Ciclos de vida y procesos de desarrollo de software

Un **ciclo de vida** representa el conjunto de **etapas que recorre un proyecto o producto a lo largo del tiempo**. Define:

- Las **actividades** del proceso (requisitos, diseño, pruebas...).
- El **orden** en que se ejecutan.
- Los **criterios para pasar** de una etapa a otra.

Relación Producto vs Proyecto

- El **producto** tiene un ciclo de vida más **largo** que el proyecto.
- Un mismo producto puede tener **varios proyectos** (actualizaciones, mejoras, etc.).
- El ciclo de vida del **proyecto** termina con la entrega del software; el del **producto** continúa mientras el software siga en uso.

Tipos de Ciclos de Vida

1. Secuencial (Cascada)

- Etapas **lineales y rígidas**: no se avanza hasta terminar la anterior.
- Requiere requisitos **claros y estables desde el inicio**.
- Genera mucha documentación.
- El producto final se ve **recién al final** del proyecto.

Útil cuando:

- El contexto es **predecible** y bien definido.

Dificultades:

- No se adapta bien al **cambio**.
- Altos riesgos si aparece un error tarde.

2. Iterativo/Incremental

- Se desarrolla por **versiones** que **agregan funcionalidad** progresivamente.
- Cada iteración entrega **software usable y evaluable**.
- El cliente participa y da feedback continuo.

Ventajas:

- Ideal para **requisitos cambiantes**.
- El cliente ve resultados **temprano**.
- Menor retrabajo.

Dificultades:

- Puede haber **poca visibilidad global**.
- Requiere esfuerzo constante en documentación y mantenimiento de calidad.

3. Recursivo (Espiral)

- Se enfoca en **gestionar riesgos**.
- Cada vuelta de la espiral incluye:
    1. Objetivos
    2. Evaluación de riesgos
    3. Desarrollo y pruebas
    4. Planificación próxima vuelta
- Usa **prototipos evolutivos**.

Ventajas:

- Útil para **sistemas complejos** y de alto riesgo.

Dificultades:

- Costoso, difícil de reutilizar.
- Requiere **alta intervención del cliente**.

Procesos de Desarrollo de Software: Empíricos vs. Definidos

Un **proceso de desarrollo de software** es la forma organizada en que se lleva adelante un proyecto, combinando actividades, métodos y herramientas para construir un sistema. Incluye tareas como la especificación de requerimientos, el diseño, la implementación, la validación y la evolución del software.

Este proceso está influenciado por factores clave como los procedimientos definidos, el uso de herramientas adecuadas, y sobre todo, la capacidad y motivación de las personas involucradas.

Procesos Definidos

Los **procesos definidos** están inspirados en las líneas de producción. Asumen que, si se siguen los mismos pasos con las mismas entradas, se obtendrán siempre los mismos resultados, sin importar qué personas lo ejecuten.
Por eso, se enfocan fuertemente en la planificación y la documentación como mecanismos de control.

Se los considera adecuados cuando se necesita **previsibilidad, repetibilidad y estructura**, como en sistemas críticos.
Su administración se basa en seguir un plan detallado, y cada fase se documenta y valida antes de avanzar.

Sin embargo, este tipo de procesos tiene limitaciones: **no se adaptan bien a los cambios**, y muchas veces, la gestión del proceso se vuelve más importante que el avance real del software.

Procesos Empíricos

En cambio, los **procesos empíricos** no parten de la idea de que todo se puede planificar de antemano. Asumen que el desarrollo de software es una actividad compleja, creativa y cambiante, y que cada proyecto tiene un contexto particular.

Por eso, se basan en ciclos cortos de trabajo con **inspección y adaptación constantes**, permitiendo ajustarse a lo que realmente está ocurriendo.
No buscan predecir el futuro, sino adaptarse a él con base en la experiencia directa del equipo.

En este enfoque, la administración se realiza observando los resultados obtenidos, compartiendo el conocimiento dentro del equipo y tomando decisiones colaborativas.
La transparencia, la adaptación y el aprendizaje son pilares fundamentales.

Este tipo de proceso **es ideal para entornos ágiles** y se aplica normalmente con ciclos de vida iterativo-incrementales, como en Scrum. Por el contrario, no se recomienda usarlo con modelos secuenciales, ya que no toleran bien el cambio.

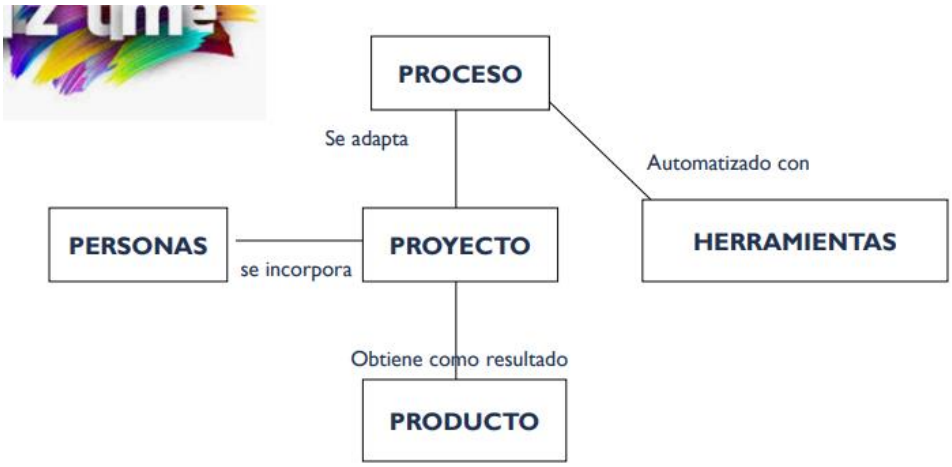
Eleccion del Ciclo de vida

La elección del ciclo de vida depende del contexto del proyecto: claridad de requerimientos, riesgo, necesidad de entregas parciales, y tipo de proceso (definido o empírico).

- **Secuencial**
  - Usa procesos definidos.
  - Ideal con **baja incertidumbre** y **requerimientos claros**.
  - No permite entregas parciales.
  - Se prioriza la **planificación y documentación**.
- **Iterativo / Incremental**
  - Apto para procesos definidos o empíricos.
  - Funciona bien con **cambios frecuentes** y **feedback continuo**.
  - Permite entregas parciales y adaptación.
  - Útil cuando el cliente necesita ver resultados rápido.
- **Recursivo (Espiral)**
  - Usa procesos definidos.
  - Se enfoca en **gestión de riesgos**.
  - No permite entregas parciales tempranas.
  - Ideal para proyectos **grandes o complejos**.

Proyecto de sistema de información

Un **proyecto** es un esfuerzo **temporal**, planificado para obtener un **producto, servicio o resultado único**.
En el contexto del software, es la **unidad organizativa** donde se adapta el proceso teórico a los recursos, personas y herramientas disponibles.



Características de un proyecto

- **Único**: Cada proyecto es irrepetible, aunque sea similar a otros.
- **Orientado a objetivos**: Los objetivos deben ser claros, medibles y alcanzables.
- **Temporal**: Tiene inicio y fin definidos.
- **Basado en tareas interrelacionadas**: Hay coordinación entre esfuerzos, tiempos y recursos.

Administración de Proyectos de Software

Administrar un proyecto implica **organizar personas y recursos** para cumplir con:

- **Tiempo** pactado.
- **Presupuesto** definido.
- **Calidad** y funcionalidades acordadas.

Dos grandes actividades:

1. **Planificación** (objetivos, riesgos, alcance, recursos).
2. **Monitoreo y control** (seguir lo planificado y ajustar si hace falta).

Líder de Proyecto

El **líder de proyecto** es la persona que tiene la **responsabilidad total sobre el proyecto**.
Su rol principal es **coordinar al equipo**, **tomar decisiones**, **asignar tareas** y **comunicarse con los actores externos** del proyecto, como clientes, gerentes, subcontratistas u organismos regulatorios.

En enfoques tradicionales, el líder **toma las decisiones solo**, sin participación activa del equipo.
Es el **punto de contacto principal** entre el proyecto y el entorno.

Equipo de Proyecto

Es un **grupo de personas comprometidas con alcanzar objetivos comunes**, de los cuales se sienten **mutuamente responsables**.

Características:

- Tienen conocimientos y habilidades **diversos**.
- Suelen ser **grupos pequeños**.
- Buscan una **buena comunicación y sinergia**.
- Se sienten **responsables como unidad**.

Plan de Proyecto

El **plan de proyecto** (o plan de desarrollo de software) es un documento de gestión que actúa como **hoja de ruta** para todo el desarrollo.
En él se deja asentado **qué se va a hacer, cómo, cuándo y quién lo hará**, incluyendo tareas, responsables, tiempos, riesgos, métricas y controles.
Es una pieza central para coordinar el trabajo del equipo y tomar decisiones a lo largo del proyecto.

1. Definición del Alcance

El **alcance** se divide en dos conceptos:

- **Alcance del producto**: son todas las **características y funcionalidades** que debe tener el software para cumplir con lo que el cliente necesita.
- **Alcance del proyecto**: es **todo el trabajo y solo el trabajo necesario** para entregar ese producto con las características definidas.

📏 **¿Cómo se mide el alcance?**

- El cumplimiento del **alcance del producto** se mide contra la **Especificación de Requerimientos (ERS)**.
- El cumplimiento del **alcance del proyecto** se mide contra el **Plan de Proyecto**.

2. Definición del proceso y ciclo de vida

Consiste en decidir **cómo se va a desarrollar el proyecto**, eligiendo un proceso y un ciclo de vida adecuados al contexto.
El proceso define la forma de trabajo general (por ejemplo, ágil o tradicional), mientras que el ciclo de vida determina **qué se hace en cada fase, cuándo y con quién**.
En metodologías tradicionales se puede elegir cualquier ciclo; en metodologías ágiles se exige un enfoque **iterativo e incremental**.

3. Estimación

La **estimación** busca anticipar cuánto va a costar, durar o requerir un proyecto.
En enfoques tradicionales, la realiza el líder de proyecto.

Sigue un orden lógico:

- **Primero se estima el tamaño del producto**, basado en los requerimientos (por ejemplo, líneas de código, puntos de función).
- A partir de ahí se calcula el **esfuerzo necesario**, es decir, cuántas horas-persona se requerirán.
- Luego se estima el **calendario**, definiendo en qué momentos se harán las tareas.
- Por último, se calcula el **costo**, que depende directamente del esfuerzo y del tamaño.

Una mala estimación del tamaño al principio genera errores en todo lo demás. Además, las habilidades del equipo influyen mucho: un grupo con más experiencia puede reducir el esfuerzo total requerido.

4. Gestión de Riesgos

Un **riesgo** es un **problema esperando para suceder**. Es un evento que **podría comprometer el éxito del proyecto** si llegara a ocurrir.
Por eso, la gestión de riesgos debe hacerse **a lo largo de todas las fases del proyecto**, de forma constante y proactiva.

Proceso de gestión (según la imagen):

1. **Identificar**
   Se reconocen posibles amenazas usando experiencias previas (base de conocimiento de riesgos) y contexto del proyecto.
2. **Analizar**
   Se evalúa la **probabilidad de ocurrencia** y el **impacto** de cada riesgo. Con esto se crea una **lista priorizada** (top N).
3. **Planificar**
   Se definen estrategias para mitigar los riesgos más importantes o responder si ocurren (planes de contingencia).
4. **Seguimiento y Control**
   Se revisa periódicamente si los riesgos cambiaron, si siguen siendo relevantes, o si aparecieron nuevos.
5. **Aprendizaje**
   Todo lo aprendido se reutiliza para proyectos futuros, alimentando la base de conocimiento de riesgos.

5. Asignación de responsabilidades (o recursos)

Consiste en **asignar roles concretos a las personas** del equipo de proyecto.
Tu profe aclara que no le gusta llamarlos “recursos humanos”, ya que el desarrollo de software es una **actividad humano-intensiva**, y las capacidades del equipo impactan directamente en la calidad del producto.

Una misma persona puede cumplir **más de un rol** si el proyecto es chico o hay limitaciones de presupuesto.
Para dejarlo claro, suele armarse una **tabla de roles y responsabilidades**, detallando qué hace cada integrante.

6. Programación de proyectos (calendarización)

Esta etapa toma como base la estimación previa del calendario y la **refina al máximo nivel de detalle**.
Se descompone el proyecto en **tareas específicas**, indicando quién las hace, cuándo y cuánto esfuerzo llevarán.

Suele representarse con un **Diagrama de Gantt**, usando herramientas como Microsoft Project, para visualizar la duración y dependencia entre tareas.

7. Definición de métricas

Las **métricas** son valores numéricos que permiten **medir el estado o avance** del proyecto, proceso o producto.
En enfoques tradicionales, se definen desde el inicio y se usan para hacer seguimiento real.

Las métricas más comunes son:

- **Tamaño del producto**
- **Esfuerzo**
- **Tiempo (calendario)**
- **Defectos**
- **Costos**

Estas métricas deben ser claras, representativas y útiles para evaluar el progreso.
En ágil se mide diferente: el mayor indicador de avance son los **incrementos entregados**.

8. Definición de controles

Los controles se basan en las métricas definidas, y su función es **verificar que todo se cumple como fue planificado**.

Incluye:

- **Reuniones periódicas**
- **Informes de avance**
- **Puntos de revisión**

Son fundamentales para **monitorear** y **ajustar** el rumbo del proyecto en tiempo real.

Manifiesto Ágil

Es un documento redactado en 2001 por expertos de la ingeniería en software, que **prioriza el valor entregado al cliente** y la **flexibilidad ante los cambios** por sobre procesos rígidos.

Valores Agiles:

1) **Individuos en interacciones, por sobre procesos y herramientas:** Es preferible un equipo de personas motivados con un buen funcionamiento, pero con herramientas pobres, a tener un equipo desmotivado con las mejores herramientas
2) **Software funcionando por sobre documentación extensiva:** Se prioriza el software para lograr versiones estables y mejoradas al finalizar cada iteración, y solo se documenta aquello que agregue valor al producto y se centre en el cliente
3) **Colaboración con el cliente por sobre negociación contractual:**
   -Se busca un cliente involucrado en el desarrollo, no solo al final del proceso.
   -La comunicación constante evita conflictos y malentendidos sobre requisitos.
   -Los contratos son necesarios, pero no deben impedir la flexibilidad ni la adaptación a nuevas necesidades.
4) **Responder a cambios por sobre seguir un plan:** Es imposible que los usuarios sepan con detalle y certeza todas las funcionalidades que necesitan, y el equipo ágil debe estar abierto a cambio, respondiendo rápido y con el mayor valor posible. Sigue los valores de inspección y adaptacion

Principios del manifiesto ágil:

1) **Nuestra mayor prioridad es satisfacer al cliente:** Queremos que el cliente esté contento, y eso se logra entregando software que realmente le sirva, y rápido.
2) **Aceptar que los requisitos cambien:** El equipo ágil esta abierto al cambio, logrando asi que el equipo sea flexible y responda a las necesidades cambiantes del cliente
3) **Entregar software funcional frecuentemente:** Al finalizar cada iteración se debe haber transformado uno o mas requerimientos, en código ya desarrollado
4) **Técnicos y no técnicos trabajando juntos durante todo el proyecto:** El equipo de desarrollo y el cliente tienen que estar en contacto constante. El PO es clave acá.
5) **Desarrollamos proyectos en torno a individuos motivados:** Los miembros del equipo deben estar comprometidos a colaborar activamente con los objetivos del proyecto, y no solo a cumplir tareas aisladas.
6) **La mejor forma de comunicar es cara a cara:** es la forma mas eficiente para transmitir información ya que facilita la comprensión
7) **La mayor métrica del progreso es el software funcionando:** El avance se mide en base a la entrega del software útil para el cliente, y no en la cantidad de documentación generada
8) **Los procesos agiles promueven el desarrollo sostenible:** se fomenta la estabilidad de los equipos a lo largo del tiempo, para mejorar su dinámica de trabajo y capacidad de estimación
9) **Excelencia técnica y buen diseño:** la calidad del producto no es un aspecto negociable. Cuanto mejor diseñado está, más fácil es adaptarlo después.
10) **la simplicidad es esencial**: se debe enfocar en las necesidades que aporten valor y evitar funcionalidades innecesarias sin valor
11) **Las mejores arquitecturas, diseños y requerimientos surgen de equipos autoorganizados:** La toma de decisiones debe surgir dentro del equipo, sin depender de una restructura jerárquica
12) **El equipo evalúa su desempeño y ajusta la manera de trabajar:** a intervalos regulares, el equipo revisa su desempeño y ajusta su forma de trabajo para optimizar la respuesta a las necesidades del cliente

Filosofía Ágil:

La filosofía ágil **no es un proceso ni un framework**, es una **forma de pensar**. Es lo que está detrás del manifiesto y los principios.

Se basa en un ciclo de vida **iterativo – incremental**, Equipos **autoorganizados**, Trabajo **colaborativo y continuo con el cliente**, **Mejora continua** basada en retroalimentación.

El enfoque ágil reduce la documentación y prioriza la comunicación informal con el cliente.

Pilares del empirismo:

Transparencia: nos permite crecer como equipo, y el conocimiento de cada miembro del equipo se transforma en conocimiento de todo el equipo.
La transparencia permite la inspección. La inspección sin transparencia genera engaños y desperdicios

inspección: Se revisan productos y procesos para detectar problemas y oportunidades de mejoras
La inspección permite la adaptación. La inspección sin adaptación se considera inutil

Adaptación: Se hacen ajustes constantes para optimizar el proceso y mejorar el producto

Filosofía Lean:

Busca maximizar el valor al cliente con el mínimo uso de recursos, eliminando todo lo que no aporte valor. Se enfoca en entregar productos de alta calidad, bajo costo y en el momento exacto en que el cliente los necesita.

7-Principios Lean

1) **Eliminar desperdicios:** El enfoque Lean busca eliminar todo aquello que no aporta valor en el proceso de desarrollo, mejorando la eficiencia y evitando actividades innecesarias.
   Tipos de desperdicios en Lean:
   a. Características extras: Funcionalidades que el cliente no pidió o no necesita.
   b. Trabajo a medias: Cosas empezadas, pero no terminadas. Se acumulan y generan retrabajo.
   c. Proceso extra: Pasos innecesarios en el desarrollo que no agregan valor.
   d. Búsqueda de información: Tiempo perdido buscando datos, documentos, mails, accesos.
   e. Defectos: Bugs o errores que llegan a producción o que se descubren tarde.
   f. Esperas: Tiempo perdido por bloqueos o dependencias.
   g. Cambios de tareas (Multitasking): Saltar de una tarea a otra todo el tiempo disminuye la eficiencia y aumenta el tiempo de finalización.
   h. Talento no utilizado: No aprovechar el conocimiento y creatividad del equipo.

2) **Amplificar el aprendizaje:** Convertir el conocimiento implícito en explícito para fortalecer el equipo. En lugar de escribir todo en un documento, preferimos **probar, aprender y compartir**.
3) **Embeber la integridad conceptual:** Significa que el sistema sea **coherente, simple y fácil de entender**, tanto por el cliente como por los desarrolladores. No alcanza con que funcione: tiene que estar **bien hecho**, que tenga un diseño limpio y lógico.
4) **Diferir compromisos hasta el último momento responsable:** Tomar decisiones cuando haya suficiente información sin retrasarlas demasiado.
5) **Dar poder al equipo:** Equipos multifuncionales, autogestionados y motivados. El equipo tiene que poder **tomar decisiones técnicas y de diseño**, porque es quien mejor conoce el sistema.
6) **Ver el todo:** No sirve optimizar partes si el sistema completo funciona mal. Este principio busca **tener una mirada global**, no parcial.
7) **Entregar lo antes posible:** Cuanto antes el cliente pueda ver y usar el producto, mejor. Lean promueve entregas **rápidas y pequeñas**, que den valor real.

Relación Lean-Ágil

Aunque Lean nació en la industria y Ágil en el software, **Ágil se inspiró directamente en Lean**.

Muchos principios y valores ágiles **vienen de Lean**, adaptados al mundo del software. Podés pensar que:

- **Lean es la raíz (filosofía madre)**.
- **Ágil es una evolución pensada para equipos de desarrollo.**

Lean	**Ágil**
Entregar valor al cliente	Entregar software funcional
Eliminar desperdicio	Simplicidad, evitar funciones innecesarias
Mejora continua	Retrospectiva, adaptación
Empoderar al equipo	Equipos autoorganizados
Decisiones just-in-time	Adaptación al cambio
Flujo continuo	Iteraciones frecuentes

Lean es una filosofía de producción que busca entregar valor al cliente con la menor cantidad de desperdicio. Ágil toma esa misma idea, pero enfocada en software. Ambas promueven entregas tempranas, mejora continua y equipos empoderados. Por eso se habla de una gestión Lean-Agile: son dos filosofías muy compatibles y complementarias.

Requerimientos Agiles

- En lugar de definir todo desde el principio (como en cascada), los requerimientos en ágil se trabajan de forma **colaborativa, iterativa y liviana**.
- Se van **descubriendo a lo largo del desarrollo**, en contacto directo con el cliente y el equipo.
- No se busca especificar todo con lujo de detalle, sino enfocarse en **lo mínimo necesario** para empezar a construir valor desde temprano.
- Son **flexibles** y se adaptan con el tiempo, en función del feedback del cliente y los cambios en el negocio.

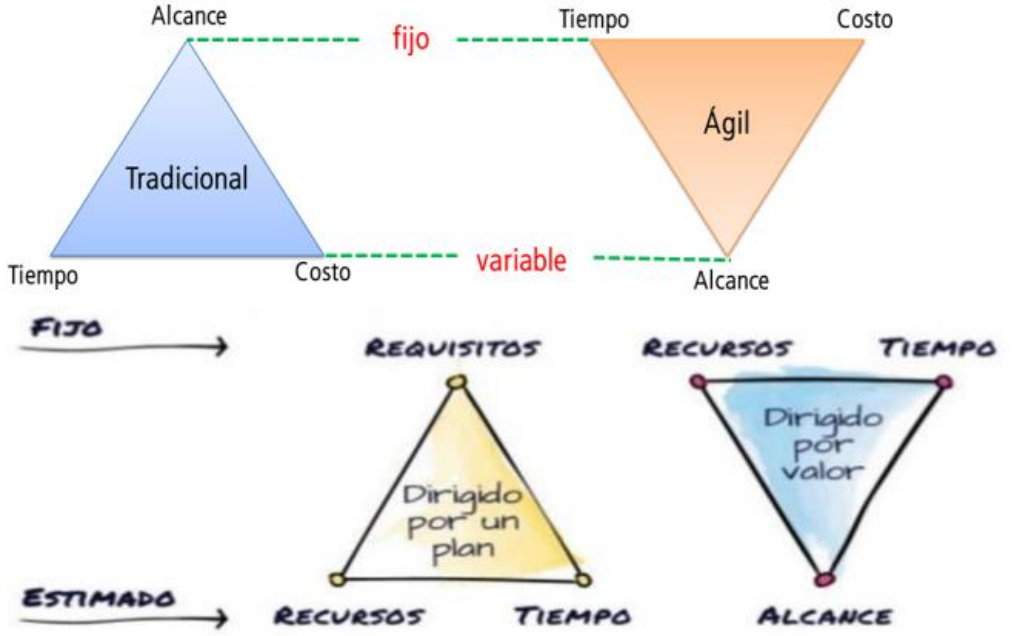
Los requerimientos agiles se basan en 3 pilares:

- **Usar el valor para construir el producto correcto:** Lo único importante es dejar contento al cliente con software funcionando que le sirva, por lo que, el foco de este enfoque de gestión ágil apunta a construir valor de negocio. Hay que construir un producto que le haga la vida más fácil a las personas que lo utilizan.
- **Determinar que es "sólo lo suficiente":** hace referencia a que los requerimientos deben ser encontrados de a poco, no buscar todos los requerimientos desde un inicio y quedarnos solo con eso.
- **Usar historias y modelos para mostrar que construir**: En lugar de documentación pesada, se usan herramientas como User Stories que representan lo que el cliente necesita de forma clara y visual

Triple restricción enfoque ágil y tradicional

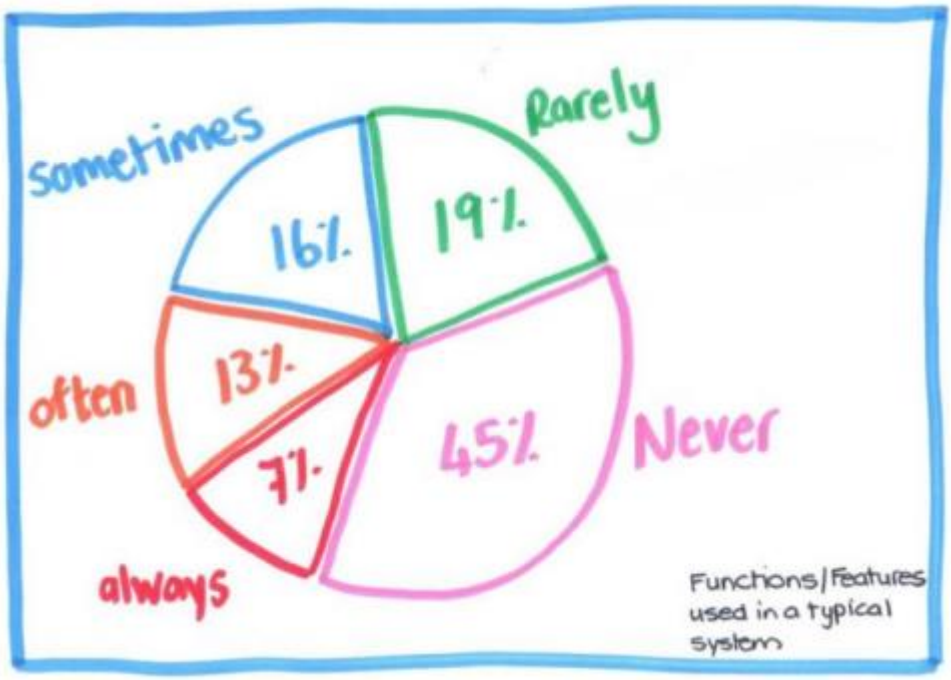
En el enfoque ágil, se trata de aceptar la realidad en la cual el alcance (requerimientos) del producto puede cambiar a lo largo del desarrollo, y no es posible definirlos al comienzo de este. Entonces, el enfoque trata de dejar el tiempo y los recursos fijos, teniendo en cuenta "cuánto software" le puedo entregar en tanto tiempo y con tal equipo de desarrollo.

Por otro lado, en el enfoque tradicional, el alcance se fija y en función de eso derivan los recursos (costo) y el tiempo. El cliente desea controlar el alcance, ya que este comprende lo que él realmente necesita. Luego el cliente pregunta por el tiempo y el costo, los cuales son derivados por el equipo. Por esta razón es tan costosa incorporar cambios de requerimientos, ya que se estaría modificando el alcance y como consecuencia los recursos y el tiempo.



BRUF – Big Requirements Up Front

La imagen adjuntada expresa la frecuencia con la que los usuarios utilizan determinadas funcionalidades de un software. A lo que se apunta con este concepto de BRUF es a notar que, en una primera instancia de un proyecto de software, desarrollar sólo unas pocas funcionalidades de gran valor e importancia para el cliente, puede cubrir gran parte de sus necesidades, a medida que se desarrolla el resto de los requerimientos.



Product Backlog

La gestión ágil busca evitar la situación planteada con anterioridad donde gran parte del software no se termina utilizando. El Product Backlog es una lista priorizada de características y requerimientos del software. Es el Product Owner quien se encarga de definir esta lista y se va a encargar de decidir qué requerimientos necesita primero según el valor que éstos tengan para el negocio. A medida que avanza el proyecto, **se agregan, modifican o eliminan elementos** del Product Backlog. **Nunca está completo al 100%**: siempre está evolucionando.

En contraposición con el enfoque tradicional, donde teníamos un cliente que no estaba dispuesto a comprometerse y dejaba la priorización de los requerimientos al equipo que hace el software, se terminaba construyendo un producto que no dejaba satisfecho al cliente.

Requerimientos Just in Time

Evita desperdicios. Especificar requerimientos, que después cambian, y yo perdí un montón de tiempo especificándolo. El producto no va a estar especificado al 100% desde un principio, sino que vamos a ir encontrando y describiendo requerimientos, conforme nos haga falta. Ni antes, ni después.

En el enfoque tradicional se aplica el “Up Front Specification”, mediante el cual se especifica todos los requerimientos antes de comenzar el desarrollo. Esto implica que un error en la captura de requerimientos tiene un alto impacto.

Fuente de requerimientos

Los requerimientos ágiles, basados en el principio de que el mejor medio de comunicación es el cara a cara, promueven que los requerimientos del software se obtienen conversando cara a cara con el cliente.

Prioriza documentar y especificar aquellos que sean de valor para el cliente, antes que hacer toda una ERS errónea.
En los ambientes tradicionales, los requerimientos se encuentran especificados en un documento de especificación de requerimientos (ERS) y el cliente luego lo lee para validarlo

Momento de la captura de requerimientos

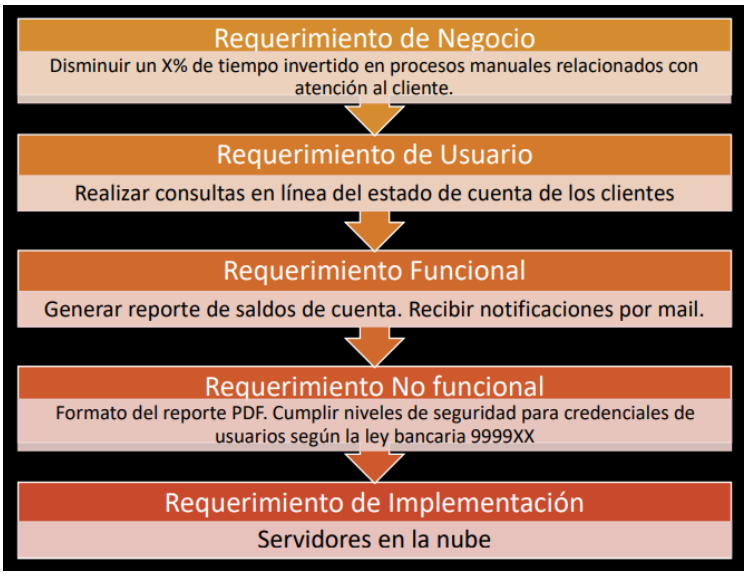
En el enfoque tradicional, los requerimientos son definidos al principio del proyecto.
En el enfoque ágil, los requerimientos son relevados continuamente durante todo el proyecto.

Contenedor y persistencia de los requerimientos

El enfoque tradicional, la ERS contiene los requerimientos a lo largo del ciclo de vida del producto todo el tiempo. En caso de que surjan cambios, es necesario actualizar la ERS.

En el enfoque ágil, se utiliza al Product Backlog como un contenedor transitorio de los requerimientos, ya que una vez implementados estos se "persisten" en el producto funcionando.

Tipos de requerimientos



A nivel de la gestión ágil de requerimientos se trabaja con:

- Requerimiento de negocio
- Requerimiento de usuario

Las User Stories son requerimientos de usuario alineados a un requerimiento de negocio.

Todo parte de un **requerimiento de negocio**, que marca el objetivo. A partir de ahí se define qué necesita el usuario **(requerimiento de usuario),** luego cómo debe comportarse el sistema **(funcionales**), en qué condiciones debe operar **(no funcionales**), y con qué tecnología se implementa **(implementación).**

User story

Es una técnica ágil para representar **requerimientos de usuario** de forma simple y comprensible.

Tiene un **nivel de abstracción alto**, es decir, **qué se necesita** y no **cómo se hace**.

Se escribe en lenguaje de negocio, no técnico.

La escribe el Product Owner, pero se conversa con el equipo.

Partes de la User Story

1) **Tarjeta:** es una descripción de la historia. Tiene una sintaxis particular que es la siguiente

• Nombre del rol: representa quien está realizando la acción o quien recibe el valor de la actividad.
• Actividad: representa la acción que realizará el sistema.
• Valor de negocio: es la más importante y representa el por qué es necesaria la actividad.
• Frase verbal: no es obligatoria, es una forma corta de referenciar la US. Representa la funcionalidad que se expresa en la User Story.
• Criterios de aceptación: son diferentes criterios sobre la actividad que se necesita implementar, que permiten definir los límites de la U.S. para que el equipo tenga una mejor visión de esta. Además, facilitan las tareas de desarrolladores y testers para probar la funcionalidad.
• Pruebas de usuario: es donde se capturan todos los detalles de la User, y asi probar si la implementación ha sido realizada de forma correcta y completa.

2) **Conversación:** la comunicación entre el Product Owner y el Equipo de desarrollo que permite compensar la falta de especificación y detalle. son discusiones acerca de la historia que sirven para desarrollar los detalles de la historia

3) **Confirmación:** Definición de un acuerdo entre el Product Owner y el Equipo para demostrar que la característica del producto realmente se implementó. El PO decide si las pruebas de aceptación son válidas. Pruebas que se usan para determinar cuándo una US está completa.

Product Backlog y las user stories

Es una pila de PBIs (Product Backlog Ítems) con sus correspondientes estimaciones de tamaño, en un formato determinado. En la parte más alta de la pila, se ubican las U.S. de mayor prioridad (que son aquellas U.S. con granularidad fina), y las de menor prioridad (normalmente aquellas User de granularidad gruesa, sobre las cuales no se tiene mucho detalle ya que no se consideran indispensables) al fondo. En cualquier momento se puede cambiar esa priorización, según cómo lo indique el Product Owner, quien es el encargado de realizar el proceso de priorización. En cualquier momento del proyecto se van agregando nuevas U.S. y actividades a realizar, según se vayan detectando nuevos requerimientos.

Se planifica cuántas US se van a sacar por iteración, según la **velocidad del equipo** (Story Points / Sprint)

Porciones verticales

Las US son porciones verticales, es decir, para poder agregar valor al cliente debemos tener un poco de interfaz, un poco de lógica de negocio y otro poco de base de datos.

Debemos diseñar el pedacito de cada cosa que nos hace falta para que una determinada característica funcione.

Tamaño

Es recomendable tener una gran cantidad de historias con tamaños relativamente pequeños, que tener historias muy grandes. El tamaño de las U.S. se mide en Story Points.

Hay 3 aspectos que determinan el tamaño de una U.S.:

- Complejidad: hace referencia a una dificultad de una funcionalidad, que no permiten dividirla en U.S. más pequeñas.
- Esfuerzo: es el tiempo en horas de trabajo que requerirá la implementación.
- Incertidumbre
    - Técnica: cuando se tiene dudas en el dominio de la solución
    - De negocio: cuando hay incertidumbre respecto a cómo interactuará el usuario con el sistema

Story Points

Son unidades de medida relativas (no importa el valor en sí, sino su comparación), las cuales miden el tamaño de un producto, NO de una medida basada en el tiempo (por ejemplo, horas hombre). Esto se debe a que estimar basándonos en el tiempo, conduce a muchos errores.

Diferentes niveles de abstracción

User Story: cumple con el criterio de Listo (DoR) para ingresar a una iteración.

Épica: es una User muy grande. El tamaño es relativo a si entra en una iteración o no, ya que las User se empiezan y se terminan en la iteración. Si la User no entra en la iteración es considerado una épica.

Temas: son un conjunto de User Stories agrupadas, debido a que conceptualmente están relacionadas.

Modelado de Roles

El foco está en entender **quién va a usar el sistema** y qué características tiene.

Se usan **tarjetas de rol de usuario** para anotar características generales de cada perfil.

Es muy importante porque **las funcionalidades están pensadas para esos roles**.

Técnicas adicionales

- Personas: Se describe una persona concreta que va a usar el sistema. Muy detallado.
- Personajes extremos: Se modelan usuarios en situaciones límite o poco comunes.
- Proxies (Usuarios Representantes): es un representante de todos los usuarios. Debe tener capacidad de tomar decisiones desde el negocio.

Criterios de aceptación

Define un acuerdo respecto a cómo se debe comportar el software para que el PO acepte la implementación. El criterio de aceptación es información concreta que tiene que servirnos a nosotros para saber si lo que implementamos es correcto o no. Debemos saber si el PO nos va a aceptar esta característica implementada o no, porque la va a aceptar si respeta los criterios de aceptación.

- Definen límites para una User Story (US)
- Ayudan a que los PO respondan lo que necesitan para que la US provea valor
- Ayudan a que el equipo tenga una visión compartida de la US.
- Ayudan a desarrolladores y testers a derivar las pruebas.

Pruebas de aceptación

Son declaraciones de intención de que hay que probar. Se prueban escenarios exitosos y escenarios que fallen. En las pruebas de aceptación se encuentran las más importantes, ya que no es posible escribir todas las posibles pruebas de aceptación. El Product Owner acepta la User como implementada si todas las pruebas de aceptación se cumplen.

DoR (Definition of Ready) – Definición de Listo

Es una medida de calidad que determina si la User Story está en condiciones de entrar a una iteración de desarrollo. Para que la US pueda ser implementada, mínimamente debe satisfacer el INVEST Model. Entre el equipo de desarrollo y el PO pueden definir características o condiciones extras al INVEST para que una US se considere lista para entrar a la iteración de desarrollo.

INVEST Model

Es un modelo de calidad que nos ayuda a verificar si la User cuenta con las características de la calidad mínima para satisfacer la definición de Ready y poder ingresar en una iteración de desarrollo.

- **Independiente:** la User es calendarizable e implementable en cualquier orden.

- **Negociable:** una user story no es un contrato rígido ni un documento detallado, sino una descripción breve y flexible de una necesidad del usuario.

- **Valuable:** Una US debe ser valorable para los clientes o usuarios, es decir, las U.S. deben aportar valor de negocio a quiénes estará destinado el producto.

- **Estimable:** la US debe contener la cantidad de información suficiente para estimar el tamaño, la complejidad y el esfuerzo necesario para realizarlo.

- **Small (Pequeña):** la US debe ser lo suficientemente pequeña como para ser finalizada en una iteración.

- **Testeable:** la US debe poder probarse con pruebas de aceptación

DoD (Definition of Done) – Definición de Hecho

Determina si la US está terminada. Son los criterios que establecen cuándo una User Story ha sido implementada de forma correcta y completa, de forma que puede ser mostrada al Product Owner, para que éste decida si se pasa a producción o no.

Spikes

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

Investment Themes (Temas de Inversión)

Representan un conjunto de iniciativas o propuestas de valor que conducen la inversión de la empresa en sistemas, productos, aplicaciones y servicios con el objetivo de lograr una diferenciación en el mercado y/o ventajas competitivas.

Estimaciones

Una estimación es una suposición informada sobre cuánto esfuerzo, tiempo o costo va a requerir hacer una parte del software. No es una promesa ni un compromiso, es solo una forma de ayudarnos a planificar mejor. La idea es poder decir si algo es viable, cuánto trabajo implica y cómo organizarlo.

Errores en las estimaciones

- Cuando no tenemos suficiente información sobre el software
- Cuando el proyecto esta mal definido.
- Cuando generemos imprecisión en el proceso de estimación.
- Cuando omitimos actividades necesarias para la estimación del proyecto.

¿Por qué estimamos?

Existen dos razones principales por las cuales se llevan adelante las estimaciones:

- Para predecir completitud.
- Para administrar riesgos.

Antes de que el proyecto comience, el líder del proyecto y el equipo de software deben estimar el trabajo que habrá de realizarse, los recursos que se requerirán y el tiempo que transcurrirá desde el principio hasta el final.

Métodos utilizados para estimar

1) **Basados en la experiencia**
    a. Datos históricos: consiste en recolectar datos básicos de otros proyectos para ir generando una base de conocimientos que sean de utilidad para futuras estimaciones. Esto no va acorde a la gestión ágil.
    b. Juicio experto: Existen dos técnicas para aplicarlo:
        i. Juicio experto puro: un experto estudia las especificaciones y realiza una estimación. Esta estimación se basa fundamentalmente en la experiencia del experto.
        ii. Juicio Wideband Delphi: es similar al anterior pero grupal, un grupo de personas se reúne con el objetivo de converger a una estimación común, tanto en esfuerzo como en duración.
2) **Analogía**
   Consiste en tomar a un proyecto e ir comparando factores, por ejemplo:
   - Tamaño: es menor o mayor al proyecto anterior.
   - Complejidad: más complejo de lo usual.
   - Usuarios: si hay más usuarios habrá más complicaciones

Estimaciones en ambientes agiles

En metodologías ágiles, el equipo de desarrollo (sin el Product Owner) es quien estima colectivamente para mayor precisión.

Errores comunes en estimaciones:

- Subestimar por miedo a que el cliente rechace un valor alto.
- Sobreestimar por precaución, inflando el esfuerzo real.

Las estimaciones no son compromisos ni planificación, sino una referencia que se ajusta con el avance del proyecto. El beneficio principal es el aprendizaje continuo: estimar mejora la capacidad del equipo para futuras estimaciones. No se debe gastar demasiado tiempo en estimar, ya que la precisión perfecta es costosa.

Consideraciones clave:

- Son medidas relativas, no absolutas.
- No se basan en tiempo, sino en comparación con otros trabajos.
- Favorecen la dinámica grupal y el pensamiento en equipo.

La Unidad de medida es el Story Point, que considera complejidad, esfuerzo e incertidumbre para comparar User Stories.

¿Qué se estima?

Lo que se estima en ambientes ágiles, es solo el tamaño de las US usando como unidad de medida los Story Points, y teniendo en cuenta 3 aspectos de estas: incertidumbre, complejidad y esfuerzo. se utilizan comúnmente series numéricas, siendo una de las más comunes, la serie de Fibonacci.

Es importante tener en claro que no se estima tiempo, ya que la duración de los Sprints es fija (concepto de Timebox). Por esto, es que no se recomienda estimar el tamaño de las US en horas, para no confundir con el esfuerzo

Tamaño vs esfuerzo

- Tamaño: Relacionado con la complejidad del proyecto, independiente de quién lo haga. la medida de cuán compleja y grande es una US
- Esfuerzo: las horas humano lineal requeridas para el desarrollo de la US. Por lo tanto, una US de mismo tamaño puede implicar esfuerzos diferentes en función de la persona que se esté evaluando

Formas de estimar el tamaño

Existen diferentes escalas para estimar el tamaño y siempre son por comparación, entre ellas:

- Número del 1 al 10.
- Talles de remeras: S, M, L
- Serie exponencial de base 2: 2, 4, 8, 16, 32, 64, etc. 4.
- Serie de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, etc.

Una vez definida la escala, no se cambia. En caso de cambiarla, también se debe cambiar el patrón de medición

Velocidad

Es una métrica del progreso de trabajo de un equipo. Se calcula (no se estima) sumando el número de Story Points asignados al estimar una US, que el equipo completa durante cada iteración, lo cual implica que la US se ha desarrollado y testeado por completo (DoD), y además el Product Owner la ha aceptado.

Si la US se completó parcialmente, no se cuentan sus Story Points en esa iteración.

¿Cómo se estima la duración del proyecto?

Si la estimación se realiza utilizando Story Points para medir la complejidad relativa de las User Stories, para determinar la duración de un proyecto se realiza la derivación tomando el número total de story points de sus US y dividiéndolas por la velocidad del equipo.

Planificación

En ambientes ágiles, es necesario que quien planifica el proyecto de desarrollo de software, sea capaz de analizar el contexto del proyecto cada cierto período y hacer aquellos ajustes que sean necesarios, para lograr los objetivos planteados.

Los equipos ágiles afrontan esta situación, planificando al menos en 3 niveles:

- **Planificación de Release:** Este proceso se da al inicio del proyecto, donde se analiza y determina qué US, épicas o Temas serán desarrollados en una nueva entrega (Release) del producto. Una vez que se priorizan las US a implementar, se estima su tamaño con Story points, y teniendo en cuenta la velocidad de trabajo del equipo, se deriva la duración de la Release o del proyecto.
- **Planificación de Iteración**: Este proceso se da al comienzo de cada iteración, donde el PO identifica aquellas US de mayor prioridad que deberán ser implementadas en la iteración, basándose en la iteración anterior ya finalizada.
- **Planificación del día:** Este proceso permite coordinar y sincronizar el trabajo del equipo, normalmente en reuniones diarias. Allí sólo se discuten las tareas a realizar durante el día

Poker Planning o Poker Estimation

Se basa en la colaboración del equipo para estimar su propio trabajo. El Product Owner participa como moderador, pero no estima.

Escala de Fibonacci para estimaciones:

- 1 o 1/2: Funcionalidad pequeña.
- 2 - 3: Funcionalidad pequeña a mediana.
- 5: Funcionalidad mediana.
- 8: Funcionalidad grande (se evalúa si puede dividirse).
- 13 o más: Necesariamente se divide, ya que no cabe en un sprint.

Procedimiento:

- Se define una lista de elementos a estimar.
- Se elige una User Story canónica como referencia.
- Cada miembro selecciona una carta con su estimación.
- Se revelan simultáneamente las estimaciones y se debaten diferencias.
- Si hay grandes discrepancias, se justifican y se vuelve a votar.
- Si las diferencias persisten, se acuerda un valor intermedio.

Consideraciones:

- La estimación debe incluir el criterio de Done, no solo la programación.
- Se trabaja en time boxing (tiempo limitado).
- Se estima solo lo necesario para evitar compromisos prematuros.

Ventajas:

- Permite una visión multidisciplinaria del equipo.

- Mejora la comprensión del dominio del proyecto.
- Las estimaciones individuales combinadas dan mejores resultados.
- Es una actividad dinámica y entretenida para el equipo.

Estimaciones en Lean

En Lean, el cual es más extremo que las metodologías ágiles, asume que las estimaciones son opcionales y en algunas ocasiones pueden convertirse en un desperdicio dado que tienen una alta probabilidad de no coincidir con la realidad, dado que cualquier variable que se modifica, modificara a la estimación también.

Diferencias entre estimaciones en tradicional y en ágil

1. 🔺 **Qué está fijo y qué se adapta (las tres dimensiones)**
   - **En ágil**:
     - Se fija el **tiempo** (por ejemplo, un sprint de 2 semanas).
     - Se fija el **equipo** (los recursos).
     - Lo que se **adapta** es el **alcance** (qué funcionalidades se llegan a hacer).
     - Se parte de una **visión general**, no de un producto 100% definido.
   - **En tradicional**:
     - Se fija el **alcance** (todo lo que hay que hacer).
     - Y a partir de eso se estiman el tiempo y los recursos.
     - El producto está completamente definido desde el inicio.
2. 🎯 **Precisión**
   - En tradicional se busca una estimación **precisa** porque de ahí se arma todo el plan.
   - En ágil no se busca precisión extrema porque sería **un desperdicio de esfuerzo** (según Lean). Se prefiere una estimación **rápida, útil y ajustable**.
3. 📏 **Tipo de estimación: absoluta vs. relativa**
   - **Tradicional**: estimación **absoluta** (por ejemplo: "esto va a tardar 40 horas").
   - **Ágil**: estimación **relativa**, comparando tareas entre sí (por ejemplo: "esta historia es el doble de grande que aquella").
4. 🕯️ **Quién estima**
   - En tradicional: estima el **jefe de proyecto** o un **experto** externo.
   - En ágil: estima **todo el equipo de desarrollo**, cada uno sobre su propio trabajo.
5. ⛔ **Confusión entre estimar y planificar**
   - En tradicional: muchas veces se **toma la estimación como un compromiso**, como si fuera una promesa fija.
   - En ágil: la estimación es **una probabilidad, no un compromiso**.

6. 🕓 Cuándo se estima

- En tradicional: se estima **al comienzo del proyecto**, después de definir los requerimientos y el diseño.
    - Si cambia el alcance, hay que **volver a estimar todo**.
    - Las primeras estimaciones pueden tener un **error de hasta 400%**, porque hay poca información y mucha incertidumbre.
- En ágil: se estima al **comienzo de cada sprint**, sobre historias ya priorizadas.
    - Eso permite que las estimaciones sean más certeras y actualizadas.

SCRUM

Scrum es un marco ligero que ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptables para problemas complejos. En pocas palabras, Scrum requiere un Scrum Master para fomentar un entorno donde:

- Un propietario del producto (Product Owner) ordena el trabajo de un problema complejo en un Product Backlog.
- El equipo de Scrum convierte una selección del trabajo en un Incremento de valor durante un Sprint.
- El equipo de Scrum y sus partes interesadas (stakeholders) inspeccionan los resultados y realizan los ajustes necesarios para el próximo Sprint.
- Repetir

Scrum se basa en el empirismo y el pensamiento Lean. El empirismo afirma que el conocimiento proviene de la experiencia y la toma de decisiones basadas en lo que se observa. El pensamiento Lean reduce los desperdicios y se centra en lo esencial.

Scrum emplea un enfoque iterativo e incremental para optimizar la previsibilidad y controlar el riesgo.

Pilares de Scrum

- **Transparencia:** El proceso y el trabajo emergentes deben ser visibles para aquellos que realizan el trabajo, así como para los que reciben el trabajo. La transparencia permite la inspección. La inspección sin transparencia genera engaños y desperdicios.

- **Inspección:** Los artefactos de Scrum y el progreso hacia objetivos acordados deben ser inspeccionados con frecuencia para detectar problemas indeseables. La inspección permite la adaptación. La inspección sin adaptación se considera inútil.

- **Adaptación:** Si algún aspecto de un proceso se desvía fuera de los límites aceptables o si el producto resultante es inaceptable, el proceso que se está aplicando o los materiales que se producen deben ajustarse. El ajuste debe realizarse lo antes posible para minimizar la desviación adicional.

El Equipo Scrum (Scrum Team)

El **equipo Scrum** esta compuesto por:

1. Scrum Master
2. Product Owner
3. Desarrolladores

No hay jerarquías ni sub-equipos, y se auto-organizan para completar el trabajo en cada **Sprint**.

Roles dentro del equipo Scrum:

- **Desarrolladores:**
  - Crean los incrementos de valor en cada Sprint.
  - Planifican el Sprint (**Sprint Backlog**).
  - Aseguran la calidad siguiendo la **Definición de Hecho**.
  - Adaptan su plan según el objetivo del Sprint.

- **Product Owner:**
  - Maximiza el valor del producto.
  - Gestiona el **Product Backlog**:
    - Define y comunica el **Objetivo del Producto**.
    - Organiza y prioriza los elementos del backlog.
    - Garantiza transparencia y comprensión del backlog.
- **Scrum Master:**
  - Facilita la adopción de Scrum en el equipo y la organización.
  - Ayuda al equipo a mejorar su **autogestión y productividad**.
  - Asegura el cumplimiento de los eventos de Scrum.
  - Capacita en la gestión empírica y colabora con partes interesadas.

El equipo Scrum trabaja en **Sprints** con un ritmo sostenible, asegurando incrementos valiosos en cada ciclo.

Eventos de Scrum

En Scrum, los eventos permiten la inspección y adaptación de los artefactos, promoviendo la transparencia y minimizando reuniones innecesarias. Todos los eventos ocurren dentro del **Sprint**, que tiene una duración máxima de **un mes** y comienza inmediatamente después del anterior.

1. El Sprint

Es el corazón de Scrum, donde se convierte una idea en valor. Dentro del Sprint se realizan todos los eventos:

- **Duración:** Máximo **un mes**.
- **Reglas:**
  - No se hacen cambios que comprometan el **Objetivo del Sprint**.
  - Se mantiene la calidad.
  - Se refina el Product Backlog según sea necesario.
  - Se puede renegociar el alcance con el Product Owner.
- **Cancelación:** Solo el **Product Owner** puede cancelarlo si el objetivo se vuelve obsoleto.

2. Planificación del Sprint (Sprint Planning)

Define el trabajo a realizar en el Sprint mediante la colaboración del equipo Scrum.

- **Duración:** Máximo **8 horas** para un Sprint de un mes (proporcionalmente menor para Sprints más cortos).
- **Preguntas clave:**
  1. **¿Por qué este Sprint es valioso?** → Se define el **Objetivo del Sprint**.
  2. **¿Qué se hará en el Sprint?** → Los desarrolladores seleccionan elementos del **Product Backlog**.
  3. **¿Cómo se hará?** → Se descomponen las tareas en unidades más pequeñas.

El resultado de esta planificación es el **Sprint Backlog**, que contiene el **objetivo, los elementos seleccionados y el plan de trabajo**.

3. Scrum Diario (Daily Scrum)

Reunión breve donde los desarrolladores inspeccionan el progreso y ajustan el plan. Solo participan los desarrolladores

- **Duración: Máximo 15 minutos**.
- **Objetivo:**
    - Revisar avances hacia el **Objetivo del Sprint**.
    - Identificar impedimentos.
    - Planificar el trabajo del día.

Este evento mejora la comunicación y la autogestión, reduciendo la necesidad de reuniones adicionales.

4. Revisión del Sprint (Sprint Review)

Se hace al final del Sprint con el equipo y las partes interesadas (clientes, usuarios, etc.).

- **Duración: Máximo 4 horas** para un Sprint de un mes.
- **Actividades:**
    - Presentación de lo logrado.
    - Discusión sobre cambios en el entorno.
    - Posibles ajustes en el **Product Backlog**.

Es una sesión de trabajo, no solo una presentación.

5. Retrospectiva del Sprint (Sprint Retrospective)

Permite al equipo analizar su desempeño y mejorar la calidad y eficiencia del proceso. Sirve para que el equipo se mire a sí mismo y mejore.

- **Duración: Máximo 3 horas** para un Sprint de un mes.
- **Objetivo:**
    - Que salió bien
    - Que se puede mejorar
    - Que cosas cambiaria para el próximo sprint

Este evento **concluye** el Sprint.

6. Refinamiento del Product Backlog

No es un evento formal, pero es una actividad continua en la que se revisa y prioriza el **Product Backlog** para mejorar la claridad y el detalle de los elementos.

Estos eventos estructuran Scrum, garantizando la inspección y adaptación constantes del producto y del proceso.

Artefactos de Scrum

Los artefactos de Scrum representan el trabajo o el valor generado dentro del marco de trabajo. Cada uno está diseñado para garantizar transparencia y permitir la inspección y adaptación. Además, cada artefacto tiene un **compromiso** asociado que ayuda a medir el progreso.

1. Product Backlog (Pila del Producto)

Es una lista emergente y ordenada con todo lo necesario para mejorar el producto. Es la única fuente de trabajo para el equipo Scrum.

- Los elementos que pueden completarse en un Sprint se consideran **listos** para su selección.
- El **refinamiento del Product Backlog** descompone y detalla los elementos para hacerlos más manejables.
- Los desarrolladores estiman el tamaño de los elementos, mientras que el Product Owner guía la selección.

🔹 **Compromiso: Product Goal (Objetivo del Producto)**
Es el estado futuro deseado del producto y guía el trabajo dentro del Product Backlog.

2. Sprint Backlog (Pila del Sprint)

Es el plan de trabajo específico del Sprint, compuesto por:

- **Sprint Goal (Objetivo del Sprint):** Define el propósito del Sprint.
- **Elementos seleccionados del Product Backlog:** Lo que se trabajará en el Sprint.
- **Plan de acción:** Cómo los desarrolladores ejecutarán el trabajo.

Se mantiene actualizado a lo largo del Sprint y permite inspeccionar el progreso en el **Scrum Diario**.

🔹 **Compromiso: Sprint Goal (Objetivo del Sprint)**
Proporciona enfoque y coherencia, permitiendo ajustes en el trabajo sin afectar el objetivo del Sprint.

3. Increment (Incremento)

Es el resultado tangible de un Sprint, un paso concreto hacia el **Objetivo del Producto**.

- Cada Incremento se suma a los anteriores y debe funcionar correctamente con ellos.
- Puede entregarse a las partes interesadas antes del fin del Sprint si está listo.
- Un Incremento solo se considera terminado si cumple con la **Definición de Hecho**.

🔹 **Compromiso: Definition of Done (Definición de Hecho)**
Es la descripción formal de cuándo un Incremento cumple con los estándares de calidad. Si un elemento no cumple con la **Definition of Done**, vuelve al Product Backlog para futuras iteraciones.

Timebox

Esto existe para que no se pierda más tiempo del necesario en las distintas tareas y para aprender a negociar en términos del alcance del producto y no del tiempo o los costos.

Si no se llega al final de un sprint, en vez de extender el tiempo, se entrega menos.

Herramientas de SCRUM

Tarjeta de tarea

Es una tarjeta que cuenta con 5 partes:

- Código del Ítem Backlog: es un identificador del ítem en el Product Backlog.
- Nombre de la actividad: suele ser una frase verbal que define qué es lo que hay que hacer.
- Iniciales del responsable de realizar la tarea.
- Número de día del sprint.
- Estimación del tiempo para finalizar la tarea.

Tablero

Tablero utilizado por el Equipo de Desarrollo en el cual se colocan las tarjetas de tareas a realizar durante el sprint. Se encuentra dividido en 5 secciones:

- 1) Story: el nombre de la historia.
- 2) To Do: aquí se encuentran las tareas por hacer en el Sprint.
- 3) Work In Process: aquí se encuentran las tareas que se están realizando.
- 4) To Verify: aquí se encuentran las tareas finalizadas que deben verificarse.
- 5) Done: aquí se encuentran las tareas terminadas: finalizadas y verificadas.

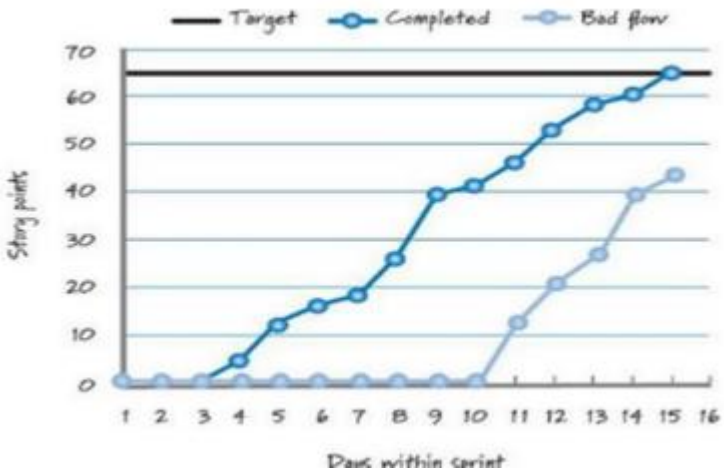
Lo ideal es que el nivel de granularidad de las tareas sea lo más fino posible, de tal forma que se detalle profundamente el avance del sprint, teniendo varias tareas en cada sección del tablero

Gráficos del Backlog

Gráficos que brindan información acerca del progreso de un Sprint, de una release o del producto. El backlog de trabajo es la cantidad de trabajo que queda por ser realizado (en horas), mientras que la tendencia del backlog compara esta cantidad con el tiempo (medido en días).

- Sprint Burn-Down Chart: Visualiza con una curva descendente cuánto trabajo queda para terminar. En el eje X se coloca el tiempo que va desde el inicio del sprint. En el eje y se coloca los puntos de historia a realizar en el sprint.

- Sprint Burn-Up Chart: Visualiza con una curva ascendente cuánto trabajo se ha completado a lo largo de la release, es decir que visualiza los puntos de historia terminados en cada sprint



- Combined Burn Chart: Visualiza cuánto trabajo queda para terminar y cuanto trabajo se ha realizado.

¿Qué es una métrica?

Una **métrica** es un número que usamos para **medir algo de manera objetiva y concreta**.
Puede ser algo del **proceso**, del **producto** o del **proyecto**.

Sirve para **Controlar** cómo va un proyecto, **Predecir** cuánto va a llevar algo (por ejemplo, en estimaciones), **Evaluar** costos, calidad, rendimiento, y sirve Para **mejorar** el proceso, el proyecto o el producto.

Características de una métrica válida:

- No es una escala cualitativa (Ej: NS, S, B, MB, E), sino la **cantidad** de cada categoría (3 de cada 5 usuarios hicieron clic).
- Debe ser medible en términos claros.
- Tiene un **costo asociado** en planificación, ejecución y análisis, así que **solo vale la pena si te sirve para algo concreto**.

Dominios de Métricas

Las métricas se dividen en tres dominios, cada uno con un enfoque distinto de medición:

1. Métricas de Proceso

- **Objetivo:** generar indicadores que permitan mejorar los procesos de software a largo plazo.
- **Características:**
  - Son estratégicas.
  - Son **públicas.**
  - Permiten evaluar la eficacia de un proceso e identificar áreas de mejora.
  - Son responsabilidad del **Ingeniero de Procesos**.
  - Apuntan a ver cómo trabaja toda la organización, no un solo equipo
- **Ejemplos:**
  - ¿Cuánto nos desviamos en las estimaciones, en general?
  - ¿Cuánto tiempo tardamos en planificar proyectos, en promedio?

2. Métricas de Proyecto

- **Objetivo:** Evaluar costos, esfuerzos, estimaciones y tiempos en un proyecto específico.
- **Características:**
  - Son privadas y solo accesibles a los involucrados en el proyecto.
  - Se utilizan para mejorar la planificación y reducir riesgos.
  - Permiten ajustes para optimizar calidad y minimizar costos.
  - Son responsabilidad del **Líder de Proyecto**.
- **Ejemplos:**
  - ¿El costo real fue igual al estimado?
  - ¿Las fechas reales coincidieron con lo planificado?

3. Métricas de Producto

- **Objetivo:** Medir la calidad y efectividad del software desarrollado.
- **Características:**
    - Se enfocan en el producto final y ver si se cumplio con los requerimientos.
    - Se utilizan para garantizar calidad y reducir defectos.
    - Son responsabilidad del **equipo de Desarrollo y Testing**.
- **Ejemplos:**
    - ¿Cuántas líneas de código tiene el producto?
    - ¿Cuántos métodos hay por clase?
    - ¿Cuántos casos de uso hay y cuán complejos son?

Métricas en el enfoque tradicional

En el enfoque tradicional se hace énfasis en los 3 dominios arriba mencionado. Se basan en la gestión de proyectos definidos.

Cuando se arma un proyecto, también se define:

- Qué métricas se van a usar.
- Quién va a ser responsable de medirlas.
- Cómo se van a calcular.
- Con qué frecuencia se revisan.

No todas las métricas son relevantes para todos los roles; la importancia de cada métrica varía según el perfil y el contexto.

Interes según el rol:

• Testers: Cuántos defectos se detectan y Cuánto esfuerzo lleva testear.

• Líder de proyecto: Se enfoca en Fechas de entrega, Costos y presupuesto, Plazos acordados con el cliente.

Las métricas de software en el enfoque tradicinal Son las mínimas e imprescindibles que uno tiene que utilizar si tiene la intención de desarrollar una cultura de medición:

• Tamaño del producto: asociada a métrica de producto. (ej: líneas de código, casos de uso)

• Esfuerzo: asociada a métrica de proyecto. (ej: horas trabajadas por persona o por actividad)

• Calendario: asociada a métrica de proyecto. (ej: fechas reales vs. planificadas)

• Defectos: asociada a métrica de producto. (ej: cantidad y severidad de errores detectados)

Estas métricas están relacionadas con la triple restricción en un proyecto (esfuerzo, alcance y tiempo).

Métricas en Ambientes Ágiles

El enfoque ágil prioriza la medición del **producto** sobre el proceso o el proyecto, siguiendo el principio de que "la mejor métrica de progreso es el software funcionando".

Principales métricas en Ágil:

- **Métrica de Velocidad (métrica del producto):**
    - Mide la cantidad de **puntos de historia** completados y aceptados por el Product Owner en un Sprint.
    - No se cuentan historias parcialmente terminadas.
    - Se representa con gráficos de barras para evaluar la estabilidad del equipo.
- **Métrica de Capacidad (métrica de proyecto):**
    - Mide cuántas **horas ideales de trabajo** tiene disponible el equipo en un Sprint.
    - Se usa para saber **cuánto se puede comprometer** el equipo en ese Sprint.
    - Se puede estimar en **Horas ideales** o también en **Story Points**
    - Se calcula al **inicio del Sprint**, durante la planificación.

- **Running Tested Features (RTF) (métrica de producto):**
    - Cuenta **cuántas funcionalidades están listas y funcionando** (es decir, completas y testeadas).
    - Se mide en cantidad absoluta: no importa si eran más o menos complejas.
    - Es útil para ver si **el producto avanza de verdad**.
    - Una curva plana o en descenso indica problemas en el desarrollo.

Métricas de software en Lean (KANBAN)

En Lean, y particularmente en **Kanban**, el foco de las métricas está puesto en el **proceso**, no en el producto ni en un proyecto cerrado.
¿Por qué? Porque **Kanban no trabaja por proyectos** con principio y fin, sino que se basa en un **flujo continuo de trabajo**.

Entonces, se mide **cómo se comporta ese flujo**:
→ ¿Cuánto tiempo tarda en fluir una tarea desde que entra hasta que sale?

- **Lead Time o Elapsed Time**
  Métrica de vista o **perspectiva del cliente**. Es la más importante para el cliente. Mide desde el momento en que el cliente me pide algo (entra al Backlog) hasta que yo se lo entrego.

- **Cycle Time**
  Mide el tiempo desde que el equipo comenzó a trabajar sobre una funcionalidad pedida, hasta que se lo entrega, eliminando el tiempo de espera en el Backlog. Sirve para saber cuánto tarda el equipo en **producir una funcionalidad** una vez que la empieza.

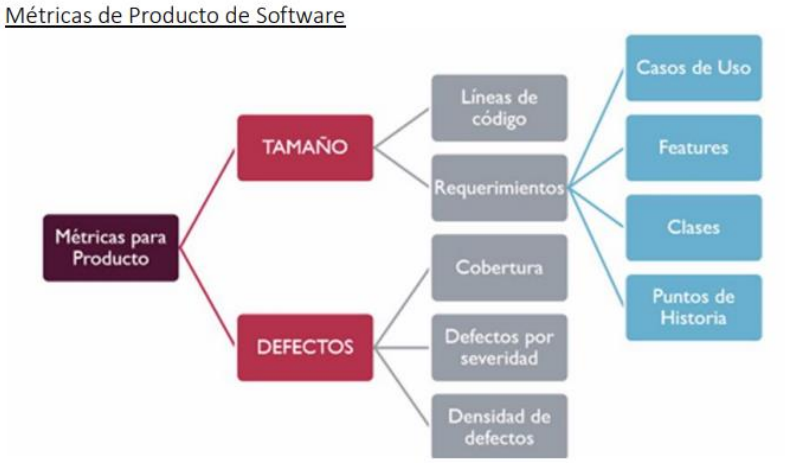
- **Touch Time**
  Mide los tiempos en los cuales las personas están efectivamente trabajando sobre una unidad de

trabajo, sin tener en cuenta aquellos tiempos de espera. Esta métrica muestra **cuánto tiempo real se invierte en hacer algo**, sin contar demoras.

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

- **Eficiencia del ciclo de proceso**
  Esta métrica mide la eficiencia del tiempo para entregar un trabajo, respecto al tiempo destinado a su ejecución. Se calcula como Touch time / Lead time. Mientras más cercano a 1 sea su valor, más eficiente es el proceso, ya que todo el tiempo se estuvo trabajando sobre alguna funcionalidad, y no hubo muchos momentos de espera en columnas de acumulación.

Métricas de producto de software



Gestión de Configuración de software

Definición

Es una disciplina de **soporte**, que **atraviesa todo el proyecto** y se aplica a distintas áreas.
Su objetivo principal es **garantizar la integridad del producto de software durante todo su ciclo de vida.**

Surge como una respuesta a la crisis del software, donde problemas como pérdida de componentes o superposición de cambios generaban muchos errores. SCM busca controlar esto de forma técnica y administrativa.

¿Qué busca resolver?

En el desarrollo de software pasan muchas cosas como:

- Se pierde código,
- Se pisan cambios,
- Se sube una versión vieja,
- Se arregla algo y después vuelve a romperse (regresión),
- Dos personas editan lo mismo al mismo tiempo,
- Se hacen cambios sin avisar o sin validar.

Como el software es muy fácil de modificar (¡a veces con un click!), se necesita **una gestión clara y controlada de todos los elementos importantes del sistema**.

¿Qué hace la GCS concretamente?

Aplica dirección y monitoreo (tanto **administrativo como técnico**) para:

1. **Identificar** los elementos clave del sistema (llamados *ítems de configuración*).
2. **Documentar** sus características funcionales y técnicas.
3. **Controlar los cambios** que se hagan sobre ellos.
4. **Registrar y reportar** los cambios.
5. **Verificar que el software siga cumpliendo lo que se espera** (esto se llama **trazabilidad**).

¿Qué significa “mantener la integridad”?

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

Se dice que un producto mantiene su **integridad** cuando:

1. **Cumple con lo que el usuario necesita.**
2. Tiene **trazabilidad**: podés rastrear todo lo que se cambió, cuándo y dónde impacta.
3. Cumple con criterios de **performance y requisitos no funcionales**.
4. **Cumple con las expectativas de costo**, incluyendo que el equipo esté satisfecho (no solo el cliente).

Ítem de Configuración (IC)

Es **cualquier artefacto del software** que:

- Forma parte del producto o proyecto.
- Puede cambiar con el tiempo.
- Necesitamos **rastrear** su evolución.

Ejemplos: código fuente, documentación, manual de usuario, plan de proyecto, plan de calidad, casos de prueba, scripts, etc.

Se gestionan porque forman parte de lo que entregamos y pueden impactar si cambian.

Versión

Instancia específica de un ítem de configuración en un momento determinado. Cada versión es única e identificable, permitiendo rastrear la evolución del software. Se utilizan atributos como números de versión para gestionar cambios y establecer relaciones entre versiones.

Variante

Es una configuración alternativa de un ítem de configuración que evoluciona de manera independiente. Se usa cuando se requieren versiones adaptadas a diferentes entornos. Por ejemplo, una versión para Windows y otra para Linux, o una con módulo premium y otra sin él. Se asocia con el uso de **branches** en repositorios de control de versiones.

Configuración

Es el **conjunto de ítems de configuración y sus versiones** en un momento determinado del proyecto.

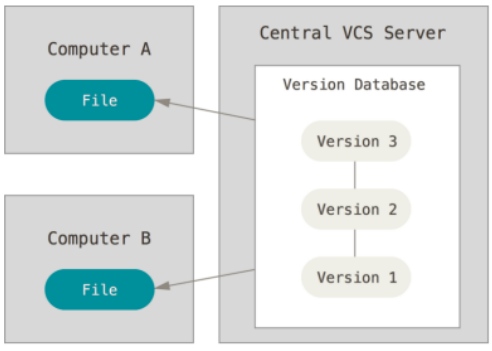
Por ejemplo: "la configuración entregada en la versión 1.2 del producto" incluye el código, los documentos y los datos con las versiones específicas de ese momento.

Incluye código fuente, documentos y estructuras necesarias para definir una versión del producto a entregar.

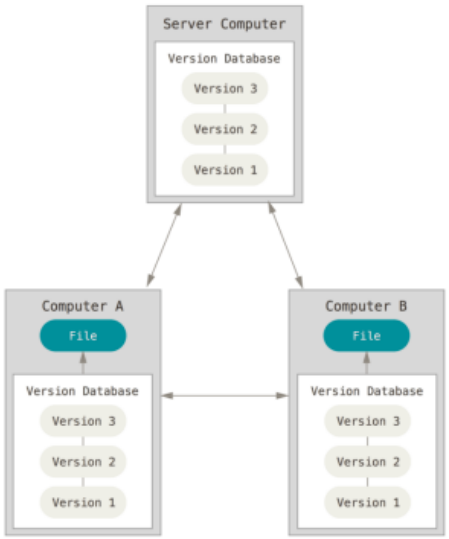
Repositorio

Sistema que almacena y gestiona los ítems de configuración, manteniendo su historial y estructura. Puede ser:

1. **Centralizado:** Un solo servidor almacena todas las versiones, con mayor control pero dependencia del servidor. Falla el servidor y "estamos al horno“



2. **Descentralizado:** Cada usuario tiene una copia completa del repositorio, evitando fallos únicos y permitiendo múltiples flujos de trabajo. Si un servidor falla sólo es cuestión de “copiar y pegar”



Release

Entrega oficial de una versión del software lista para ser utilizada por clientes o usuarios

Rama (Branch)

Línea de desarrollo separada para modificar o experimentar con el software sin afectar la versión principal. Las **ramas** permiten trabajar en paralelo al desarrollo principal

Integración de Ramas (Merge)

Proceso de combinar cambios de una rama con otra (generalmente hacia la principal) para mantener una versión estable del software. Es recomendable mantener la rama principal estable y fusionar cambios una vez que hayan sido verificados y aprobados.

Rol de las Líneas Base y su Administración

Línea Base

Una **línea base** es una **configuración estable del producto** que fue revisada formalmente y sobre la cual se llegó a un **acuerdo**.

Sirve como **punto de referencia** a partir del cual se desarrollan nuevas versiones del producto.

Se identifican mediante "tags" y se utilizan como punto de control en la evolución del proyecto.

Administración de Líneas Base

- Se identifican con etiquetas o marcas (por ejemplo: v1.0.0, release-candidato, etc.).
- Se registran en el repositorio como una configuración estable.

- Para cambiar una línea base, se sigue un protocolo formal de control de cambios gestionado por un **Comité de Control de Cambios**. Si se aprueba una modificación, se actualizan los parámetros y se comunica al equipo para tomar la nueva versión como referencia.

Utilidad de las Líneas Base

- Congelar un conjunto de ítems de configuración (IC) y sus versiones, en un momento determinado del proyecto.
- Sirven como punto de referencia estable.
- Facilitan el **rollback** en caso de errores.
- **Comparar versiones** (antes y después de un cambio).
- **Asegurar la trazabilidad** entre requisitos, código, pruebas, etc.
- En otras palabras, marca un "**hasta acá llegamos, esto es lo estable**" para seguir avanzando con seguridad.

Tipos de Líneas Base

1. **Operacionales**: Contienen versiones ejecutables del producto que han pasado controles de calidad. La primera línea base operacional coincide con la primera **release**.
2. **De Especificación o Documentación**: Son las primeras líneas base, ya que no contienen código, sino documentos clave como requerimientos y diseños del sistema.

Planificación de la gestión de configuración de software

Es el proceso de **definir cómo se va a aplicar la gestión de configuración en un proyecto específico**.

Este plan incluye:

- Herramientas que se van a utilizar
- Reglas de nombrado de los ítems de configuración
- Estructura del repositorio;
- Roles e integrantes del Comité de Control de Cambios
- Procedimientos formales de cambios
- Plantilla de formularios
- Procesos de auditoría

Actividades relacionadas a la gestión de configuración

Identificación de Ítems de Configuración

Esto implica una identificación unívoca para cada ítem

En el equipo se definirán políticas y reglas de nombrado y versionado para todos ellos.

También, se debe definir la estructura del repositorio, y la ubicación de los ítems de configuración dentro de esa estructura.

Además, implica una definición cuidadosa de los componentes de la línea base.

Se clasifican en:

Luego de identificar los ítems se debe armar la estructura del repositorio con nombres representativos.

Luego se vinculan los ítems de configuración con el lugar físico del repositorio donde se van a almacenar.

Control de cambios

Es el procedimiento formal para gestionar los cambios que se proponen sobre ítems de configuración ya identificados (especialmente los que están en línea base).

Comité de control de cambios: Este proceso es llevado a cabo por el comité de control de cambios, el cual se reúne para autorizar la creación y cambios sobre la línea base.

Etapas del proceso

1- Se recibe una propuesta de cambio sobre una línea base determinada, no sobre un ítem.
2- El comité de control de cambios realiza un análisis de impacto del cambio para evaluar el esfuerzo técnico, el impacto en la gestión de los recursos, los efectos secundarios y el impacto en la funcionalidad y la arquitectura del producto.
3- En caso de que se autorice la propuesta de cambio, se genera una orden de cambio que define lo que se va a realizar, las restricciones a tener en cuenta y los criterios para revisar y auditar.
4- Luego de realizado el cambio, el comité vuelve a intervenir para aprobar la modificación de la línea base y marcarla como línea base nuevamente, es decir que realiza una revisión de partes.
5- Finalmente se notifica a los interesados los cambios realizados sobre la línea base.

Auditorías de Configuración

Son controles realizados por un auditor externo al equipo de desarrollo para evaluar líneas base, asegurando la calidad e integridad del software. Estas auditorías permiten “congelar” el estado del software en un momento dado y verificar el cumplimiento de estándares.

Tipos de Auditoría

- Física o de contingencia: Se realiza primero. Valida que los ítems de configuración coincidan con la documentación y que el repositorio y la nomenclatura sean correctos.
- Funcional de configuración: Verifica que **la funcionalidad del producto cumple con los requerimientos definidos**. Solo se ejecuta si la auditoría física es exitosa.

Procesos Relacionados

- Validación: Confirma que cada ítem de configuración cumple con lo que el cliente necesita.
- Verificación: Asegura que el producto cumple con los objetivos de la línea base y que las pruebas son exitosas.

Informe de estado

Son **reportes que muestran el estado actual y la evolución** de los ítems de configuración y de la gestión en general. mantiene un registro de cómo evoluciona el sistema, y dónde está ubicado el software, comparado con lo que está publicado en la línea base.

Permite responder a preguntas como:

a. ¿Cuál es el estado del ítem?

b. ¿La propuesta de cambio fue aceptada o rechazada por el comité?

c. ¿Qué versión de ítem implementa un cambio de una propuesta de cambio aceptada?

d. ¿Cuál es la diferencia entre dos versiones de un mismo ítem?

Gestión de Configuración en Metodologías Ágiles

En contextos ágiles, la Gestión de Configuración (SCM) **se adapta**. Ya no busca imponer controles rígidos, sino **acompañar el trabajo del equipo de forma flexible y útil**. SCM en Agile se enfoca en que **el producto se mantenga sano, trazable y de calidad** a lo largo del tiempo.

¿Qué cambia con respecto al enfoque tradicional?

- En vez de que SCM sea una actividad separada y formal, **se integra al trabajo diario del equipo**.
- **No hay comités de control de cambios** ni auditorías pesadas como en enfoques tradicionales.
- El objetivo sigue siendo el mismo: **mantener la integridad del producto**, pero con prácticas más livianas y automáticas.

Características de la Gestión de Configuración en Agile

1. **Adaptabilidad a cambios:** No se busca evitarlos, sino gestionarlos eficazmente.
2. **Automatización:** Se implementan herramientas para optimizar la integración y entrega de software.
3. **Retroalimentación continua:** Se monitorea la calidad e integridad del software.
4. **Integración con otras tareas:** SCM no es una tarea separada, sino que forma parte natural del flujo de trabajo
5. **Responsabilidad compartida:** Todo el equipo participa en la gestión de configuración
6. **Uso de prácticas continuas**

Prácticas Continuas en Agile

Las **prácticas continuas** permiten optimizar el desarrollo y entrega del software, asegurando calidad y eficiencia.

1. Integración Continua (Continuous Integration - CI)

Es una práctica que consiste en que los desarrolladores integren su código en un **repositorio compartido** varias veces al día.

- **¿Cómo funciona?**
  - Cada vez que alguien hace un push, se ejecutan **pruebas automáticas**.
  - Si las pruebas fallan, se deben corregir de inmediato antes de seguir.
  - Se utiliza **TDD (Test-Driven Development)**, donde primero se escriben las pruebas y luego el código que las hace pasar.

🔹 **Objetivo:**
Garantizar que el código siempre sea funcional, evitando acumulación de errores y facilitando la colaboración entre desarrolladores.

2. Entrega Continua (Continuous Delivery - CD)

La **Entrega Continua** amplía la Integración Continua, asegurando que el software esté siempre listo para ser lanzado a producción.

🔹 **¿Cómo funciona?**

- Se automatizan pruebas de aceptación y validaciones.
- Se garantiza que la versión en desarrollo pueda ser desplegada en producción en cualquier momento.
- La **decisión de desplegar** sigue dependiendo del **Product Owner** o del equipo de negocio.

🔹 **Beneficio:**
El equipo puede lanzar nuevas funcionalidades rápidamente cuando sea necesario, sin demoras por problemas técnicos.

3. Despliegue Continuo (Continuous Deployment - CD)

El **Despliegue Continuo** lleva la entrega de software al siguiente nivel: cada cambio aprobado en el código se **despliega automáticamente** en producción sin intervención manual.

🔹 **¿Cómo funciona?**

- Se utiliza un **pipeline de despliegue** que ejecuta pruebas y verifica que el código está listo.
- Si todas las validaciones se cumplen, el software se actualiza automáticamente en producción.

🔹 **Diferencia con Entrega Continua:**

- En **Entrega Continua**, el equipo decide cuándo lanzar la versión.
- En **Despliegue Continuo**, la nueva versión se implementa automáticamente.

🔹 **Beneficio:**
Permite realizar lanzamientos frecuentes y reducir el tiempo de entrega al usuario final.

4. Estrategias de Deployment

Para minimizar riesgos y evitar interrupciones en la producción, existen estrategias de despliegue seguras:

🔹 **Canary Deployment**

- Se lanza la nueva versión solo para un **pequeño grupo de usuarios**.
- Si la versión es estable, se amplía a más usuarios. Si falla, se revierte rápidamente.
- **Ventaja:** Permite detectar errores con un impacto reducido.

🔹 **Blue-Green Deployment**

- Se mantienen **dos versiones** del software en producción:
    - **"Blue" (actual)**: Versión estable en uso.

    - **"Green" (nueva):** Versión en pruebas.
- Cuando la versión **Green** es validada, se redirige el tráfico a ella y la **Blue** queda de respaldo.
- **Ventaja:** Reduce el tiempo de inactividad y permite una reversión rápida en caso de fallos.

Aseguramiento de calidad de Proceso y de Producto

Calidad de software

En software, un producto tiene calidad cuando:

- Cumple con los requerimientos funcionales y no funcionales.
- Satisface al cliente, al usuario, al equipo y a la organización.
- Se puede mantener, escalar, usar y confiar en él.

¿Por qué hace falta la calidad en el software?

• Atrasos en las entregas

• Requerimientos no claros

• Falta cumplimiento de los compromisos

• Software que no hace lo que debería

• Costos excedidos

• Trabajo fuera de hora

• Fenómeno 90-90 (90% hecho 90% restante)

• No se aplica SCM.

Principios de Calidad

1. La calidad NO se inyecta, debe estar embebida en todo el proceso desde el inicio
2. Es una responsabilidad de todos los involucrados en el proyecto.
3. Las personas son la clave para lograrla: capacitación
4. Se necesita apoyo del nivel gerencial, pero se puede empezar por uno.
5. Se debe liderar con el ejemplo: Si los líderes no respetan los estándares, nadie lo hará.
6. No se puede controlar lo que no se mide: Hay que establecer métricas para saber si hay calidad o no.
7. Simplicidad, empezar por lo básico.
8. El aseguramiento de calidad debe planificarse.
9. El aumento de las pruebas no aumenta la calidad.
10. Debe ser razonable para mi negocio.

Visiones de Calidad

A la calidad se la puede analizar desde distintas perspectivas o visiones:

• **Visión del usuario:** Calidad es que el producto cumpla las expectativas del usuario final. Lo importante no es tanto si se cumple un estándar, sino si el usuario está satisfecho.
Ejemplo: Una app móvil que puede no ser perfecta técnicamente, pero el usuario la percibe como fluida, práctica, y “le hace la vida más fácil”.

• **Visión del producto:** La calidad está determinada por los atributos del producto que se pueden medir y observar.
Ejemplo: Un software que tiene un tiempo de respuesta menor a 0,5 segundos, no se cae ante alta carga, y es fácil de instalar.

• **Visión del proceso (manufactura):** Si un producto cumple con lo que fue diseñado o planificado, entonces es de calidad. Se focaliza en que el software siga estándares y normas técnicas.
Ejemplo: Una empresa exige que todos sus módulos tengan una cobertura del 85% en pruebas unitarias. Si eso se cumple, desde esta visión, el producto tiene calidad.

• **Visión del valor:** Calidad es encontrar un equilibrio en la relación costo-beneficio, para obtener siempre el mayor valor para el cliente posible y, obviamente generar ganancias con el desarrollo del software.
Ejemplo: Un sistema desarrollado con recursos limitados que cumple adecuadamente su función y resuelve un problema concreto a bajo costo.

• **Visión Trascendental:** Es aquello que hace que algo "se sienta bien hecho", excelente, superior. cuando el producto supera las expectativas: diseño visual, detalle, fluidez, consistencia.
Ejemplo: Un sistema que no solo cumple, sino que "impresiona": diseño elegante, respuesta perfecta, detalles cuidados, sin errores.

Calidad del Producto

Es la medida en que un producto de software **cumple con los requisitos funcionales y no funcionales esperados**, y **satisface a sus usuarios y partes interesadas**.

Modelos de Calidad de Producto

Dado que cada producto tiene distintos requerimientos, no existe un único modelo de calidad universal. Sin embargo, algunos modelos permiten medir ciertos aspectos de la calidad:

- **Modelo de Barbacci / SEI:** Evalúa calidad en base
    - performance
    - confiabilidad
    - Seguridad.
- **Modelo McCall:** Clasifica la calidad en tres factores
    - revisión del producto
    - operación del producto
    - transcision del producto.
- **ISO 25000:** Permite medir requisitos no funcionales (RNF) del software.

Estándares de Producto

Definen las características que deben cumplir los componentes del software. su objetivo es garantizar uniformidad y calidad en el producto final.

Calidad del Proceso

Es el grado en que los procesos usados para desarrollar y mantener software **están definidos, controlados, medidos y mejorados sistemáticamente**, asegurando que se produzcan resultados de calidad.

Para garantizar la calidad del proceso se implementan auditorías, revisiones técnicas y herramientas como SCM. Se deben definir procesos claros que sean conocidos por todo el equipo

Modelos de calidad para la Mejora de Procesos

La **mejora continua de procesos** implica analizar los procesos existentes y modificarlos para **incrementar la calidad del producto, reducir costos y optimizar tiempos**.

Se basa en la idea de que la **calidad del producto depende de la calidad del proceso**.

Los modelos de mejora de procesos **no indican cómo hacer las cosas**, sino que **describen** estrategias para mejorar los procesos dentro de una organización.

Modelo IDEAL (Initiating, Diagnosis, Establishing, Acting, Learning)

Es un **modelo cíclico** que guía la mejora de procesos dentro de una organización, asegurando cambios progresivos y sostenibles.

- **Fases del Modelo IDEAL:**
  1. **Inicialización:** Se identifican las necesidades de cambio y se busca apoyo organizacional (*sponsor*).
  2. **Diagnóstico:** Se analiza el estado actual de los procesos y los riesgos de mejora.
  3. **Establecimiento:** Se elabora un plan detallado con acciones específicas y prioridades.
  4. **Acción:** Se implementan los cambios en un **proyecto piloto** antes de extenderlos a toda la organización.
  5. **Aprendizaje:** Se evalúan los resultados y se ajustan los cambios para futuros ciclos de mejora.

Modelo SPICE (Software Process Improvement Capability Evaluation - ISO 15504)

Es un modelo diseñado para **evaluar y mejorar procesos de desarrollo de software** en base a **niveles de madurez**.

- **Características:**
  - Evalúa el nivel de madurez de los procesos en **6 niveles**.
  - Se enfoca en la **calidad del proceso** y su **capacidad de mejora**.
  - Se complementa con el modelo **IDEAL** para generar proyectos de mejora dentro de la organización.

Este modelo establece conjuntos predefinidos de procesos con objeto de definir un camino de mejora para una organización. En concreto, establece 6 niveles de madurez para clasificar a las organizaciones.

🔴 **Nivel 0 – Organización inmadura**

No hay procesos efectivos. Las actividades se hacen de forma informal o improvisada. No hay control.

🟠 **Nivel 1 – Organización básica**

Se logran los objetivos de los procesos, pero **sin un enfoque estructurado**. Se hace lo necesario, pero sin estandarización ni seguimiento formal.

🟡 **Nivel 2 – Organización gestionada**

Los procesos ya son **planificados, ejecutados y controlados**. Hay gestión del trabajo y de los resultados. Comienza a haber orden y documentación.

🔵 **Nivel 3 – Organización establecida**

Los procesos están **definidos, estandarizados y adaptados** a la organización. No solo se siguen, sino que **son parte de la cultura**.

🟣 **Nivel 4 – Organización predecible**

Los procesos se **miden y gestionan con métricas**. Se pueden anticipar resultados, hacer comparaciones y tomar decisiones con base cuantitativa.

🟢 **Nivel 5 – Organización optimizando**

La organización busca **mejora continua**. Se aprende de la experiencia, se innova y se ajustan los procesos para **alinearlos con los objetivos del negocio**.

Modelo de Calidad para Evaluar Procesos

Son modelos que **permiten evaluar cómo se están ejecutando los procesos en la práctica**, comparándolos contra una referencia teórica (modelo).
No te dicen cómo hacer las cosas, sino qué **resultados** o **niveles de madurez** deberías alcanzar.

Capability Maturity Model Integration (CMMI)

📌 Es el **modelo de referencia más reconocido** para evaluar procesos de software.

- Se basa en **buenas prácticas reales** de organizaciones exitosas.
- No dice *cómo* hacer las cosas, sino **qué objetivos alcanzar** en cada nivel.
- Las evaluaciones pueden ser **objetivas y subjetivas**, realizadas por entidades certificadas.

Constelaciones CMMI

CMMI se divide en tres constelaciones, cada una enfocada en un área específica:

- **CMMI-DEV (Desarrollo):** Guía para medir y gestionar procesos de desarrollo de software.
- **CMMI-ACQ (Adquisición):** Ayuda a gestionar la compra y subcontratación de productos o servicios.
- **CMMI-SVC (Servicios):** Facilita la gestión de la entrega de servicios internos o externos

Representaciones de CMMI-DEV

Existen dos formas de aplicar CMMI-DEV:

1. **Por Etapas:** Evalúa la madurez organizacional en cinco niveles. Para avanzar de nivel, se deben cumplir todos los requisitos de los niveles anteriores.
    - **Inicial:** No hay procesos definidos, el éxito depende del esfuerzo individual.
    - **Administrado:** Se gestionan requisitos y procesos, asegurando calidad y planificación.
    - **Definido:** Procesos bien definidos y documentados.
    - **Cuantitativamente administrado:** Uso de métricas y análisis estadísticos.
    - **Optimizado:** Mejora continua basada en innovación tecnológica.

2. **Continua:** Se mide la capacidad de cada área de proceso de forma individual, sin evaluar la organización completa. Ideal para empresas que desean mejorar áreas específicas sin adoptar CMMI en su totalidad

Estándares de Proceso

Establecen cómo deben ser implementados los procesos de software, incluyendo **definición de requerimientos, diseño, validación, pruebas y despliegue**. Estos estándares buscan estructurar el desarrollo para garantizar la calidad del software.

Definición de Procesos

Para asegurar la calidad, una organización debe **definir y comunicar su proceso de desarrollo**, incluyendo:

- **Etapas y subetapas** del desarrollo.
- **Roles y responsabilidades** dentro del equipo.
- **Momentos y métodos** para ejecutar cada tarea.

Además, un proceso de calidad debe integrar disciplinas **transversales**, como **planificación, seguimiento de proyectos, gestión de configuración (SCM) y aseguramiento de calidad (QA)**, sin importar si la metodología es **tradicional o ágil**.

El proceso debe **aportar valor al producto** y ser **adaptable a cada proyecto**, sin alterar aspectos clave como **Testing**, que es obligatorio. Se pueden aplicar modelos de mejora continua como **Kanban** para optimizar el desarrollo.

Aseguramiento de calidad

Es la definición de procesos y estándares que deben conducir a la obtención de productos de alta calidad.

La planeación de calidad es el proceso de desarrollar un plan de calidad para un proyecto. El plan de calidad debe establecer las cualidades deseadas de software y describir cómo se valorarán. Por lo tanto, define lo que realmente significa software de “alta calidad” para un sistema particular.

Los administradores de calidad deben enfocarse también a desarrollar una “cultura de calidad” en la que todo responsable del desarrollo del software se comprometa a lograr un alto nivel de calidad del producto.

funciones del Aseguramiento de Calidad de SW:

• Prácticas de aseguramiento de calidad.
• Evaluación de la planificación del proyecto de SW.
• Evaluación de requerimientos.
• Evaluación del proceso de diseño.
• Evaluación de las prácticas de programación
• Evaluación del proceso de integración y prueba del software
• Evaluación de los procesos de planificación y control de proyectos.
• Adaptación de los procedimientos de calidad para cada proyecto

Administración de la Calidad de Software

Es el conjunto de actividades que **busca garantizar que el producto cumpla con los niveles de calidad requeridos**, no por arte de magia, sino **definiendo y aplicando estándares, procesos y buenas prácticas** durante todo el desarrollo.

El objetivo es **instalar una cultura de calidad**, donde **todo el equipo esté comprometido**.

Actividades principales de la administración de calidad

1. Aseguramiento de Calidad (QA)

- Define:
    - Estándares
    - Procedimientos
    - Modelos de calidad (por ejemplo: qué atributos se esperan)

2. Planificación de la Calidad

- Se planifica:
    - **Qué actividades se van a hacer**
    - **Cuándo se van a hacer**
    - Qué **estándares se aplican en este proyecto específico**

3. Control de Calidad

- Es la **ejecución práctica** de lo que se planificó.
- Se asegura que el equipo respete los procedimientos.

Auditorías de Proyecto y al Grupo de Calidad

Las auditorías son evaluaciones independientes de productos o procesos de software para asegurar el cumplimiento de estándares, lineamientos y procedimientos. Representan una herramienta clave en el aseguramiento de calidad del software.

Beneficios de las Auditorías

- Permiten obtener opiniones objetivas e independientes.
- Ayudan a identificar áreas de insatisfacción del cliente.
- Aseguran el cumplimiento de expectativas y estándares.
- Ofrecen visibilidad sobre los procesos y su efectividad.
- Identifican oportunidades de mejora y fortalecen la toma de decisiones.

Tipos de Auditorías

1. **Auditoría de Proyecto:** Evalúa si un proyecto ha seguido el proceso definido, contrastando evidencias con documentación como la ERS, arquitectura y diseño. En metodologías ágiles, el propio equipo la realiza en la retrospectiva del Sprint.
2. **Auditoría de Configuración Funcional:** Compara el software con los requerimientos para asegurar que el código implementa correctamente lo especificado.
3. **Auditoría de Configuración Física:** Verifica que los ítems de configuración cumplan con la documentación técnica, asegurando trazabilidad y consistencia.

Roles en una Auditoría

- **Auditado:** Generalmente el Líder de Proyecto, quien propone la fecha, entrega evidencia y responde las preguntas del auditor.
- **Auditor:** Persona externa al proyecto, puede pertenecer al grupo de aseguramiento de calidad. Se encarga de definir el alcance, recolectar evidencias y realizar el informe.
- **Gerente de SQA:** Supervisa la auditoría, asigna recursos y resuelve no conformidades entre el auditor y el auditado.

Proceso de Auditorías

Etapas de una Auditoría

1. **Preparación y Planificación**: Se acuerda la fecha, alcance y objetivos de la auditoría. Se definen las métricas a evaluar.
2. **Ejecución**: Incluye una reunión de apertura, revisión de evidencia, análisis documental y entrevistas con los responsables.
3. **Análisis y Reporte de Resultados**: Se documentan los hallazgos, clasificándolos en buenas prácticas, observaciones o desviaciones.
4. **Seguimiento**: Se implementan planes de acción para corregir desviaciones y se monitorea su resolución.

Resultados y Hallazgos

- **Buenas Prácticas**: Implementaciones que superan los estándares establecidos.
- **Observaciones**: Posibles riesgos que aún no constituyen desviaciones.
- **Desviaciones**: Incumplimientos de procesos o estándares, que requieren corrección.

Herramientas y Técnicas Utilizadas

- **Checklists**: Listados de verificación con preguntas estándar para asegurar la consistencia de la auditoría.
- **Muestreo**: Se selecciona una muestra representativa de procesos o productos a evaluar.
- **Revisión de Registros**: Análisis de documentos históricos para verificar cumplimiento.
- **Herramientas Automatizadas**: Software especializado para análisis y monitoreo de calidad.

Métricas de Auditoría

Cada organización define sus propias métricas, algunas de las más comunes incluyen:

- **Esfuerzo por auditoría**: Tiempo invertido en la evaluación.
- **Duración de la auditoría**: Tiempo total del proceso de auditoría.
- **Cantidad de desviaciones**: Número total de hallazgos clasificados.

Revisiones técnicas

Las **revisiones técnicas** son una forma de **verificación** que permite **detectar defectos** en documentos o productos de software **antes de que se ejecuten o lleguen al cliente**.

Se aplican sobre cualquier producto de trabajo legible:

- Requisitos, diseño, código, casos de prueba, documentación, etc.

Detectar errores en las etapas tempranas del desarrollo para **evitar retrabajo y costos elevados** en fases posteriores.

Verificación y Validación

- **Verificación:** Asegura que el producto se construye correctamente, cumpliendo con los procesos definidos. ¿Estamos construyendo el producto correctamente?
- **Validación:** Confirma que se está construyendo el producto correcto, alineado con las necesidades del cliente. ¿Estamos construyendo el producto correcto?

El proceso abarca desde la revisión de requisitos hasta las pruebas finales, siguiendo principios como la prevención de errores, la retroalimentación constante y la priorización de lo rentable.

Tipos de Revisiones Técnicas

1. **Walkthrough o Recorrido (Informal):**
   Técnica en la que un diseñador o programador guía a otros a través del producto de software, permitiendo preguntas y comentarios. No hay proceso formal, se realiza en reuniones informales.
   Objetivos:
   - Mínima sobrecarga.
   - Capacitación de desarrolladores.
   - Rápido retorno.
2. **Inspecciones (Formales):**
   Proceso estructurado con un *checklist* y roles específicos. Permite recolectar métricas y elaborar un informe al final de la inspección.
   Objetivos:
   - Descubrir errores.
   - Verificar que el software cumpla sus requisitos.
   - Garantizar el cumplimiento de estándares.
   - Lograr un desarrollo uniforme del software.
   - Facilitar la manejabilidad de los proyectos.

Roles Participantes en Inspecciones Formales

1. **Autor:** Creador del producto, selecciona al moderador y entrega el producto a inspeccionar.
2. **Moderador:** Planifica y lidera la revisión, coordina la reunión y da seguimiento a los defectos.
3. **Anotador:** Toma nota de todos los defectos encontrados. Luego elabora un **informe formal** con los resultados de la reunión.

4. **Lector:** Lee el producto durante la reunión, guiando el análisis punto por punto. Su función es mantener el foco y que todos sigan el mismo ritmo.
5. **Inspector:** Examina el producto antes de la reunión para identificar defectos. Todos pueden ser inspectores.

Etapas del Proceso de Inspección

1. **Planificación:** El moderador define el lugar, la duración (máximo 2 horas) y los roles.
2. **Visión General (Opcional):** El autor describe el producto a inspeccionar.
3. **Preparación:** Cada rol revisa el producto para detectar defectos potenciales.
4. **Reunión de Inspección:** Se analizan los defectos, se registran los resultados y se concluye si el producto es aceptado.
5. **Corrección:** El autor realiza las correcciones necesarias.
6. **Seguimiento:** Dependiendo de la gravedad, se puede requerir una re-inspección.

Métricas

1. Densidad de defectos

Es la cantidad de defectos encontrados dividida por el tamaño del producto revisado (por ejemplo, líneas de código, páginas o ítems).

2. Esfuerzo de inspección

Es la suma del tiempo que invierten todos los participantes en la revisión (incluye preparación, reunión y seguimiento).

3. Esfuerzo por defecto encontrado

Se calcula dividiendo el esfuerzo total de inspección por la cantidad de defectos encontrados.

4. Porcentaje de reinspecciones

Es el porcentaje de inspecciones que tuvieron que repetirse porque no se solucionaron adecuadamente los defectos detectados.

Testing

Definición de Testing

- El **testing** es un proceso que somete al software a condiciones específicas para verificar si **cumple con los requerimientos**.
- Es una actividad **destructiva**, porque busca **encontrar defectos**, asumiendo que los hay.
- Sirve para ver si el software **hace lo que debe** y **no hace lo que no debe**.
- Es parte del **aseguramiento de la calidad (QA)**, pero **se aplica al producto ya construido**, mientras que QA se aplica **durante todo el ciclo de vida**.
- El testing es exitoso **cuando encuentra errores**. Si el software es de buena calidad, puede ser difícil encontrar defectos.
- Es **costoso**, por eso se debe buscar una **buena cobertura con criterio de costo-beneficio**.
- **Nunca se testea el 100%**, y la cobertura se planifica desde el inicio con los **criterios de aceptación**.

Principios del testing

1.**El testing muestra la presencia de defectos**, no su ausencia.

2.**El testing exhaustivo es imposible**, no se puede probar todo.

3. **El testing debe empezar temprano**, cuanto antes se detectan errores, menos cuesta corregirlos.

4.**Los defectos suelen agruparse**, hay partes del sistema más propensas a fallar.

5.**Paradoja del pesticida**: si siempre uso los mismos casos de prueba, dejan de ser efectivos.

6.**El testing depende del contexto**, no se prueba igual un sistema crítico que una app común.

7.**Falacia de la ausencia de errores**: que no haya fallas visibles no significa que el sistema sea bueno.

8.**Un programador debería evitar testear su propio código**, porque no lo va a ver con ojos críticos.

9.**Una unidad de desarrollo no debería probar lo que desarrolla**, es mejor que lo haga otro equipo.

10.**No alcanza con ver si hace lo que debe hacer**, también hay que ver si hace lo que no debería hacer.

11.**No planificar el testing pensando que no se van a encontrar errores**, hay que suponer que sí los habrá.

Mitos del testing

1.**"El testing consiste en probar que el software hace lo que debe hacer"**
→ Falso: también se debe verificar que **no haga lo que no debe hacer**.

2.**"El testing empieza cuando termina la codificación"**
→ Falso: el testing debe comenzar desde etapas tempranas del ciclo de vida (principio del testing temprano).

3. **"El testing garantiza la calidad del producto"**
→ Falso: el testing **detecta defectos**, pero **no crea calidad**. La calidad se construye durante todo el proceso.

4.**"El testing garantiza la calidad del proceso"**
→ Falso: eso lo hace el aseguramiento de la calidad (QA), no el testing.

5.**"El tester es el enemigo del programador"**
→ Falso: ambos deben **trabajar en equipo** para mejorar el producto.

¿Cuánto testing es suficiente?

- **Nunca se testea todo**, es imposible por tiempo y costo.
- Se testea **lo suficiente según el riesgo del sistema**.
- Se usa un **criterio de aceptación** para saber cuándo parar (por ejemplo: sin errores graves, % de tests pasados, costo aceptable).

Error vs Defecto

- **Error**: se detecta y corrige en la misma etapa.
- **Defecto**: es un error que llega a etapas posteriores (por ejemplo, testing).

Severidad vs Prioridad de un defecto

- **Severidad**: qué tan grave es técnicamente (bloqueante, crítico, mayor, menor, cosmético).
- **Prioridad**: qué tan urgente es arreglarlo para el negocio (urgente, alta, media, baja).

Casos de Prueba

Un **caso de prueba** es la unidad básica del testing. Sirve para comprobar que una funcionalidad del sistema se comporta como se espera. Son clave para **reproducir defectos** y se pueden reutilizar muchas veces.
Está compuesto por:

Objetivo: qué funcionalidad se quiere verificar.

Datos de entrada y ambiente: valores y condiciones necesarias antes de probar.

Resultado esperado: qué debería hacer el sistema si todo funciona bien.

Ciclo de Prueba

Es la ejecución completa de **todos los casos de prueba planificados** sobre una **misma versión del sistema**. Permite verificar el comportamiento del sistema antes de pasar a producción.
El **primer ciclo (ciclo 0)** suele ser manual y sirve para preparar el entorno. Luego pueden automatizarse los siguientes.

Ciclo de Prueba con Regresión

Después de corregir errores y generar una nueva versión, se deben **volver a ejecutar los casos de prueba anteriores**, incluso los que no fallaron, para asegurarse de que **los cambios no hayan roto otras partes**.
Esto se llama **testing con regresión**.
Lo ideal es hacerlo en todos los ciclos, pero en la práctica suele limitarse por tiempo y costos. Automatizar pruebas ayuda mucho a aplicar regresión.

Niveles de Prueba

Los niveles de prueba definen **qué tan grande es lo que se está testeando**, desde un componente aislado hasta el sistema completo en manos del usuario.

1. **Testing Unitario**

- Lo realiza el **desarrollador**, apenas termina de codificar un componente.
- Se prueba el **componente de forma aislada**, con acceso al código.
- Los errores se corrigen en el momento, **sin registrarlos formalmente**.

2. **Testing de Integración**

- Verifica que **los componentes funcionen bien juntos**.
- Se hace de forma **incremental**:
    - **Top-down**: de lo general a lo específico.
    - **Bottom-up**: de lo específico a lo general.
- Lo ideal es combinar ambos esquemas.

3. **Testing de Sistema**

- Se prueba el **sistema completo** como si estuviera en producción.
- Se validan **requerimientos funcionales y no funcionales** (como rendimiento, seguridad, recuperación ante fallas, etc.).
- El entorno debe **simular lo más fielmente posible** el entorno real.
- Se usan **casos de prueba** bien documentados.

4. **Testing de Aceptación**

- Lo hace el **usuario final**, para verificar que el sistema **cumple con sus necesidades**.
- El objetivo es generar **confianza y familiaridad**, no buscar errores.
- En ágil, esto ocurre durante la **Sprint Review**, con participación de todo el equipo.

Ambientes de Testing

Son los entornos (hardware + software) donde se desarrolla, prueba o ejecuta el sistema, y **varían según la etapa del proyecto**.

1. **Ambiente de Desarrollo**

- Lo usan los **desarrolladores** para programar y probar.
- Incluye IDEs, librerías, compiladores, etc.
- Se hacen pruebas **unitarias** y a veces de **integración**.

2. **Ambiente de Pruebas (Testing)**

- Lo usan los **testers** exclusivamente (los desarrolladores no deben acceder).
- Se hacen las **pruebas de sistema**.
- Está configurado para testear funcionalidades sin afectar al entorno real.

3. **Ambiente de Preproducción**

- Intenta ser **idéntico al entorno real de uso**.
- Se hacen **pruebas de aceptación**.
- Es difícil replicarlo exactamente (por costo o complejidad), pero se busca que sea lo más parecido posible.

4. **Ambiente de Producción**

- Es el entorno **real**, donde trabaja el usuario final.
- **No se prueba acá**, ya que los errores pueden afectar directamente al negocio.
- Solo se despliega una versión cuando ya pasó todas las pruebas y cumple con la **Definition of Done**.

Proceso de Pruebas

1. **Planificación**
   → Se arma el plan de pruebas: qué se va a probar, cómo, con qué recursos y cuál es el criterio para aceptar.
2. **Diseño**
   → Se definen y priorizan los casos de prueba, se preparan los datos y se analiza si los requerimientos son testeables.
3. **Ejecución**
   → Se corren los casos de prueba, se registran los resultados y se reportan los defectos encontrados.
4. **Evaluación y Reporte**
   → Se hace seguimiento de errores, se revisa si se cumple el criterio de aceptación y se comunica el resultado final.
5. **Fin de pruebas**
   → Se cierra el testing cuando no quedan errores graves. Se puede dejar un informe final.

Estrategias de Prueba

Sirven para **probar bien el sistema usando la menor cantidad posible de casos de prueba**, porque siempre hay límites de tiempo y presupuesto.
Lo ideal es **usar varias técnicas combinadas**, no solo una.

Caja Negra

No se ve el código, solo se prueban las entradas y salidas del sistema, como si fuera una "caja cerrada".

Técnicas:

- **Partición de Equivalencias**: se agrupan entradas parecidas que deberían dar el mismo resultado, y se prueba una por grupo.
- **Valores Límite**: se prueban los extremos válidos e inválidos de cada grupo (ej: 0, 1, 100).
- **Adivinanza de Defectos**: el tester usa su experiencia para probar donde cree que puede fallar.
- **Exploratorio**: se prueba libremente, sin seguir casos fijos, para descubrir fallas inesperadas.

Ideal para validar formularios, reglas de negocio y comportamiento general.

Caja Blanca

Sí se analiza el código. Se busca recorrer todos los posibles caminos que puede seguir el programa.

Técnicas:

- **Enunciados / Caminos Básicos**: cubrir todos los caminos independientes del flujo.
- **Sentencias**: ejecutar todas las líneas de código al menos una vez.
- **Decisiones**: probar tanto la rama verdadera como falsa de cada condición (if).
- **Condiciones**: probar cada condición dentro de una decisión en true y false.
- **Decisión-Condición**: combina ambas anteriores.
- **Cobertura Múltiple**: prueba todas las combinaciones posibles (muy completo pero costoso).

Usa **diagramas de flujo** y la **complejidad ciclomática** para saber cuántos caminos hay que cubrir.

Muy útil en pruebas internas del código, como en testing unitario.

Tipos de Prueba

- **Smoke Test**
  - Es una **primera prueba rápida** para ver si el sistema está en condiciones de seguir probándose.
  - Busca detectar **fallas graves o catastróficas** antes de arrancar el ciclo de pruebas formal (ciclo 0).
  - Si falla acá, no se sigue probando hasta corregir.
  - Se llama "smoke test" por una analogía con electrónica: si al prender algo sale humo, está todo mal.

- **Testing Funcional**
  - Prueba que el sistema haga lo que **se espera que haga** (el "qué").
  - Se basa en los **requerimientos funcionales** y procesos del negocio.
- **Tipos:**

- **Basado en Requerimientos**: prueba funcionalidades puntuales definidas (ERS, criterios de aceptación, etc.).
- **Basado en Proceso de Negocio**: prueba **todo un flujo completo**, como hacer una venta de principio a fin.

🔹 **Testing No Funcional**

- Evalúa **cómo** trabaja el sistema, no solo si hace lo correcto.
- Se enfoca en **requerimientos no funcionales**, como velocidad, estabilidad o facilidad de uso.

🔸 **Incluye:**

- **Performance**: velocidad y tiempos de respuesta.
- **Carga**: qué pasa con el hardware y el sistema al haber muchos usuarios o procesos.
- **Stress**: se fuerza el sistema más allá de lo normal, buscando que falle.
- **Mantenimiento**: facilidad para corregir errores o modificar el sistema.
- **Usabilidad**: que sea cómodo, entendible y práctico para el usuario.
- **Portabilidad**: que funcione en distintos entornos acordados.
- **Fiabilidad**: que se pueda confiar en el sistema y no falle.
- **Interfaz de usuario (UI)**: pruebas sobre pantallas, formularios, botones, etc.
- **Configuración**: se prueba el comportamiento ante distintos ajustes del sistema.

Modelo en V

Es un modelo que **relaciona cada etapa del desarrollo con su etapa de prueba correspondiente**, mostrando **cuándo** se deben hacer las pruebas.

Mientras el desarrollo va **de lo general a lo específico**, el testing va **de lo específico a lo general** (granularidad inversa).

Relación desarrollo – testing:

- Requerimientos → Pruebas de aceptación
- Diseño del sistema → Pruebas de sistema
- Diseño detallado → Pruebas de integración
- Implementación (código) → Pruebas unitarias

Testing Ágil

Testing Ágil es un enfoque de pruebas de software que acompaña al desarrollo ágil desde el primer momento. No se trata de una fase separada al final del ciclo, sino de una actividad continua e integrada en cada iteración. El objetivo principal no es solo encontrar errores, sino **prevenirlos** desde etapas tempranas, aportando valor rápidamente al producto y mejorando la calidad en equipo.

Se caracteriza por:

- Participación activa de todos (desarrolladores, testers y usuarios).
- Feedback constante y rápido.
- Pruebas automatizadas y manuales combinadas.
- Validación temprana y continua de funcionalidades.

Diferencias entre Testing Tradicional y Testing Ágil

En el enfoque **tradicional** (modelo en cascada), el testing se realiza al final del proceso de desarrollo. Primero se definen los requisitos, se diseña, se desarrolla y luego se prueba. Esto hace que los errores se detecten tarde, aumentando el costo de corregirlos. Además, los testers suelen estar separados del equipo de desarrollo.

En cambio, en **ágil**, el testing es continuo, se realiza desde el inicio y en cada iteración del desarrollo. Los testers trabajan junto al equipo de desarrollo, ayudando a **prevenir defectos**, no solo a encontrarlos. El foco está en los requerimientos del usuario y en generar confianza en el producto mediante feedback constante.

Otra diferencia clave está en la **pirámide de testing**:

- En tradicional predomina el testing GUI (de interfaz), que es costoso y frágil.
- En ágil se prioriza el testing unitario automatizado, que permite detectar errores rápidamente, seguido por pruebas de aceptación y en menor medida pruebas GUI.

Cuadrantes del Testing Ágil

Esta matriz ayuda a clasificar los distintos tipos de pruebas según dos ejes:

- **¿Para quién está orientada la prueba?**
    - Orientada al negocio (cliente, usuario)
    - Orientada a la tecnología (desarrolladores, sistema)
- **¿Cuál es su propósito?**
    - Apoyar al equipo
    - Criticar (evaluar) el producto

Cuadrante 1 (Q1): Soporte técnico al equipo, orientado a tecnología

Objetivo: Asegurar que el código funciona correctamente desde el principio.
Tipo de pruebas:

- Tests unitarios
- Tests de componentes

Son pruebas técnicas, automatizadas, rápidas y frecuentes. Ayudan a detectar errores mientras se programa.

Cuadrante 2 (Q2): Soporte al equipo, orientado al negocio

Objetivo: Validar que el software hace lo que el negocio necesita.
Tipo de pruebas:

- Tests funcionales
- Historias de usuario con ejemplos

- Prototipos y simulaciones

Son pruebas que suelen crearse junto al Product Owner. Pueden automatizarse o hacerse manualmente.

Cuadrante 3 (Q3): Evaluación del producto, orientado al negocio

Objetivo: Explorar y evaluar el producto desde la perspectiva del usuario final.
Tipo de pruebas:

- Pruebas exploratorias
- Escenarios
- Pruebas de usabilidad
- Pruebas de aceptación de usuario (UAT, alfa, beta)

Generalmente son **manuales** y buscan detectar problemas que afectan la experiencia del usuario.

Cuadrante 4 (Q4): Evaluación técnica del producto, orientado a tecnología

Objetivo: Evaluar cualidades no funcionales del sistema.
Tipo de pruebas:

- Pruebas de rendimiento y carga
- Pruebas de seguridad
- Pruebas de "-ilities" (como confiabilidad, mantenibilidad, escalabilidad, etc.)

Estas pruebas se hacen con **herramientas especializadas** y se enfocan en aspectos como velocidad, estabilidad, etc.

Principios del Testing Ágil

1. El testing comienza desde el inicio del proyecto
2. Testing no es una fase, es una actividad continua
3. Todo el equipo participa del testing
4. Feedback rápido ante los errores
5. Las pruebas reflejan lo que el usuario espera
6. El código debe mantenerse limpio y los errores corregirse rápido
7. Menos documentación, más pruebas claras y útiles
8. Una funcionalidad está "terminada" solo si pasa sus pruebas
9. Se prueba desde el comienzo y las pruebas guían el desarrollo (TDD, criterios de aceptación)

Prácticas comunes de Testing Ágil

1. Pruebas de unidad e integración automatizadas

Se escriben tests automáticos para comprobar que cada parte del sistema (unidad) y cómo se relacionan entre ellas (integración) funcionen bien. Esto permite detectar errores rápidamente mientras se desarrolla.

2. Pruebas de regresión a nivel de sistema automatizadas

Se automatizan pruebas completas del sistema para asegurar que al hacer cambios no se rompa nada que antes funcionaba. Esto se repite en cada iteración o Sprint.

3. Pruebas exploratorias

Se prueba sin seguir un caso predefinido, usando la intuición y experiencia del tester. Es útil para encontrar errores que se escapan en pruebas estructuradas.

4. TDD (Test Driven Development)

Primero se escribe una prueba unitaria **antes** de escribir el código. Luego se programa lo justo para pasar esa prueba. Se repite este ciclo mejorando el código. Es una forma de diseñar software guiado por las pruebas.

5. ATDD (Acceptance Test Driven Development)

Parecido a TDD, pero se enfoca en **pruebas de aceptación**, que se escriben con el usuario o Product Owner antes de programar. Así todos entienden qué se espera del sistema desde el principio.

6. Control de versión de las pruebas con el código

Las pruebas forman parte del proyecto y se versionan junto al código fuente, usando herramientas como Git. Así se asegura que el código y sus pruebas estén siempre sincronizados.

TDD – Test Driven Development

Es una técnica de desarrollo donde **primero se escriben las pruebas** (principalmente **unitarias**) y recién después se escribe el código para que esas pruebas pasen.

Se basa en dos prácticas principales:

1. **Test First Development**
    - Se escriben **pruebas unitarias antes del código**.
    - Eso obliga a **entender bien qué debe hacer el sistema** antes de implementarlo.
    - Ayuda a que el código esté **modularizado** (una función para cada cosa).
    - También puede aplicarse a pruebas de integración.
2. **Refactoring**
    - Una vez que el código pasa las pruebas, se **mejora internamente** sin cambiar lo que hace. Se busca hacerlo más **claro, simple y mantenible**

Ciclo de TDD (conocido como Red-Green-Refactor):

1. **Red**: Escribir un test que debe fallar (porque el código aún no está implementado).
2. **Green**: Escribir el código mínimo necesario para que el test pase.
3. **Refactor**: Mejorar el código manteniendo todos los tests en verde (pasando correctamente).

Las 3 leyes de TDD (según Robert C. Martin):

1. No escribir código de producción sin antes haber escrito un test que falle.
2. No escribir más de un test que falle (mantenerlo simple).
3. No escribir más código del necesario para pasar ese test.

Kanban

Kanban no es:

- Un proceso de desarrollo de software.
- Una metodología de gestión de proyectos.

Entonces, ¿qué es?

Kanban es un enfoque para gestionar procesos de trabajo con el objetivo de lograr una mejora continua. Se basa en el principio de **"empezar por donde estés"**, lo que significa que no exige cambios bruscos o disruptivos, sino que **introduce mejoras graduales** sobre procesos existentes. Esta estrategia **reduce la resistencia al cambio** y fomenta la **evolución progresiva** del trabajo en equipos o en organizaciones.

Aunque **surgió en Toyota** en los años 40 para mejorar el almacenamiento y el stock en supermercados (**origen Just in Time**), luego fue **adaptado al desarrollo de software y servicios de conocimiento**, donde intervienen la creatividad, el diseño y la colaboración.

Valores de Kanban

- Transparencia
- Equilibrio
- Colaboración
- Foco en el cliente
- Flujo
- Liderazgo
- Entendimiento
- Acuerdo
- Respeto

Prácticas de Kanban

1. Visualizar el trabajo

¿Qué significa?
Consiste en **hacer visible todo el trabajo** que se está realizando, en qué etapa está, y cuánto hay en curso. Esto se logra mediante **un tablero Kanban**, donde:

- Cada tarjeta representa una unidad de trabajo (U.S., feature, bug, etc.).
- Las columnas del tablero representan las **etapas del proceso**, por ejemplo: *To Do* → *In Progress* → *Testing* → *Done*.
- Se diferencia entre:
    - **Columnas de producción** (trabajo activo).
    - **Columnas de acumulación** (trabajo listo para avanzar, pero esperando).

¿Para qué sirve?

- Mejora la **transparencia**.
- Ayuda al equipo a **coordinar, tomar decisiones y detectar cuellos de botella** fácilmente.

2. Limitar el WiP (Work in Progress)

¿Qué significa?
Consiste en **poner un límite explícito** a la cantidad de trabajo que puede estar en curso en cada columna.

¿Para qué sirve?

- Evita que se inicie demasiado trabajo a la vez (causa de ineficiencia).
- Obliga al equipo a **finalizar lo que ya comenzó antes de comenzar algo nuevo**.
- Facilita el flujo constante (flujo "pull", no "push").
- Disminuye tiempos de entrega y mejora la calidad.

Ejemplo: Si en “En desarrollo” hay un límite de 3 tareas, no se puede agregar una cuarta hasta que una termine.

3. Gestionar el flujo de trabajo

¿Qué significa?
Implica **observar, medir y mejorar el flujo de trabajo** de manera continua, detectando:

- Cuellos de botella.
- Recursos ociosos.
- Tiempos de espera y entrega.

¿Para qué sirve?

- Lograr que el flujo sea **constante, estable y predecible**.
- Aumentar la eficiencia y satisfacción del cliente.

Dato importante:
En Kanban no hay iteraciones como en Scrum. El tablero es **flujo continuo**, y puede ser compartido por varios proyectos o equipos.

4. Hacer explícitas las políticas

¿Qué significa?
Las **reglas del proceso deben estar claras, visibles y accesibles** para todos los involucrados. Estas políticas pueden incluir:

- Qué condiciones debe cumplir una tarjeta para moverse a otra columna (**DoD**).
- Qué hacer si se rompe una regla (por ejemplo, exceder el WiP).

- Cómo se prioriza el trabajo.
- Cuándo reponer trabajo en el tablero.

¿Para qué sirve?

- Reduce la ambigüedad.
- Mejora la autonomía del equipo.
- Fomenta la mejora continua y adaptación rápida.

5. Mejorar colaborativamente y evolucionar experimentalmente

¿Qué significa?
Kanban fomenta una **cultura de mejora continua**, basada en:

- **Cambios graduales, no disruptivos.**
- Experimentación basada en hipótesis (como en el método científico).
- Colaboración en todos los niveles.

¿Cómo se mejora?

- Observando el flujo real.
- Proponiendo mejoras visibles y pequeñas.
- Ajustando políticas, procesos o límites según los resultados.

6. Circuitos de retroalimentación

¿Qué significa?
Implica **tener reuniones periódicas** para reflexionar sobre el proceso y tomar decisiones de mejora.

¿Qué se debe definir?

- Frecuencia adecuada (ni muy seguido ni muy espaciado).
- Participantes, objetivos y duración.

¿Para qué sirve?

- Asegura un proceso controlado y evolucionado.
- Permite ajustar el sistema basándose en evidencia del trabajo real.

¿Cómo aplicar Kanban?

1. **Empezar con lo que hay**
   Se respeta el proceso actual, sin hacer cambios bruscos.
2. **Definir unidades de trabajo**
   Qué se va a gestionar: historias de usuario, defectos, cambios, etc.
3. **Identificar clases de servicio**
   Diferentes tipos de trabajo tienen distintas políticas (urgente, estándar, etc.).

4. **Visualizar el flujo de trabajo**
   Se arma un tablero con columnas que representan las etapas del proceso.

5. **Definir políticas claras**
   DoD, límites de WiP, colores por clase, reglas visibles y simples.

6. **Detectar cuellos de botella y resolverlos**
   Ajustando recursos, límites o reglas para mejorar el flujo.