite, por lo que deben ser entre 3 y 9 equipos Scrum

La composición del NIT puede cambiar en el tiempo.

Existe un único Product Backlog 🡪 1 Producto.

Las responsabilidades son:

* + Declaración del DoD aplicable al incremento integrado.
  + Coaching sobre estándares de desarrollo, arquitectura e infraestructura.
  + Consultoría.
  + Generación de advertencias sobre dependencias y problemas entre equipos.
  + Puede trabajar en el Product Backlog.
  + Se realiza 1 Daily por equipo **Y** 1 Daily Nexus, en donde asiste un representante de cada equipo Nexus (se hacen distintas preguntas en cada tipo de Daily).

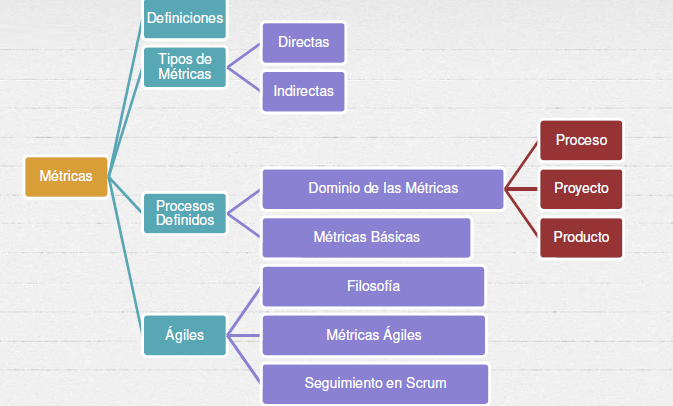
En cuanto a sus eventos:

* **Nexus Daily Scrum:** Se enfoca en el impacto de cada equipo sobre el incremento integrado, y discuten:
  + ¿El trabajo del día anterior fue integrado exitosamente? Si no fue así. ¿Por qué no?
  + ¿Cuáles nuevas dependencias han sido identificadas?
  + ¿Qué información necesita compartirse entre los equipos Nexus?
* **Retrospectiva del Sprint Nexus:** Consta de tres partes:
  + Representantes de todo un Nexus identifican problemas que hayan impactado en más de un quipo 🡪 **Transparentear problemas compartidos**.
  + Cada equipo Scrum realiza su propia retrospectiva, y aborda lo siguiente:
    - ¿Se dejó algún trabajo sin realizar? ¿El Nexus generó deuda técnica?
    - ¿Todos los artefactos se integraron con éxito, al menos 1 vez al día?
    - ¿El software se desplegó con frecuencia suficiente para prevenir la acumulación abrumadora de dependencias sin resolver?
  + Representantes apropiados de equipos Scrum se reúnen y acuerdan cómo visualizar y rastrear las acciones identificadas.

Por último, haciendo referencia a sus artefactos:

* **Product Backlog:** Que como se mencionó antes, es 1 solo-
* **Nexus Sprint Backlog.**
* **Incremento Integrado.**

# Métricas



Como primer asunto, hay que considerar el motivo por el cual se mide: Se mide para cumplir con un objetivo, para obtener información a partir de la cual tomar una decisión, para satisfacer una necesidad como, por ejemplo:

* De informar.
* De motivar.
* De comparar.  
  De entender.
* De evaluar.
* De predecir.
* De mejorar.

**Medida:** Proporciona una indicación cuantitativa de la cantidad, dimensiones, o tamaño de algunos atributos de un producto o proceso.

**Medición:** Acto de determinar una medida.

**Métrica:** Es una medida del grado en que un sistema, componente o proceso posee un atributo dado, en un ámbito determinado (proceso, proyecto o producto). Es un número (ya que es algo cuantitativo), que se encuentra sobre una base objetiva.

**Indicador:** Es una métrica, o una combinación de métricas, que proporcionan una visión profunda del proceso del software, del proyecto de software, o del producto en sí. Es la interpretación de una métrica.

Dentro del ámbito de los procesos definidos, se tiene que existen dos tipos de métricas:

**Métrica Directa:** Una métrica de la cual se pueden realizar mediciones sin depender de ninguna otra métrica, y cuya forma de medir es un método de medición. Se obtiene de forma simple, sin utilización de algoritmos ni métodos complejos.

* La forma de medir una métrica directa es un método de medición.
* Una métrica directa puede ser utilizada en funciones de cálculo.

Ejemplos:

* Líneas de código fuente escritas.
* Horas-programador diarias.
* Costo por hora-programador, en unidades monetarias.

**Métrica Indirecta:** Una métrica cuya forma de medir es una función de cálculo, es decir, las mediciones de dicha métrica utilizan las medidas obtenidas en mediciones de otras métricas directas o indirectas. Tiene trabajo de cálculo más complejo.

* La forma de medir una métrica indirecta es una función de cálculo.
* Una métrica indirecta puede usarse en una función de cálculo.

Ejemplos:

* Líneas de código fuente por hora de programador.
* Costo total actual del proyecto, en unidades monetarias.
* Costo por línea de código fuente.

Analizando los factores:

* Tamaño (a nivel producto).
* Esfuerzo (a nivel proyecto) 🡪 Medido en horas-persona lineales **(MÉTRICA DIRECTA)**.
* Tiempo (a nivel proyecto) 🡪 Calendario
* Costo (a nivel producto) 🡪 Se deriva del esfuerzo **(MÉTRICA INDIRECTA)**.

**Escala:** Un conjunto de valores con propiedades definidas.

* Toda escala es de un cierto “Tipo de Escala”.

Ejemplos: Nominal, Ordinal, Intervalo, Ratio y Absoluta.

* Los valores que puede tomar la métrica “lenguaje de programación usado en un proyecto”: Pascal, C, Java (Nominal).
* El nivel de madurez CMM: 1, 2, 3, 4, 5 (Ordinal).
* El tamaño de un código software expresado en líneas de código: Conjunto de los números naturales (Ratio).
* La temperatura expresada en grados centígrados o grados Fahrenheit (Intervalo).

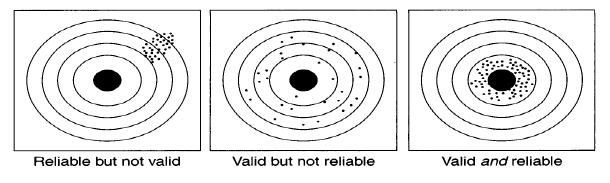
# Confiabilidad y Validez.

Estos dos conceptos muestran la calidad de la métrica.

**Confiabilidad:** Que la métrica no tenga errores. Es la contraparte de precisión.

**Validez:** Grado en el cual la métrica refleja el significado real del concepto que se observa. Si me sirve o no para alcanzar el objetivo planteado. Es la contraparte de exactitud.

* Métricas confiables no son necesariamente válidas, y viceversa:



**Métricas de Software:** La aplicación continua de mediciones basadas en técnicas para el proceso de desarrollo de software y sus productos, para suministrar información relevante a tiempo, así el líder de proyecto, junto con el empleo de estas técnicas mejorará el proceso y sus productos. Estas métricas tienen 3 dominios:

* **De Proceso:** Despersonalizada. Métricas Estratégicas. Tienen un alto nivel de impacto, porque su alcance es muy amplio. Son lineamientos generales de alto nivel. Métricas organizacionales que buscan mejorar; y son públicas (se arman consolidando métricas privadas). Ejemplos:
  + Eficiencia de la remoción de defectos.
  + Desviación organizacional de estimaciones.
  + Defectos por severidad en los productos de la organización.
* **De Proyecto:** Personalizadas. Métricas Tácticas. La información que obtengo en el momento con la intención de mejorar. Tienen un alcance bien concreto y específico del proyecto en cuestión. Sirven para lo que dura el proyecto, y so privadas (los resultados son solo visibles para los involucrados en el proyecto). Ejemplos:
  + Costos estimados vs reales.
  + Esfuerzo por etapa.
  + Eficiencia de estimaciones del proyecto.
  + Desviación de calendario.
* **De Producto:** Miden características directas de un producto de software. Generalmente, hacen referencia a tamaño, defectos, calidad (requerimientos). Son privadas (ya que se construyen en contexto de un proyecto). Ejemplos:
  + Líneas de código.
  + Defectos por severidad.
  + Cantidad de casos de uso por complejidad.

# Indicadores.

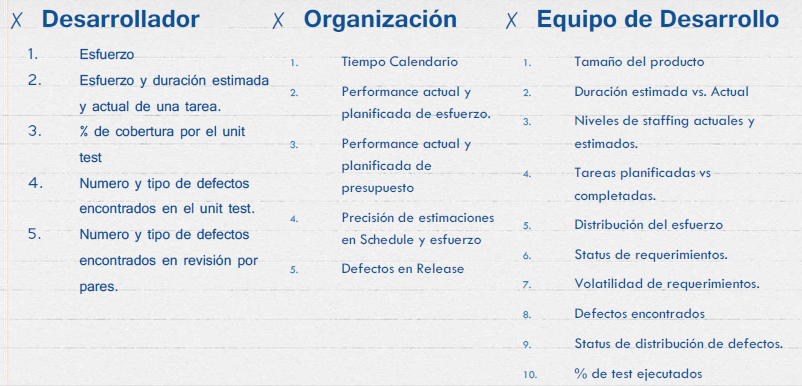
Los **Indicadores de Proceso** son **ESTRATÉGICOS** y permiten:

* Tener una visión profunda de la eficacia de un proceso existente.
* Que los gestores evalúen lo que funciona y lo que no.

Los **Indicadores de Proyecto** son **TÁCTICOS** y permiten:

* Evaluar el estado del proyecto en curso.
* Seguir la pista de los riesgos potenciales.
* Detectar las áreas de problemas antes de que se conviertan en “críticas”.
* Ajustar el flujo y las tareas del trabajo.
* Evaluar la habilidad del equipo del proyecto en controlar la calidad de los productos de trabajo de la ingeniería del software.

Métricas adecuadas al nivel adecuado:



Algunas consideraciones:

* Las métricas no resuelven problemas. Las personas los resuelven.
* Las métricas proveen información para que las personas puedan tomar las mejores decisiones.
* Las métricas no son el problema. Las métricas dan visibilidad a los problemas.
* Hay que ver si se van a usar las métricas o no, ya que la precisión es cara, y por ende, sacarlas también.
* **Métrica Estimación:** Puedo usar métricas para ayudarme con las estimaciones, pero las métricas tienen asociado un valor de probabilidad, y son una proyección; mientras que las estimaciones son hechos consumados, que sirven para dar visibilidad de una acción que está pasando para tomar una decisión.
* Feedback 🡪 Definir para quién es cada métrica, para luego comunicarle los resultados de la misma.
* Nunca hay que “pegar” los resultados/desempeño de una métrica a los castigos 🡪 Dan lugar a métricas inventadas.
* Nunca hay que usar solo una métrica.
* No hay que ignorar los datos.
* No hay que medir individuos.

# Métricas Ágiles.

La medición es una salida, no una actividad. Hay que medir lo que sea necesario y nada más. Hay dos principios ágiles que guían la elección de las métricas:

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

Esto hace referencia a la **Velocidad** (métrica ágil por excelencia), es decir, los SP que el PO acepta por ser software funcionando.

**Capacidad:** Una estimación de cuánto trabajo puede completarse en un periodo de tiempo dado. Basado en la cantidad de tiempo ideal disponible del equipo. Mide el compromiso de trabajo de un equipo para un sprint, y se puede medir en esfuerzo (horas), o en SP.

**Velocidad:** Una observación empírica de la capacidad del equipo para completar el trabajo por iteración. Cuántos SP fueron aceptados por el PO al final de un Sprint. La velocidad no es una estimación, ni tampoco un objetivo a alcanzar.

La misma está basada en el tamaño de los ítems de trabajo propio de cada equipo, y **no** está basada en estimaciones o tiempo real, así como tampoco determinada o impuesta por nadie más que por los miembros del equipo.

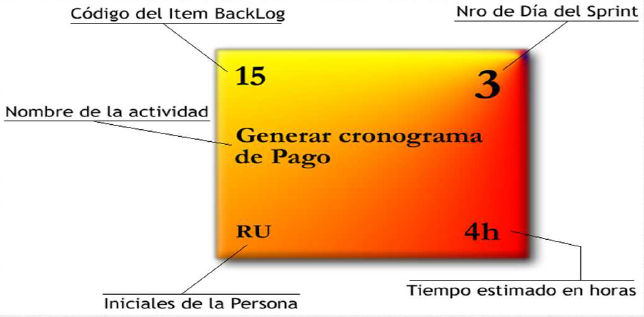
La velocidad es comparable entre iteraciones para **un equipo dado** y **un proyecto dado**, y no es comparable entre equipos, ni comparable entre proyectos, es decir, no es extrapolable.

Mediciones problemáticas – No relevantes con los métodos ágiles:

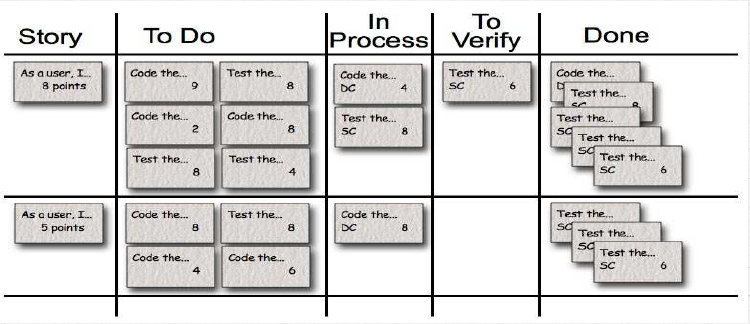
* Diagrama de Gantt.
* Porcentaje completado.
* Tiempo de cada miembro del equipo por tarea.
* Tiempo Real vs Tiempo Estimado.

# Herramientas de SCRUM.

Tarjeta de Tarea (Task):



Taskboard (Tablero):



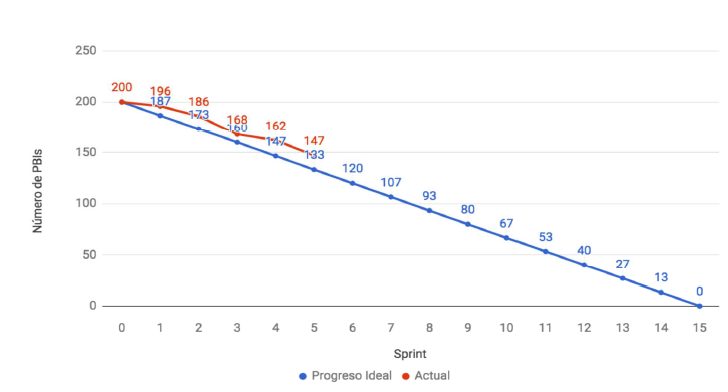
El tablero de SCRUM sirve para poder visualizar el trabajo.

Con respecto a los Burn-Down Charts, se hacen por Sprint, mientras que los Burn-Up Charts por proyecto. En las Dailys es cuando éstos se actualizan (ambos), mientras que en el Review es cuando calculo la velocidad. A toda esta información la introduzco en la Retrospectiva a modo de aprendizaje para el siguiente Sprint.

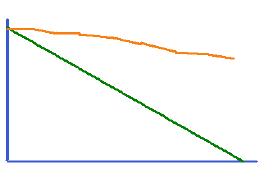
* Para los gráficos de Backlog se requiere:
  + Progreso del Sprint.
  + Progreso del Release.
  + Progreso del producto.
* El Backlog de trabajo es la cantidad de trabajo que queda por ser realizado.
* Tendencia del Backlog: Trabajo que queda vs Tiempo.



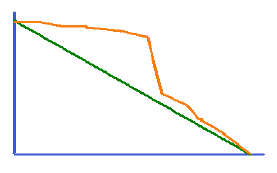
* **Burn-Down Chart:** ¿Cuánto trabajo queda para terminar?



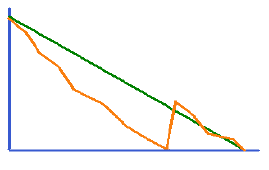
* + No todo lo incluido en el Sprint será entregado en la fecha propuesta. El Backlog no fue modificado para alcanzar la fecha (remover requerimientos o modificar la fecha.
  + Posibles causas:
    - Pobres técnicas de estimación.
    - Mal manejo de riesgos.
    - Rotación del equipo o renuncias
  + El gráfico siguiente muestra que el Scrum Master y el Product Owner no reaccionaron para evitar esta situación:



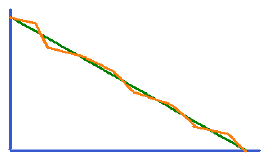
* + Hay que evitar llegar a este punto.
  + El gráfico siguiente es mejor que el anterior:



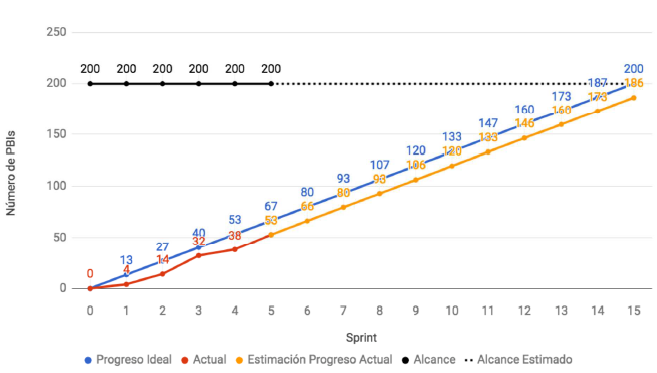
* + Al comienzo del Sprint hubo problemas, pero en el medio, el equipo de trabajo reaccionó y tomó medidas correctivas. Por la línea se nota que el PO quitó requerimientos para mantener la fecha de release.
  + También pone en evidencia un mejor manejo de riesgos, lo que facilitó aumentar la velocidad en la segunda parte del Sprint.



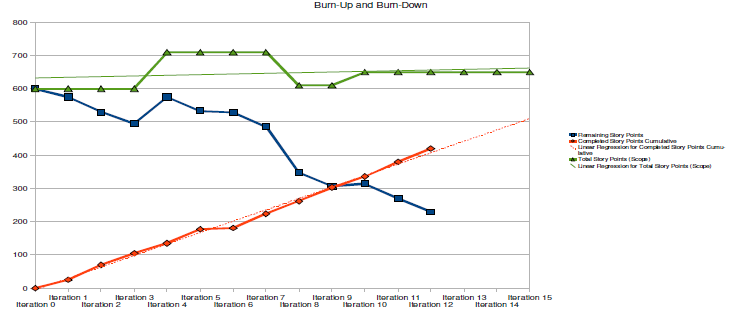
* + Este último gráfico también tiene características negativas.
  + No es un ejemplo de desarrolladores con las mejores habilidades técnicas.
  + Demuestra pobres técnicas de estimación.
  + El PO y el SM deberían haber agregado más requerimientos para ese release.



* + Ideal.
  + Buenas técnicas de estimación.
  + Buena velocidad.
  + Todos los requerimientos prometidos son los entregados.
  + Buen manejo de la asignación de tareas (tareas cortas, reporte diario).
  + Quizás… Demasiado bueno para ser verdad.
* **Burn-Up Chart:** ¿Cuánto trabajo se ha completado?



* **Combined Burn Chart:** ¿Cuánto trabajo se ha completado y cuánto trabajo queda?



Testing de Software.

El testing es una actividad que se realiza para asegurar la calidad, tanto del proceso, como del producto, pero para eso hay que tener en cuenta: **Asegurar la Calidad vs. Controlar la Calidad**:

* Una vez definidos los requerimientos de calidad, tengo que tener en cuenta que:
  + La calidad no puede “inyectarse” al final.
  + La calidad del producto depende de tareas realizadas durante **todo** el proceso.
  + Detectar errores en forma **temprana** ahorra esfuerzos, tiempo, recursos.
  + La calidad no solamente abarca aspectos del producto, sino también del proceso, y como éstos se pueden mejorar, esto se traduce en la posibilidad de evitar defectos recurrentes.
  + El testing **NO** puede asegurar ni calidad en el software, ni software de calidad.
  + Por lo mencionado en el punto anterior, más testing no garantiza una mejor calidad 🡪 . El testing solo evidencia lo que se hizo mal, es decir, identifica y advierte la presencia de defectos.

Desde el contexto de los Procesos Definidos, se asume que solo se pueden controlar los procesos, por lo que se hace aseguramiento de calidad del mismo y, como consecuencia, se logra el aseguramiento de calidad del producto. Es decir, en el contexto del **proyecto**, se hace control de aseguramiento del **proceso** (a través de auditorías); mientras que, para el lado de los **productos** en sí, además de las auditorías, se hace el control de calidad del testing comparando y determinando si lo que hace el software, es lo que efectivamente debe hacer (Validación).

* **Verificación:** ¿Estamos construyendo el sistema correcto? Por ejemplo, que el software acepte letras, cuando solo debe aceptar números.
* **Validación:** ¿Estamos construyendo el sistema correcto? Es decir, que el software HAGA LO QUE SE SUPONE QUE TENGA QUE HACER.

*Un aspecto a considerar es que la gestión de configuración provee un ambiente más sano para brindar calidad.*

**Testing:**

* Proceso **Destructivo** de tratar de encontrar defectos (cuya presencia se asume) en el código.
* Se debe ir con una actitud negativa para demostrar que algo es incorrecto.
* Testing exitoso 🡪 Aquel testing que encuentra defectos.
* Mundialmente: 30% a 50% del costo de un software confiable es representado por el testing (razón por la cual no se hace testing).

En cuanto a definiciones más formales se tiene:

* Aquel proceso que consiste en todas las actividades en el ciclo de vida, tanto estáticas como dinámicas, y que se refieren a la planeación, preparación y evaluación de los productos de software, y que están relacionados a productos de trabajo para determinar si satisfacen determinados requerimientos, para demostrar que se adaptan para los propósitos establecidos, y para detectar defectos.
* El proceso de operar un sistema o componente bajo ciertas condiciones específicas, observando o grabando los resultados, y haciendo una evaluación de algún aspecto del sistema o componente.
* El proceso de analizar un ítem de software para detectar las diferencias entre las condiciones existentes, y las condiciones requeridas (bugs), y evaluarlas.

Sin embargo, hay que tener en cuenta algunos aspectos que no son verdaderos sobre el testing (mitos):

* El testing es una etapa que comienza al terminar de codificar.
* El testing es probar que el software funciona.
* Testing = Calidad de PRODUCTO.
* Testing = Calidad de PROCESO.
* El tester es el enemigo del programador.

**Principios del Testing:**

* El testing muestra presencia de defecto.
* El testing exhaustivo es imposible (no se puede probar todo, ya que una cobertura del 100% no es posible por motivos de costo-beneficio).
* Testing temprano, y tratar de incorporarlo con aseguramiento de calidad.
* Agrupamiento de defectos.
* Paradoja del pesticida.
* El testing es dependiente del contexto, por lo que hay que ir cambiando tanto los casos de prueba, como los testers: Distintas visiones permiten encontrar defectos que pueden haberse pasado por alto (no hay que hacer siempre lo mismo, y que lo haga la misma gente).
* Falacia de la ausencia de errores.
* Un programador debería evitar probar su propio código, ya que, si uno pruebo lo que hace, es complicado no tener una mirada subjetiva que derive en no encontrar errores.
* Una unidad de programación no debería probar sus propios desarrollos.
* Examinar el software para probar que no hace lo que se supone que debería hacer es la mitad de la batalla, la otra mitad es ver que hace lo que no se supone que debería hacer.
* No planificar el esfuerzo de testing sobre la suposición de que no se van a encontrar defectos.

**Pero. ¿Cuánto testing es suficiente?**

* El testing exhaustivo es imposible.
* Decidir cuánto testing es suficiente depende de:
  + Evaluación del nivel de riesgo.
  + Costos asociados al proyecto.
* Usamos los riesgos para determinar:
  + Qué testear primero.
  + A qué dedicarle más esfuerzo de testing.
  + Qué no testear (por ahora).
* El **Criterio de Aceptación** es lo que comúnmente se usa para resolver el problema de determinar cuándo una determinada fase de testing ha sido completada.
* Puede ser definido en términos de:
  + Costos.
  + % de tests corridos sin fallas.
  + Fallas predichas aún permanecen en el software.
  + No hay defectos de una determinada severidad en el software.

**La Psicología del Testing:**

* La búsqueda de fallas puede ser vista como una crítica al producto y/o su autor.
* La construcción del software requiere otra mentalidad a la de testear el software.

**Error y Defecto:**

* **Error:** Se encuentra en la misma etapa en la que se lo introdujo.
* **Defecto:** Se encuentra en otra etapa distinta a la que se lo introdujo. Obviamente, el peor momento para encontrar defectos es en la etapa de producción.

Tanto los errores como los defectos, son **FALLAS**, que se pueden clasificar según su nivel de:

|  |  |
| --- | --- |
| Severidad | Prioridad |
| Bloqueante | Urgencia |
| Crítico | Alta |
| Mayor | Media |
| Menor | Baja |
| Cosmético |  |

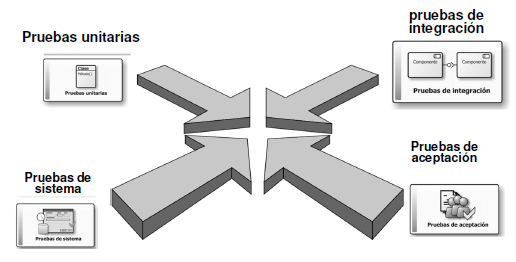
La **Severidad** hace referencia al impacto que la falla tiene sobre el producto, mientras que la **Prioridad** es un concepto del lado del cliente, es decir, qué tan urgente necesita que alguna funcionalidad del sistema ande correctamente.

¿Existe una correspondencia de uno a uno entre las severidades y prioridades?

1. Siempre, los bugs críticos tienen prioridad alta.
2. Siempre, los bugs cosméticos tienen prioridad baja.
3. Las urgencias solo pueden corresponder a bugs bloqueantes.
4. Todos los anteriores.
5. Ninguno de los anteriores.

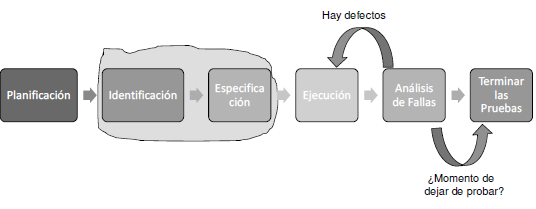
Niveles de Testing.

Partiendo de la base que el objetivo del testing es tratar de automatizarlo (siempre analizando la relación costo-beneficio con respecto a hacerlo manual). Hay distintos tipos de pruebas, que dependen del nivel de granularidad con el que se prueban las partes de un software.

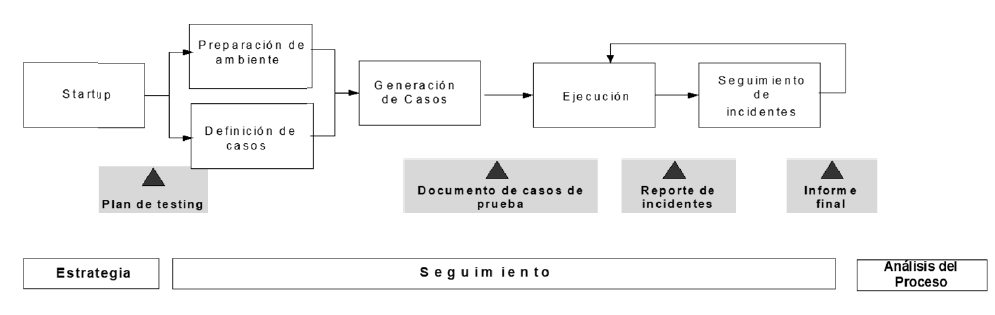


1. **Testing Unitario:**
   1. Se prueba cada componente tras su realización/construcción 🡪 Nivel de granularidad más alto.
   2. Solo se prueban componentes individuales.
   3. Cada componente es probado de forma independiente.
   4. Se produce con acceso al código bajo pruebas, y con el apoyo del entorno de desarrollo, tales como un framework de pruebas unitarias, o herramientas de depuración, es decir, lo realiza el propio desarrollador que escribió el código.
   5. Al ser realizado por el propio desarrollador, este testing viola el principio de objetividad.
   6. Los errores se suelen reparar tan pronto como se encuentran, sin constancia oficial de los incidentes.
   7. Su objetivo no es encontrar defectos (debido a que aún se está en producción), si no que lo que busca es que la funcionalidad desarrollada haga lo que tiene que hacer, es decir, que los requisitos estén bien definidos/implementados.
2. **Testing de Integración:**
   1. Test orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto (sus componentes).
   2. Cualquier estrategia de prueba de versión o de integración debe ser incremental, para lo que existen dos esquemas principales:
      1. Integración de arriba hacia abajo (top-down): De componentes más grandes a más chicos.
      2. Integración de abajo hacia arriba (bottom-up): De componentes más chicos a más grandes.
   3. Lo ideal es una combinación de ambos esquemas (sándwich).
   4. Este tipo de testing es llevado a cabo por el tester, el cual en este caso sí busca encontrar defectos.
   5. Tener en cuenta que los módulos críticos deben ser probados lo más tempranamente posible.
   6. Los puntos clave del test de integración son simples:
      1. Conectar de a poco las partes más complejas.
      2. Minimizar la necesidad de programas auxiliares.
3. **Testing de Sistema:**
   1. Es la prueba realizada cuando una aplicación está funcionando como un todo (**Prueba de Construcción Final**), es decir, se prueba una versión del producto cercana a la versión de la producción.
   2. Trata de determinar si el sistema en su globalidad opera satisfactoriamente (recuperación de fallas, seguridad y protección, estrés, performance, etc.).
   3. El entorno de prueba debe corresponder al entorno de producción tanto como sea posible, para reducir al mínimo el riesgo de incidentes debidos al ambiente específicamente y que no se encontraron en las pruebas.
   4. Deben investigar tanto requerimientos funcionales y no funcionales del sistema.
   5. Testing también realizado por el tester.
4. **Testing de Aceptación:**
   1. Es la prueba realizada por el usuario para determinar si la aplicación se ajusta a sus necesidades.
   2. La meta en las pruebas de aceptación es la de establecer confianza en el sistema, las partes del sistema o las características específicas y no funcionales del sistema.
   3. Encontrar defectos no es el foco principal en las pruebas de aceptación.
   4. Comprender tanto la prueba realizada por el usuario en ambiente de laboratorio (pruebas alfa), como la prueba en ambientes de trabajo reales (pruebas beta).

**Proceso de Pruebas:**



*Ejemplo de etapas/entregables de Testing:*



**Proceso de Testing:**

1. **Planificación y Control:**
   * La planificación de las pruebas es la actividad de verificar que se entienden las metas y los objetivos del cliente, las partes interesadas (stakeholders), el proyecto, y los riesgos de las pruebas que se pretenden abordar.
   * Construcción del Test Plan:
     + Riesgos y Objetivos del Testing.
     + Estrategia de Testing.
     + Recursos.
     + Criterio de Aceptación.
   * Controlar:
     + Revisar los resultados del testing.
     + Test coverage y criterio de aceptación.
     + Tomar decisiones.
2. **Identificación y Especificación:**
   * Revisión de la base de pruebas.
   * Verificación de las especificaciones para el software bajo pruebas.
   * Evaluar la testeabilidad de los requerimientos y el sistema.
   * Identificar los datos necesarios.
   * Diseño y priorización de los casos de las pruebas.
   * Diseño del entorno de prueba.
3. **Ejecución:**
   * Desarrollar y dar prioridad a nuestros casos de prueba.
   * Crear los datos de prueba.
   * Automatizar lo que sea necesario.
   * Creación de conjuntos de pruebas de los casos de prueba para la ejecución de la prueba eficientemente.
   * Implementar y verificar el ambiente.
   * Ejecutar los casos de prueba.
   * Registrar el resultado de la ejecución de pruebas y registrar la identidad y las versiones del software en las herramientas de pruebas.
   * Comparar los resultados reales con los resultados esperados.
4. **Evaluación y Reporte:**
   * Evaluar los criterios de aceptación.
   * Reporte de los resultados de las pruebas para los stakeholders.
   * Recolección de la información de las actividades de prueba completadas para consolidar.
   * Verificación de los entregables y que los defectos hayan sido corregidos.
   * Evaluación de cómo resultaron las actividades de testing y se analizan las lecciones aprendidas.

**Caso de Prueba:** Principal artefacto del testing. Set de condiciones o variables bajo las cuales un tester determinará si el software está funcionando correctamente o no. Una buena definición de casos de prueba ayuda a reproducir defectos.

Un caso de prueba es la unidad de la actividad de la prueba. Consta de 3 partes:

1. **Objetivo:** La característica del sistema a comprobar.
2. **Datos de Entrada y de Ambiente:** Datos a introducir al sistema que se encuentra en condiciones preestablecidas.
3. **Comportamiento Esperado:** La salida o la acción esperada en el sistema de acuerdo a los requerimientos del sistema.

A modo de resumen sobre los casos de prueba:

* **OBJETIVO:** Descubrir errores.
* **CRITERIO:** En forma completa.
* **RESTRICCIÓN:** Con el mínimo de esfuerzo y tiempo.

Otra definición: Plantea un escenario (con su nombre) que debe ser específico (todos los datos del contexto con el máximo nivel de detalle), así como también un conjunto de pasos ordenados, especificando cada acción (también con alto nivel de detalle, y sin ambigüedad), para poder brindar reproducibilidad (que se obtenga el mismo resultado ante la misma entrada).

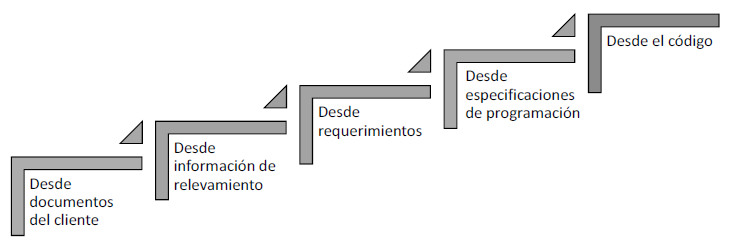
A su vez, también plantea precondiciones (obviamente también bien detalladas, con una descripción específica y completa); un resultado esperado con detalles específicos de lo que se espera (para poder decir que hay un defecto cuando el resultado obtenido es distinto a éste - -; y tiene prioridad (la cual determina el riesgo asociado, la criticidad de los defectos, y qué tan importante es encontrar un defecto).

Un escenario feliz para un caso de prueba, es aquel que tendrá prioridad **Alta**, mientras que uno que tendrá prioridad **Baja** será, por ejemplo, un caso de prueba que busque defectos en campos de textos para el ingreso de datos. Al medio (prioridad **Media**), hay una gama amplia de escenarios que tengan caminos distintos a partir de cambios de parámetros, que generen resultados inesperados.

**Condiciones de Prueba:** Esta es la relación esperada de un sistema frente a un estímulo particular; este estímulo está constituido por las distintas entradas.

Una condición de prueba debe ser probada por al menos un caso de prueba.

Derivación de Casos de Prueba:



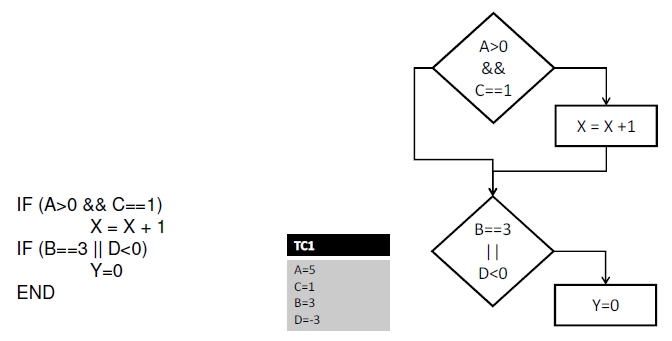
Estrategias para tratar los casos de prueba (para hacer testing en sí): Existen métodos que se utilizan debido a que el tiempo y el presupuesto son limitados y, además, porque hay que pasar por la mayor cantidad de funcionalidades con la menor cantidad de pruebas. Estos métodos son:

**Caja Negra:** No se conoce la implementación, si no que solo se conocen los requerimientos de entradas y de salidas.

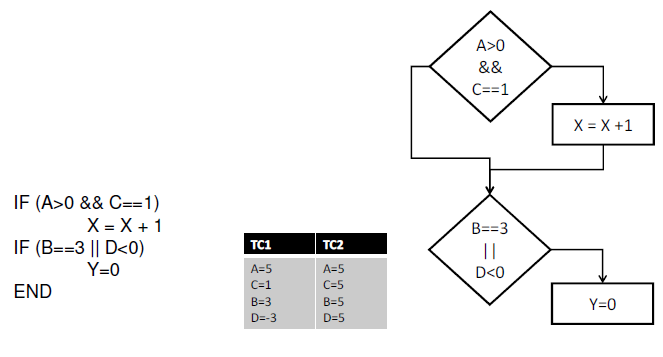
* **Basado en Especificaciones:** Con los detalles de los requerimientos, creo casos de prueba que afecten directo a las entradas y a las salidas.
  + Partición de Equivalencias: Busca dividir a una condición de entrada o salida, en un subconjunto equivalente (para variables de entrada/salida, generar un subconjunto de datos que producen un resultado equivalente). Consta de 2 pasos:
    - Identificar las clases de equivalencias (válidas y no válidas):
      * Rango de valores continuos.
      * Valores discretos.
      * Selección simple.
      * Selección múltiple.
    - Identificar los casos de prueba.
  + Análisis de Valores Límites: Es una variante de la partición de equivalencias. En vez de seleccionar cualquier elemento como representativo de una clase de equivalencia, se seleccionan los bordes de una clase (debido a que en éstos hay más probabilidades de encontrar errores).
* **Basado en la Experiencia:** Depende de la expertise de los desarrolladores.
  + Adivinanza de defectos.
  + Testing explorativo.

**Caja Blanca:** Sí se conoce la implementación (el código).

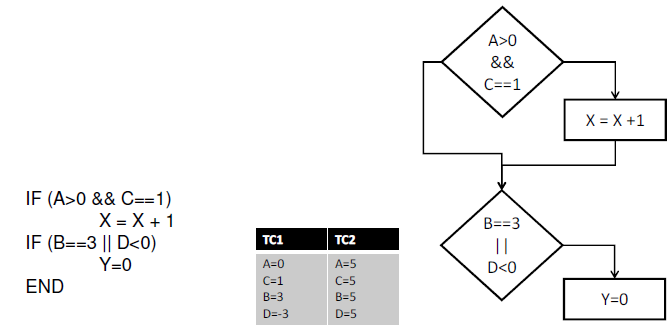
* Se basan en el análisis de la estructura interna del software, o un componente del software.
* Se pueden garantizar el testing coverage.
* Cobertura de enunciados o caminos básicos.
* Cobertura de sentencias:



* Cobertura de decisión:

****

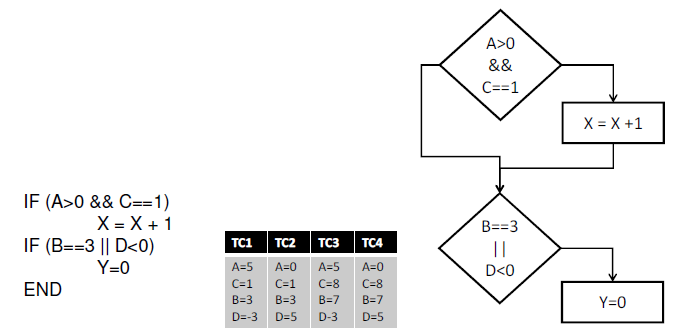
* Cobertura de condición:



* Cobertura de decisión/condición:



* Cobertura múltiple:

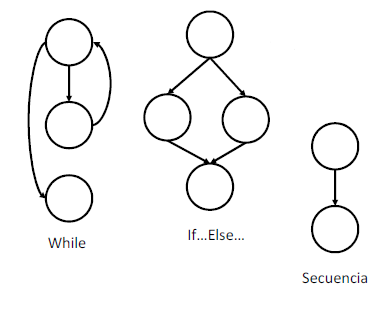


**Cobertura de Enuncias o Caminos Básicos:**

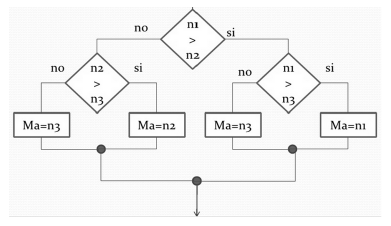
* Propuesto por McCabe.
* Permite obtener una medida de la complejidad de un diseño procedimental, y utilizar esta medida como guía para la definición de una serie de caminos básicos de ejecución.
* Para la prueba del camino básico:
  + Se requiere poder representar la ejecución mediante grafos de flujo.
  + Se calcula la complejidad ciclomática.
    - M = Complejidad ciclomática.
    - E = Número de aristas del grafo.
    - N = Número de nodos del grafo.
    - P = Número de componentes conexos, nodos de salida.

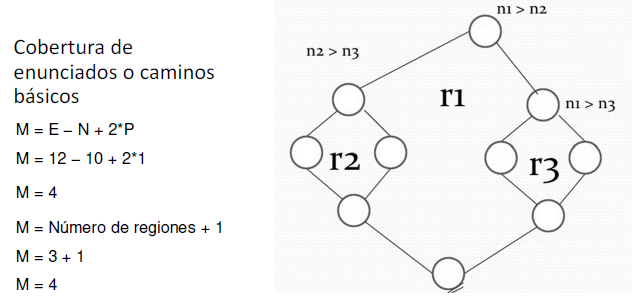
Siendo

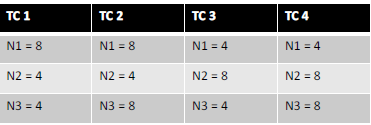
* + Dado un grafo de flujo, se pueden generar casos de prueba.

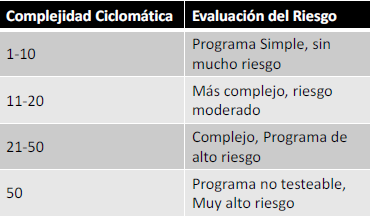


* Es una métrica de software que provee una medición cuantitativa de la complejidad lógica de un programa.
* Usada en el contexto de testing, define el número de caminos independientes en el conjunto básico y entrega un límite inferior para el número de casos necesarios para ejecutar todas las instrucciones al menos una vez.
* Ejemplo:









* Pasos del diseño de pruebas mediante el camino básico:
  + Obtener el grafo de flujo.
  + Obtener la complejidad ciclomática del grafo de flujo.
  + Definir el conjunto básico de caminos independientes.
  + Determinar los casos de prueba que permiten la ejecución de cada uno de los caminos anteriores.
  + Ejecutar cada caso de prueba, y comprobar que los resultados son los esperados.

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

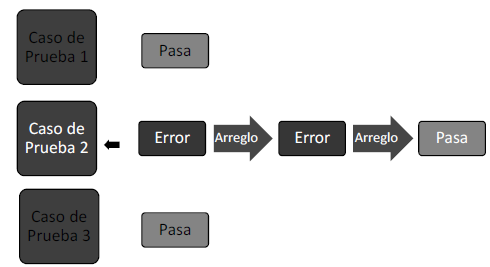
Conclusiones sobre la Generación de Casos:

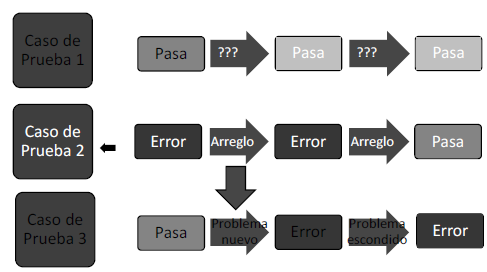
* Ninguna técnica es completa.
* Las técnicas atacan distintos problemas.
* Lo mejor es combinar varias de estas técnicas para complementar las ventajas de cada una.
* Depender del código a testear.
* Sin requerimientos todo es mucho más difícil.
* Tener en cuenta la conjetura de defectos.
* Ser sistemático y documentar las suposiciones sobre el comportamiento o el modelo de fallas.

**Ciclo de Test:** Un ciclo de pruebas abarca la ejecución de la totalidad de los casos de prueba establecidos aplicados a una misma versión del sistema a probar.

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

Ejemplos de SIN regresión:





Ejemplos CON regresión:



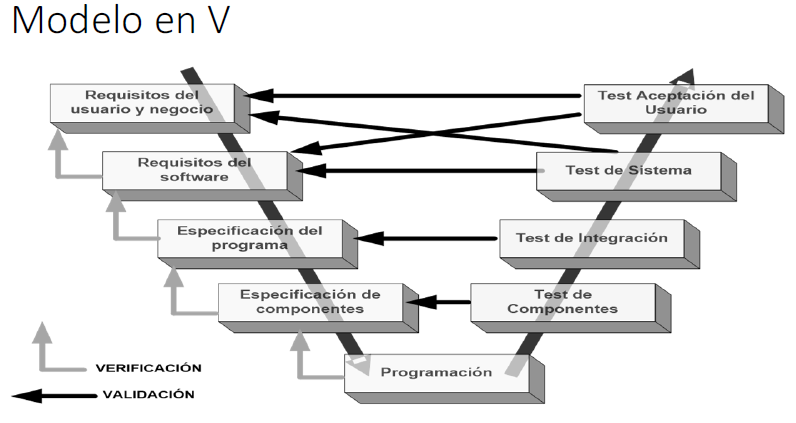
**Smoke Test:** Primer corrida de los test de Sistema que provee cierto aseguramiento de que el software que está siendo probado no provoca una falla catastrófica.

**Tipos de Pruebas:**

* **Testing Funcional:**
  + Las pruebas se basan en funciones y características (descripta en los documentos o entendidas por los testers), y su interoperabilidad con sistemas específicos.
    - Basado en Requerimientos.
    - Basado en los procesos de negocios.
* **Testing No Funcional:**
  + Es la prueba de “cómo” funciona el sistema.
  + No hay que olvidarlas. Los requerimientos no funcionales son tan importantes como los funcionales:
    - Performance Testing.
    - Pruebas de Carga.
    - Pruebas de Estrés.
    - Pruebas de Usabilidad.
    - Pruebas de Mantenimiento.
    - Pruebas de Fiabilidad.
    - Pruebas de Portabilidad.

**El Testing en el ciclo de vida del software:** Objetivos de involucrar las actividades de testing de manera temprana:

* Dar visibilidad de manera temprana al equipo, de cómo se va a probar el producto.
* Disminuir los costos de correcciones de defectos.



**TDD:** EL acto de diseñar tests es uno de los mecanismos conocidos más afectivos para prevenir errores. El proceso mental que debe desarrollarse para crear test útiles puede descubrir y eliminar problemas en todas las etapas del desarrollo.

* Desarrollo guiado por pruebas de software, o Test-Driven Development (TDD).
* Es una técnica avanzada que involucre otras dos prácticas:
  + Escribir las pruebas primero (Test FIrst Development).
  + Refactorizar (Refactoring).
* Para escribir las pruebas, generalmente se utilizan las pruebas unitarias.