RESUMEN INGENIERIA Y CALIDAD DE SOFTWARE

RESUMEN INGENIERIA Y CALIDAD DE SOFTWARE

¿QUÉ ES EL SOFTWARE?

El **software** no se limita al código; es un **conjunto de programas y documentación**. No es suficiente saber programar; se necesita comprender la totalidad del sistema para crear software de calidad.

TIPOS BÁSICOS DE SOFTWARE

1. **Software de Sistemas**: **Controla y gestiona los recursos de hardware** y **proporciona una interfaz** para que los usuarios interactúen con la computadora.
2. **Utilitarios**: Son un subconjunto del software de sistemas y **ofrecen herramientas adicionales** para mejorar el funcionamiento del **sistema operativo** y el **rendimiento** de la computadora.
3. **Software de Aplicación**: Se utiliza para realizar **tareas específicas**, como procesar texto, crear presentaciones o navegar por internet, centrándose en **satisfacer las necesidades de los usuarios**.

SOFTWARE ≠ MANUFRACTURA

El software es diferente de una línea de producción debido a varias razones:

1. **Menos Predecible**: El software **no es tan predecible** como los productos tangibles en una línea de producción, ya que su desarrollo involucra **factores más variables y complejos.**
2. **Diversidad de Requisitos**: **Cada producto de software tiende a ser único** debido a las variaciones en los requisitos y necesidades de los usuarios.
3. **Naturaleza de Errores**: **No todas las fallas** en el software **son errores**; su tratamiento difiere significativamente en comparación con la producción de productos físicos.
4. **No se Desgasta**: A diferencia de los productos físicos que se desgastan con el tiempo, el software **no envejece de la misma manera**. Se adapta a los cambios y necesidades sin experimentar desgaste físico.
5. **Sin Limitaciones Físicas**: El software **no está sujeto a las limitaciones físicas**, ya que no está gobernado por las leyes de la física, lo que significa que su creación y modificación no se ven restringidas por consideraciones físicas.

Las diferencias entre el software y la producción de productos físicos están estrechamente relacionadas con la **intangibilidad del software.**

PROBLEMAS AL DESARROLLAR/CONSTRUIR SOFTWARE

1. **Desviaciones en Tiempo y Costos**: Con frecuencia, los proyectos de desarrollo de **software exceden los tiempos y costos** presupuestados.
2. **Insatisfacción del Cliente**: La versión final del software a menudo **no cumple con las necesidades y expectativas** del cliente, lo que lleva a la insatisfacción.
3. **Escalabilidad y Adaptabilidad Limitadas**: **Agregar nuevas funciones o realizar cambios** en versiones posteriores del software **puede ser difícil** debido a problemas de escalabilidad y adaptabilidad.
4. **Documentación Inadecuada**: La **documentación desactualizada o insuficiente** puede generar inconsistencias en la gestión y el mantenimiento del software.
5. **Calidad del Software Deficiente**: La **calidad** del software **no se relaciona únicamente con la cantidad de pruebas realizadas**, lo que puede llevar a la entrega de software de baja calidad.

SOFTWARE EXITOSO

El éxito y la rentabilidad en el desarrollo de software se logran al incorporar factores clave identificados por metodologías ágiles como **Scrum y Lean**. Estos factores incluyen la **claridad** y la **evolución de los requerimientos**, la **participación del usuario**, la **retroalimentación continua** y **equipos competentes** con habilidades adecuadas involucrados en el proceso de desarrollo. Estas premisas se integran en las nuevas formas de trabajo para **garantizar el éxito** en el desarrollo de software.

SOFTWARE NO EXITOSO

El éxito del software se ve comprometido cuando se presentan problemas como **requerimientos incompletos** o **cambiantes, falta de involucramiento del usuario, recursos insuficientes, expectativas poco realistas y falta de apoyo de la gerencia.** Estos desafíos demuestran que no basta con saber programar; se requieren múltiples facetas y características para lograr un software exitoso.

EL PROCESO DE SOFTWARE

Un proceso de software es un **conjunto estructurado de actividades** que, a partir de **diversas entradas**, **genera una salida**. Estas actividades varían según la organización y el tipo de sistema que se está desarrollando. Para gestionarlo adecuadamente, es esencial **modelar explícitamente el proceso**. Los **inputs incluyen requerimientos, personas, materiales, energía y otros recursos**, y el **objetivo es crear un producto de software valioso.**

Según el IEEE, un proceso es una secuencia de pasos ejecutados para un propósito dado, pero esta definición es general y poco precisa. La definición del CMM (Capability Maturity Model) es más completa y clara, definiendo el proceso de software como un conjunto de actividades, métodos, prácticas y transformaciones utilizadas para desarrollar o mantener software y sus productos asociados.

PROCESO DEFINIDO

Los enfoques **basados en una concepción industrial**, inspirados en líneas de producción, suponen que un **conjunto de inputs** se utilizará en un **conjunto de actividades** para producir **resultados consistentes y predecibles**, siempre que los inputs sean los mismos. Sin embargo, esta idea es **difícilmente aplicable** al software debido a su naturaleza no tan definible y única. El software no se puede simplificar a una entrada que siempre produzca el mismo resultado, y no se puede repetir el proceso indefinidamente para obtener resultados idénticos. La administración y el control de proyectos de software a menudo se basan en la predictibilidad del proceso, pero esta suposición no siempre es válida en el desarrollo de software. Un ejemplo de esto es el Proceso Unificado de Desarrollo (PUD).

PROCESO EMPIRICO

Los enfoques empíricos en el desarrollo de software **se centran en la experiencia y el aprendizaje** de los actores involucrados en la producción para **maximizar la calidad del producto**. A diferencia de los procesos definidos, los enfoques empíricos confían en que no se pueden definir ni controlar completamente todas las variables debido a la complejidad del entorno. La adaptación es clave para garantizar la calidad del software.

En estos enfoques, **la mejora** no se centra en un punto específico como en los procesos definidos, sino que **abarca varios aspectos**, siempre poniendo énfasis en **las personas que capitalizan la experiencia**. Se utiliza un ciclo de retroalimentación para adaptarse y mejorar continuamente a partir de la experiencia acumulada.

El patrón de conocimiento de procesos empíricos **sigue un ciclo** de **hipótesis (Asumir), construcción (Construir), medición e inspección (Revisar), retroalimentación (Retroalimentar) y adaptación (Adaptar).** Este ciclo se basa en la repetición y se utiliza en procesos empíricos, como el ciclo de vida iterativo e incremental, para mejorar continuamente el proceso de desarrollo de software.

CICLOS DE VIDA

El ciclo de vida es una abstracción que define **las etapas y el orden en las que progresa un producto o proceso**. No especifica quién, qué ni cómo se hace, ni las herramientas y procedimientos, solo establece las fases y su secuencia. Tanto los productos como los proyectos tienen ciclos de vida, que son una serie de pasos a seguir en su desarrollo.

CICLOS DE VIDA DE PROYECTOS DE SOFTWARE

Representan el **proceso de desarrollo de un proyecto** de software y definen las fases y su secuencia. Estos ciclos de vida **empiezan y terminan con el proyecto** en sí.

CICLOS DE VIDA DE PRODUCTO DE SOFTWARE

Comienzan cuando un proyecto genera un producto de software y **continúan a lo largo del tiempo** mientras el producto existe. Incluyen el **mantenimiento** del producto y llegan a su **fin cuando el producto es desechado**.

TIPOS DE CICLOS DE VIDA

Existen **tres tipos** básicos de ciclos de vida en el desarrollo de software:

Secuencial: Las etapas se ejecutan **una después de otra** sin retorno. Ejemplo: el ciclo de vida en cascada.

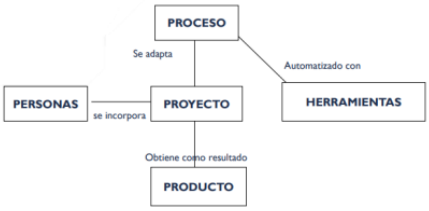
Iterativo/Incremental: Se implementa en **procesos empíricos** para adaptarse y mejorar continuamente. Se realizan **iteraciones para ganar información** y convertirla en incrementos que se incorporan al sistema.

Recursivo: Utilizado en **casos particulares**, como proyectos de alto riesgo. Ejemplos incluyen el ciclo de vida de espiral y el ciclo de vida en B, que se centran en **características específicas en cada iteración.**

La elección del ciclo de vida **depende** de las características **del proyecto y el producto**, considerando costos y el entorno. **Los ciclos de vida recursivos están en desuso** debido a su limitada utilidad demostrada.

El proceso de desarrollo de software es la implementación práctica del ciclo de vida, que es una guía que define las etapas y su orden. El proceso tiene un objetivo claro: la creación de un producto de software que debe ser específico y alcanzable.

COMPONENTES DE PROYECTO DE DESARROLLO DE SOFTWARE



Un **proyecto** se considera una instancia del **proceso** definido en un ciclo de vida. En un proyecto, las **personas** implementan **herramientas** para automatizar los procesos y producir un **producto** de software como resultado.
El **proceso** es una plantilla abstracta que se adapta a las necesidades específicas del **proyecto**. Toma requerimientos como entrada y, a través de actividades estructuradas y guiadas por un objetivo, produce un producto de software.
El **proyecto** es la unidad organizativa que gestiona recursos y personas para lograr este resultado.

Los proyectos de software tienen ciertas características:

1. **Duración Limitada**: Son **temporales** y tienen un **inicio y un fin**. Cuando se alcanzan los objetivos, el proyecto se da por concluido.
2. **Orientados a un Objetivo**: Cada proyecto tiene un **objetivo claro, no ambiguo y alcanzable**. Esto permite medir su finalización y diferencia cada proyecto de los demás.
3. **Únicos**: Aunque puedan ser similares, **cada proyecto es único** debido a sus características específicas.
4. **Tareas Relacionadas**: Las **tareas** en un proyecto están **interconectadas y dependen unas de otras**. Tienen precedencia y requieren recursos y esfuerzos asignados, lo que hace que la gestión de proyectos sea compleja. Las tareas se derivan del proceso.

Cuando trabajamos en un proyecto de un proceso definido lo llamamos:

GESTIÓN TRADICIONAL DE UN PROYECTO

La **gestión tradicional** de un proyecto se enfoca en **ejecutar las actividades** definidas, utilizando los **conocimientos, habilidades, herramientas y técnicas** necesarias para **alcanzar los objetivos** del proyecto y obtener el producto esperado. Se trata de la administración de proyectos (Project Managment) aplicada de manera convencional.

LA TRIPLE RESTRICCIÓN

La triple restricción en la gestión de proyectos se refiere a **tres factores:**

1. Alcance: Los requerimientos y objetivos del cliente, es decir, lo que se desea lograr con el proyecto.
2. Tiempo: El plazo o el tiempo necesario para completar el proyecto.
3. Costo: Los recursos y el dinero necesarios para llevar a cabo el proyecto.

Estos tres factores están **interconectados**, y si se modifica uno de ellos, afectará a los otros dos. Es **responsabilidad del PM** lograr el equilibrio entre estos factores, es **esencial** para garantizar la **calidad** del proyecto y su cumplimiento en términos de **objetivos, tiempo y presupuesto**. La **calidad** del producto **no es negociable** en la gestión de proyectos de alta calidad.

ROLES DENTRO DE UN PROYECTO

En la gestión tradicional nuestro equipo de proyecto está formado por un **EQUIPO** propiamente dicho y un **LÍDER DE PROYECTO (PM)**.

LIDER DE PROYECTO - PROJECT MANAGER (PM)

El Líder de Proyecto o **Project Manager (PM)** es responsable de **gestionar todas las actividades** necesarias para guiar al equipo hacia el logro de los objetivos del proyecto. Esto incluye la **administración de recursos**, la **organización del trabajo**, la **supervisión del progreso** y la **adaptación de los planes** según las necesidades. El PM también **se comunica con las partes** interesadas y **regula el avance** del proyecto. En la gestión tradicional de proyectos, el PM desempeña un papel central, mientras que, en los procesos empíricos, como los métodos ágiles, su papel puede ser menos relevante.

EQUIPO DE PROYECTO

El **Equipo de Proyecto** está formado por individuos comprometidos en alcanzar objetivos compartidos y mutuamente responsables. Todos **trabajan hacia un objetivo alcanzable** y medible, y **complementan sus conocimientos** y habilidades para

lograrlo. La **comunicación efectiva** y el trabajo en equipo son fundamentales, por lo que generalmente se prefieren equipos pequeños para fomentar la colaboración.

Para llevar el proyecto adelante necesitamos lo que llamamos:

PLAN DE PROYECTO

El **plan de proyecto es como una hoja de ruta** que proporciona una guía inicial para abordar un proyecto. Aunque no esté completamente definido, esboza el alcance, los objetivos, los recursos, el cronograma, las personas involucradas y sus roles. Este plan detallado **sirve como guía para gestionar y ejecutar el proyecto** desde el principio.
A través del plan de proyecto, **documentamos los siguientes aspectos**:

- ¿Qué es lo que hacemos?: Define el alcance del proyecto.
- ¿Cuándo lo hacemos?: Establece la calendarización.
- ¿Cómo lo hacemos?: Describe los recursos y las decisiones disponibles.
- ¿Quién lo va a hacer?: Asigna las tareas a las personas involucradas.

El plan de proyecto debe ser un **documento dinámico** que se **actualiza continuamente** a lo largo del proyecto, ya que en el entorno de desarrollo del proyecto pueden surgir variaciones no previstas debido a la incertidumbre inicial.

PLANIFICACIÓN DE PROYECTOS DE SOFTWARE

Cuando hablamos de planificar el proyecto de software incluimos:

DEFINCIÓN DEL ALCANCE

Es importarte hacer la diferenciación entre proyecto y producto de software.

- El **alcance del proyecto** se refiere a todo el **trabajo necesario para entregar el producto** o servicio, incluyendo todas las características y funciones especificadas en el objetivo. Se mide según el Plan de Proyecto.
- El **alcance del producto** abarca todas las **características que pueden incluirse en el producto** o servicio y se mide según la Especificación de Requerimientos (por ejemplo, casos de uso).

La relación entre ambos es que**, para definir el alcance del proyecto, primero se debe determinar el alcance del producto,** ya que un producto más grande requerirá más tareas, tiempo y recursos en el proyecto.

DEFINICIÓN DE PROCESO Y CICLO DE VIDA

Al iniciar un proyecto, es necesario definir qué **proceso y ciclo de vida** se utilizarán.

- El **proceso** de desarrollo es el **conjunto de actividades** requeridas para construir el producto de software, como el Proceso Unificado de Desarrollo (PUD).
- El **ciclo de vida** describe **cómo se ejecutan** esas actividades. En la **gestión tradicional**, **se puede elegir** tanto el proceso como el ciclo de vida. En la **gestión ágil,** las opciones de ciclo de vida suelen ser más limitadas, con un enfoque **iterativo** como el más comúnmente utilizado.

ESTIMACIÓN

Al crear un plan de proyecto, es esencial realizar **estimaciones** de diferentes características, aunque estas estimaciones siempre estarán **sujetas a cierto grado de incertidumbre**. Las **características** que **debemos definir en orden** son:

1. Tamaño: Determinar el **alcance** y las **dimensiones** del producto a construir.
2. Esfuerzo: Estimar las **horas de trabajo necesarias (horas persona)** de manera lineal.
3. Calendario: Establecer el **calendario de trabajo**, incluyendo **días y horas**, y **cuántas personas** van a trabajar
4. Costo: Determinar el **presupuesto necesario** para llevar a cabo el proyecto.
5. Recursos Críticos: Identificar los **recursos humanos y físicos** esenciales para el proyecto.

A medida que avanza el proyecto, la incertidumbre disminuye y se obtiene una mayor precisión en las estimaciones.

GESTIÓN DE RIESGOS

Los riesgos son **eventos que pueden o no ocurrir**, pero si suceden, **pueden afectar el éxito del proyecto**. Para gestionar los riesgos:

1. **Identificamos los riesgos más probables** o los que tendrían el mayor impacto en el proyecto.
2. **Calculamos la exposición de riesgo** multiplicando el **impacto potencial** por la **probabilidad de ocurrencia**.
3. **Seleccionamos los riesgos clave** que abordaremos en la gestión de riesgos.
4. **Generamos acciones para mitigar** o **evitar** la ocurrencia de los riesgos, así como para **reducir su impacto**.

5. **Reconocemos que algunos riesgos** pueden ser **difíciles de controlar** y debemos encontrar formas de **mitigarlos.**
6. **Actualizamos constantemente la gestión de riesgos** a medida que el proyecto avanza, ya que los riesgos pueden cambiar en probabilidad e impacto.

ASIGNACIÓN DE RECURSOS

La asignación de recursos es un paso crítico en la planificación de proyectos de software. Implica determinar qué recursos serán necesarios para llevar a cabo el proyecto de manera eficiente y efectiva. Algunos de los aspectos incluyen:

1. **Recursos Humanos:** Identificar y asignar roles y responsabilidades a los miembros del equipo. Esto implica seleccionar a las personas adecuadas con las habilidades necesarias para cada tarea y definir claramente sus funciones en el proyecto.
2. **Recursos Financieros:** Estimar y asignar el presupuesto necesario para el proyecto, teniendo en cuenta los costos de personal, herramientas, software, hardware y otros gastos relacionados con el desarrollo del software.
3. **Recursos Técnicos:** Determinar qué tecnologías, herramientas y equipos serán necesarios para llevar a cabo el proyecto. Esto puede incluir la adquisición o configuración de servidores, software de desarrollo y pruebas, licencias, entre otros.
4. **Planificación de la Capacitación:** Si es necesario, planificar la capacitación del equipo en nuevas tecnologías o metodologías que se utilizarán en el proyecto.

PROGRAMACIÓN DE PROYECTOS

La programación de proyectos en la planificación de proyectos de software implica crear un cronograma detallado que establece cuándo se llevarán a cabo las diferentes tareas del proyecto. Esto incluye:

1. Descomponer el proyecto en tareas más pequeñas y manejables.
2. Determinar el orden y las dependencias entre las tareas.
3. Estimar la duración de cada tarea.
4. Asignar recursos a las tareas según el cronograma.
5. Crear un calendario que muestre las fechas de inicio y finalización de cada tarea.
6. Realizar un seguimiento constante del progreso y ajustar el cronograma según sea necesario.

DEFINICIÓN DE MÉTRICAS

Las métricas de software son **herramientas esenciales** para **evaluar si un proyecto está siguiendo el plan** previsto. Se basan en **mediciones de la realidad** y se utilizan para **verificar el cumplimiento** de las estimaciones iniciales. Cada métrica debe tener **un propósito claro y contribuir al éxito del proyecto**.

Las métricas de software se pueden clasificar en **tres dominios principales:**

1. Métricas de Proceso: Estas métricas **evalúan cómo funciona** el proceso de desarrollo de software en **una organización.** Ayudan a determinar si la organización está trabajando de manera eficiente y efectiva en términos globales. Ejemplos incluyen el **porcentaje de proyectos que se completan** a tiempo y **el porcentaje de proyectos que se retrasan.**
2. Métricas de Proyecto: Estas métricas se utilizan para **evaluar si un proyecto de software específico se está desarrollando de acuerdo con el plan**. Por ejemplo, **comparar** el **tiempo calendario real con el tiempo planificado** es una métrica de proyecto común.
3. Métricas de Producto: Estas métricas se centran en las **características y la calidad del producto de software** que se está construyendo. Ejemplos de métricas de producto incluyen **métricas de defectos y métricas de tamaño del software.**

MÉTRICAS BÁSICAS

- TAMAÑO
- ESFUERZO
- TIEMPO(CALENDARIO)
- DEFECTOS

¿CADA CUÁNTO TOMO MÉTRICAS?

La frecuencia para tomar métricas en un proyecto de software debe ser regular, pero ajustada a las necesidades específicas del proyecto, evitando consumir recursos innecesarios.

MONITOREO Y CONTROL

El Project Manager (PM) **vigila el progreso** del proyecto utilizando **el plan de proyecto y métricas**, con el objetivo de **corregir desviaciones** a tiempo y asegurar que el proyecto se mantenga en curso para cumplir sus objetivos. Comparar lo planificado con lo real es una práctica común y es esencial debido a la tendencia a la optimismo en las estimaciones de desarrollo de software.

FACTORES PARA EL ÉXITO

1. Monitoreo y Feedback: tener un **monitoreo permanente** y generar **acciones correctivas** cuando sea necesario para evitar una desviación mayor.
2. Misión/Objetivo claro: saber **hacia dónde** vamos
3. Comunicación: en **todos sus aspectos**, con el líder de proyecto, con el equipo, con los clientes y con los stakeholders.

CAUSAS DE FRACASOS

1. Fallar al definir el problema
2. Planificar basado en datos insuficientes
3. La planificación la hizo el grupo de planificaciones
4. No hay seguimiento del plan del proyecto
5. Plan de proyecto pobre en detalles
6. Planificación de recursos inadecuada
7. Las estimaciones se basaron en “supuestos” sin consultar datos históricos
8. Nadie estaba a cargo
9. Mala comunicación

FILOSOFÍA ÁGIL

Hace aproximadamente 20 años, surgió el **movimiento ágil** en la industria del software, encabezado por un grupo de referentes que se reunieron en Utah y acordaron el **Manifiesto Ágil**. Este manifiesto representa un **compromiso** de las personas involucradas en proyectos de **trabajar** de una **manera específica**, buscando un **equilibrio** entre procesos rigurosos y **eficiencia** en el desarrollo de software. El objetivo era **evitar** extremos como la **falta de profesionalismo** y la **burocratización excesiva**, centrándose en la **entrega continua** de software de calidad y la **satisfacción del cliente**. Estas prácticas ágiles se aplican en procesos empíricos.

El agilismo, también conocido como **Agile**, es una **ideología** que busca un **equilibrio** entre la falta de procesos y el exceso de procesos en el desarrollo de software. Se basa en un **conjunto de principios** que guían el desarrollo del producto y se sustenta en **procesos empíricos** que se apoyan en la **experiencia** y la **retroalimentación** continua del equipo y el cliente. El enfoque ágil implica **ciclos de realimentación cortos**, comenzando con un producto mínimo viable (MVP) y realizando mejoras y correcciones a lo largo del proceso de desarrollo para adaptarse a la incertidumbre y los cambios inevitables. El empirismo se basa en **tres pilares** fundamentales:

1. Transparencia: Implica la comunicación abierta y honesta de información relevante a todas las partes interesadas, fomentando la confianza y la colaboración.
2. Adaptación: Se refiere a la capacidad de ajustar el trabajo y el proceso en respuesta a desviaciones y problemas detectados, lo que permite una respuesta efectiva a los cambios.
3. Inspección: Involucra la revisión regular y sistemática del progreso del trabajo y los productos resultantes para identificar problemas y desviaciones del plan, facilitando la toma de decisiones informadas.

MANIFIESTO ÁGIL

El manifiesto ágil, tiene 4 valores a los que llamamos:

VALORES ÁGILES

1. Individuos e interacciones por sobre procesos y herramientas: Destaca la importancia de las relaciones personales, la comunicación y la colaboración en el equipo y con las partes interesadas, priorizando estos aspectos sobre la adhesión rígida a procesos y herramientas.

2. Software funcionando por sobre documentación extensiva: Se enfoca en la entrega de software funcional y de alta calidad en lugar de centrarse en la documentación exhaustiva. Reconoce la necesidad de documentación, pero aboga por generarla cuando sea realmente necesaria y mantenerla de manera eficiente.

3. **Colaboración con el cliente por sobre negociación contractual:** Promueve la colaboración activa con el cliente para comprender sus necesidades y requisitos en lugar de basarse en acuerdos contractuales estrictos. Se trata de trabajar en conjunto y adaptarse a las cambiantes necesidades del cliente.

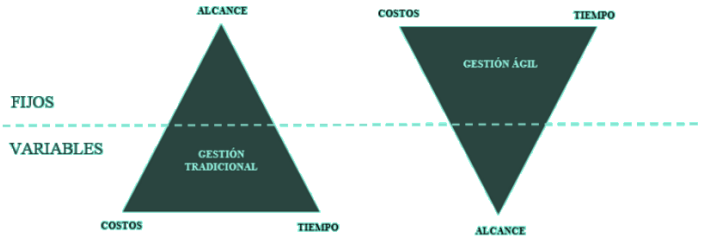
4. **Responder al cambio por sobre seguir un plan:** Destaca la importancia de la adaptación y la flexibilidad en respuesta a cambios y desviaciones en el proceso de desarrollo, en contraposición a la adhesión inflexible a un plan predefinido. Se valora la capacidad de ajustar el enfoque según las circunstancias y las necesidades del proyecto y del cliente.

Cuando hablamos de **Requerimientos Ágiles**, debemos ponerlo en contexto, se trata de una manera de trabajar con los requerimientos que está alineada con los...

12 PRINCIPIOS DEL MANIFIESTO ÁGIL

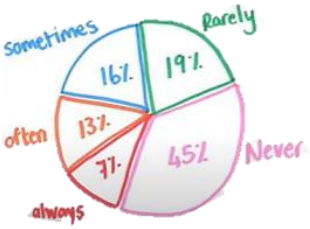
1. **Satisfacción del cliente:** La prioridad principal es **satisfacer al cliente** entregando software de **valor** de manera temprana y continua.
2. **Adaptación al cambio:** Se acepta que los requerimientos pueden cambiar, incluso en etapas avanzadas del desarrollo, y se aprovecha el cambio para obtener ventajas competitivas.
3. **Entregas frecuentes:** Se entregan incrementos funcionales de software con regularidad, priorizando la funcionalidad sobre la documentación extensa.
4. **Trabajo en equipo:** La colaboración con el cliente y las partes interesadas es fundamental durante todo el proyecto para garantizar que el software entregado satisfaga las necesidades del negocio.
5. **Personas motivadas:** Se construyen proyectos en torno a individuos motivados, proporcionándoles el apoyo y el entorno necesarios y confiando en su capacidad para realizar el trabajo.
6. **Comunicación directa:** Se considera que la comunicación cara a cara es la forma más efectiva de transmitir información dentro del equipo de desarrollo, fomentando la efectividad y la riqueza de la comunicación.
7. **Software funcionando:** El software funcionando es la medida principal de progreso en el desarrollo de software.
8. **Continuidad:** Se promueve el desarrollo sostenible, asegurando que el equipo de trabajo pueda mantener un ritmo constante a lo largo del tiempo.
9. **Excelencia técnica:** La calidad del producto no es negociable, y se busca mejorar la agilidad a través de la excelencia técnica y el buen diseño.
10. **Simplicidad:** Se busca maximizar la eficiencia y efectividad reduciendo la complejidad y eliminando tareas innecesarias.
11. **Equipos auto-organizados:** Se fomenta la autonomía y la capacidad de decisión de los equipos auto-organizados, que pueden tomar decisiones colaborativas y diseñar soluciones de manera independiente.
12. **Mejora continua:** El equipo reflexiona y ajusta su comportamiento de manera continua para mejorar su efectividad en intervalos de tiempo definidos.

TRIÁNGULO DE HIERRO vs TRIÁNGULO ÁGIL



El triángulo de hierro en la gestión tradicional de proyectos se compone de alcance, tiempo y costo, donde cualquier cambio en uno de estos elementos puede afectar los otros dos. En contraste, las metodologías ágiles consideran el **valor**, la **calidad** y la **restricción del tiempo** como los elementos fundamentales, lo que da lugar al **Triángulo Ágil**. En las metodologías ágiles, se busca **maximizar el valor entregado** al cliente, **garantizar la calidad** del producto y **cumplir con los plazos**, en lugar de centrarse en el alcance y el costo. Además, se utilizan iteraciones cortas para **adaptar el proyecto** según las necesidades cambiantes en lugar de establecer un plan fijo al inicio del proyecto. La gestión ágil se enfoca en el **valor que aportan las funcionalidades** en relación con los **recursos** y el **tiempo** disponibles.

REQUERIMIENTOS ÁGILES

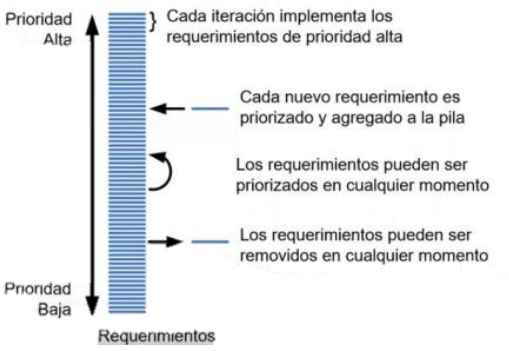


El enfoque ágil propone una gestión de proyectos y definición de requerimientos diferente. Se centra en el **valor de negocio** y en construir el **producto correcto**. **Los requerimientos se descubren a medida que avanza el proyecto**, definiendo **solo lo necesario para comenzar** y obteniendo retroalimentación temprana. La **colaboración con el cliente** es clave, y los requerimientos se expresan como **historias de usuario**, descripciones breves de las

necesidades del usuario escritas en lenguaje natural y **priorizadas según el valor que aportan** al producto. Esto contrasta con el enfoque tradicional de especificar todos los requerimientos desde el principio (BRUF - Big Requirements Up Front).

La gestión ágil de requerimientos se enfoca en **reducir el desperdicio de funcionalidades** poco utilizadas en el software. Esto se logra **priorizando las funcionalidades** que aportan valor al cliente. El **Product Owner** es fundamental en este proceso, ya que es responsable de identificar las necesidades del negocio y priorizar los requerimientos en una lista llamada **Product Backlog**, donde los más importantes están en la parte superior. Esta organización en forma de pila obliga a priorizar los requerimientos por orden de importancia.

La **gestión ágil de requerimientos** se enfoca en **trabajar en lo que agrega más valor** al cliente y **hacerlo en el momento adecuado**. Comenzamos definiendo los requerimientos más **prioritarios,** los que aportan mayor valor. Trabajamos con mayor detalle en los requerimientos en la parte superior de la lista y, a medida que los completamos, avanzamos hacia los siguientes en orden de prioridad. A medida que avanzamos, los requerimientos pueden ser modificados, reorganizados o nuevos pueden agregarse a la lista según las necesidades cambiantes del proyecto.



JUST IN TIME

En la gestión ágil de requerimientos, se aplica el principio ***"Just in Time",*** lo que significa que se **analiza** y **detalla** un requerimiento **solo cuando es necesario** y **aporta suficiente valor**. Esto ayuda a evitar desperdiciar tiempo en especificar requerimientos que luego pueden cambiar. El enfoque se centra en entregar valor de negocio al cliente en lugar de simplemente características de software.

METODOLOGIA TRADICIONAL vs METODLOGÍA ÁGIL

	TRADICIONAL	ÁGIL
Prioridad	Cumplir el plan	Entregar Valor
Enfoque	Ejecución ¿Cómo?	Estrategia (¿Por qué? ¿Para qué?)
Definición	Detallados y cerrados. Descubrimientos al inicio	Esbozados y evolutivos. Descubrimiento progresivo
Participación	Sponsors, stakeholder de mayor poder e interés.	Colaborativo con stakeholders de mayor interés (clientes, usuarios finales)
Equipo	Analista de negocios, Project Manager y Áreas de Proceso.	Equipo multidisciplinario.
Herramientas	Entrevistas, observación y formularios	Principalmente prototipado. Técnicas de facilitación para descubrir
Documentación	Alto nivel de detalle. Matriz de Rastreabilidad para los requerimientos.	Historias de Usuario. Mapeo de historias (Story Mapping)
Productos	Definidos en alcance	Identificados progresivamente
Procesos	Estables, adversos al cambio.	Incertidumbre, abierto al cambio

TIPOS DE REQUERIMEINTOS

1. **Requerimientos de Negocio:** Estos son los requerimientos que se relacionan con la visión y los objetivos del negocio. Representan las necesidades de alto nivel que el software debe abordar.
2. **Requerimientos de Usuario:** Estos requerimientos se centran en las necesidades y expectativas de los usuarios finales del software.
3. **Requerimientos Funcionales:** Estos requerimientos se asemejan a los casos de uso en la gestión tradicional de proyectos. Describen las funciones específicas que el software debe realizar.
4. **Requerimientos No Funcionales:** Estos son requerimientos que se centran en aspectos que no están directamente relacionados con las funciones específicas del software, pero que son igualmente importantes. Pueden incluir aspectos como el rendimiento, la seguridad, la usabilidad, la escalabilidad y otros atributos del sistema. Son **esenciales** para la calidad y el éxito del software.
5. **Requerimientos de Implementación:** Se consideran parte de los requerimientos no funcionales, se relacionan con cuestiones específicas de la implementación del software, como las tecnologías a utilizar, las restricciones técnicas y las consideraciones de hardware y software que deben tenerse en cuenta.

La clasificación de requerimientos en la gestión ágil es importante para distinguir entre aquellos relacionados con el dominio del problema y los específicos del software. Esto permite comprender las necesidades del usuario y las restricciones del

contexto en el que se utilizará el software, lo que facilita la adaptación a los cambios y la priorización de características esenciales.
La metodología ágil se centra en comprender lo que agrega valor al negocio, identificando los requerimientos de negocio y luego construyendo una solución que pueda entregarse al cliente. Prioriza lo que es más valioso para el cliente y se basa en entregas continuas.
La gestión ágil de requerimientos implica colaborar con personas técnicas y no técnicas para comprender las necesidades y el negocio, y construir un software que entregue valor. Luego, en conjunto con los usuarios, se descubre la mejor manera de satisfacer esas necesidades a través del Product Backlog, lo que facilita la entrega de valor de manera iterativa.

PRINCIPIOS ÁGILES RELACIONADOS A LOS REQUERIMIENTOS ÁGILES

Solo enunciamos los MÁS relacionados, pero de alguna manera podríamos encontrar la forma de relacionar TODOS los principios con lo que tiene que ver con Requerimientos Ágiles.

1. La prioridad es satisfacer al cliente a través de releases tempranos y frecuentes (2 semanas a un mes)
2. Recibir cambios de requerimientos, aun en etapas finales.
4. Técnicos y no técnicos trabajando juntos todo el proyecto: El PO es parte del equipo, porque si los no técnicos no son parte del equipo del proyecto, difícilmente pueda identificar qué da valor al negocio
6. El medio de comunicación por excelencia es cara a cara.
11. Las mejores arquitecturas, diseños y requerimientos emergen de equipos auto organizados

USER STORIES

Las "User Stories" son una **técnica para capturar requerimientos** de manera narrativa. Aunque no necesitan ser extremadamente detalladas al principio del proyecto, **expresan la intención de hacer algo**. No son especificaciones exhaustivas de requerimientos como los casos de uso, sino que se centran en la intención y **pueden evolucionar** con el tiempo. Además, **fomentan la comunicación** y pueden servir como entrada a la documentación incremental. Su enfoque es **capturar la esencia de un requerimiento** de manera efectiva y **se valoran las conversaciones y discusiones** sobre ellas.

Las 3 “C”

1. Conversación: Representa la parte no escrita y **más importante de la historia**. Es la **comunicación** y el **diálogo** entre el **equipo** de desarrollo y el **Product Owner** para comprender las necesidades y los detalles del requerimiento.
2. Confirmación: Establece las **condiciones** que deben cumplirse para considerar la User Story como **completada.** Estas condiciones se encuentran dentro de la historia y sirven como **criterios de aceptación**.
3. Card: Es la parte **visible** de la User Story y contiene la **descripción de la historia**, de qué se trata y su valor para el negocio. Esta descripción suele ser breve y se utiliza como referencia para el equipo.

FORMA DE EXPRESAR LAS HISTORIAS DE USUARIO

COMO <NOMBRE DE ROL> YO PUEDO <ACTIVIDAD> DE FORMA TAL QUE <VALOR DE NEGOCIO QUE RECIBO>
El nombre de rol representa quién está realizando la acción o quién recibe el valor/beneficio de la actividad/acción; la actividad representa la acción que realizará el sistema y el valor de negocio que recibo comunica porque es necesaria la actividad.

Debemos intentar expresar la US con una **frase verbal** que nos permita **identificar de que se trata**/que va a tener esa tarjeta.

El rol no representa a una persona, sino lo que puede estar haciendo una persona. Ejemplo: en una estación de servicio, el playero puede estar en el despacho de combustible y después puede ir y facturar, en esas instancias tiene roles distintos.

FUNCIONES DE LAS US EN EL DESARROLLO DE SOFTWARE

Las US son **multipropósito** y pueden cumplir varias funciones en el proceso de desarrollo de software:

1. Representan una **necesidad del usuario**, especificando lo que el usuario necesita.
2. **Describen una característica específica del producto**, detallando cómo debe funcionar o qué debe hacer el producto.
3. Sirven como **ítem de planificación**, ya que, al priorizar las historias, se determina qué se implementará en el proyecto.
4. **Actúan como tokens para iniciar una conversación**, recordando al equipo que es importante discutir y especificar los detalles de la historia antes de su implementación.
5. Funcionan como **mecanismos para diferir conversaciones**, permitiendo que algunas discusiones y detalles se aborden en el futuro, cuando la historia esté más arriba en la lista de prioridades.

En cuanto a la implementación de las User Stories como **porciones verticales**, es responsabilidad del equipo de desarrollo asegurarse de que cada historia **se implemente de manera completa**, desde la **lógica de interfaz** de usuario hasta la **lógica de la base de datos** (persistencia). Esto significa que el equipo debe considerar todos los aspectos técnicos y arquitectónicos necesarios para entregar una funcionalidad completamente funcional.

MODELADO DE ROLES

Al **modelar los roles** dentro del sistema para identificar las User Stories, se utiliza una **"tarjeta de rol de usuario"** que consiste en definir un **nombre específico para el rol** y proporcionar una **explicación detallada de su función**. Es importante **evitar la ambigüedad** al nombrar los roles de usuario y evitar usar términos genéricos como "usuario". También se deben evitar roles representativos como gerentes de usuarios, gerentes de desarrollo, miembros de marketing, vendedores, expertos en el dominio, clientes, capacitadores y personal de soporte al definir los roles.

CRITERIOS DE ACEPTACIÓN

Los **criterios de aceptación** son **pautas** que **definen los límites** y la **intención** de una User Story, ayudando al Product Owner (PO) a expresar los **requisitos funcionales mínimos** necesarios para que la historia aporte valor al negocio. Estos criterios se centran en la **perspectiva del usuario** y no contienen detalles de implementación. Su redacción debe ser clara y definir validaciones concretas, utilizando palabras como **"PUEDE"** para aspectos opcionales y **"DEBE"** para aspectos obligatorios. Los criterios de aceptación ayudan a los desarrolladores a saber cuándo detenerse en la implementación y a los testers a derivar las pruebas relevantes. **Pueden ser numerosos, pero no deben incluir detalles excesivos** y deben mantener un alto nivel de abstracción.

PRUEBAS DE ACEPTACIÓN

Las **pruebas de aceptación** son un **complemento** de la User Story y se encuentran en el dorso de la tarjeta. Estas pruebas **proporcionan detalles adicionales** sobre la conversación y la confirmación de que la historia cumplirá con las expectativas del Product Owner (PO). **Ayudan a profundizar** en la definición de la User Story **y a finalizar los detalles** de cómo implementarla. Las pruebas de aceptación **describen lo que se espera que suceda** cuando se ejecute la historia de usuario y pueden indicar si el resultado de la prueba es **"PASA"** o **"FALLA".**

DEFINICIÓN DE LISTO o CRITERIO DE READY

El "Listo" es un conjunto de **condiciones acordadas por el equipo** que determina si una User Story está **lista para ser implementada** en un sprint. Esta definición asegura que la User Story cumple con todos los aspectos necesarios antes de ser trabajada en el desarrollo.

El **Modelo INVEST** es un conjunto de características que se utilizan para determinar si una User Story está lista:

- I: **Independiente:** La US no debe depender de otras para su implementación y puede ser trabajada en cualquier orden.
- N: **Negociable:** Se negocia el "qué" debe hacerse, no el "cómo" debe hacerse, evitando detalles de implementación en la US.
- V: **Valuable (Valioso):** La US debe proporcionar un valor concreto para el cliente.
- E: **Estimable:** Debe ser posible estimar el esfuerzo necesario para implementar la US y ayudar al cliente a priorizar basándose en costos.
- S: **Small (Pequeña):** La US debe ser lo suficientemente pequeña como para ser completada en un sprint.
- T: **Testeable (Comprobable):** Debe ser posible demostrar que la US ha sido implementada correctamente a través de pruebas.

DEFINICIÓN DE HECHO

Esta **definición de Hecho** también es propia del equipo. También **se valida con un checklist** donde especificamos cuales son todas las características que debe tener la US y lo que indica es si la historia está decentemente **terminada para presentársela al PO/Cliente.** Debe tener las **pruebas unitarias y de aceptación en verde**, tiene que haber pasado el ambiente de prueba, etc.

NIVELES DE ABSTRACCIÓN

Puede suceder que, en diferentes momentos, los requerimientos tengan diferentes niveles de abstracción, y en ese sentido, son las User Stories que cumplen con el criterio de "ready" las que estarán incluidas en el Backlog de un proyecto en un momento en específico. Sin embargo, también podemos tener otros tipos de US, como las Épicas y los Temas, que todavía no pueden ser incluidas en el Product Backlog, pero que podrían hacerlo en el futuro.

1. **Épicas:** Son historias de usuario muy grandes y abstractas que se colocan en el Product Backlog antes de ser detalladas. Se espera que se dividan en historias más pequeñas en el futuro cuando se aborden.
2. **Temas:** Son conjuntos de User Stories relacionadas que se agrupan bajo un título. Tienen un nivel de abstracción mayor que las épicas y pueden representar ideas o propuestas de valor aún no detalladas.
3. **Spikes:** Son User Stories especiales utilizadas para investigar y eliminar riesgos o incertidumbres relacionadas con tecnologías desconocidas o interacciones del usuario antes de comprometerse con el desarrollo. Pueden ser técnicas o funcionales.
   - **Spikes Técnicas:** Se enfocan en investigar enfoques técnicos o tecnologías desconocidas.
   - **Spikes Funcionales:** Abordan incertidumbres relacionadas con la interacción del usuario con el sistema, a menudo mediante prototipos para obtener retroalimentación.

   Los "spikes" son utilizados para abordar **incertidumbres** o **riesgos antes de la implementación** de una User Story. Normalmente **se trabajan en un sprint anterior al sprint en el que se planea implementar la historia**. Pueden ser utilizados para familiarizarse con nuevas tecnologías o dominios, analizar comportamientos complejos de una historia, ganar confianza frente a riesgos tecnológicos o resolver incertidumbres funcionales relacionadas con la interacción del usuario. Los spikes deben ser **estimables, demostrables y aceptables**, cumpliendo con el **Criterio de Ready**. Se consideran una excepción y se gestionan dentro de la misma User Story siempre que sea posible.

ESTIMACIONES DE SOFTWARE

Estimar en un proyecto de software es **predecir y hacer suposiciones** sobre recursos, costos y otras características del proyecto **en un momento donde hay incertidumbre**. Las estimaciones **no son precisas**, especialmente al comienzo del proyecto, y a medida que avanzamos y obtenemos más información, debemos ajustarlas para mayor precisión. Las estimaciones **no son compromisos**, y el plan del proyecto puede variar respecto a las estimaciones iniciales. Sin embargo, cuanto mayor sea la diferencia entre las estimaciones y el plan, mayor será el riesgo en el proyecto. Planificar incluye consideraciones adicionales más allá de las estimaciones, como los objetivos del negocio.

¿PARA QUÉ ESTIMAMOS?

Estimamos en proyectos de software **para predecir en un entorno de incertidumbre**. Las estimaciones se realizan cuando tenemos menos información de la necesaria y **sirven para planificar y gestionar el proyecto**. A medida que avanzamos, obtenemos más información que puede confirmar o refutar nuestras estimaciones, pero, aun así, las estimaciones son **esenciales para lidiar con la incertidumbre** en el desarrollo del producto.

Existen **diversas técnicas** de estimación que ayudan a reducir la incertidumbre en las estimaciones, pero **ninguna elimina por completo la posibilidad de error**. Es importante entender las razones detrás de posibles errores en las estimaciones, que pueden incluir la falta de organización en el proyecto, la falta de conocimiento sobre el producto, la falta de claridad en los recursos disponibles y las capacidades del equipo, y posibles errores en las técnicas de estimación. Para mitigar estos errores, **es recomendable utilizar múltiples técnicas** de estimación y comparar sus resultados.

La estimación en proyectos de software comienza con la evaluación del **tamaño del software,** que implica medir **cuán grande será el trabajo** por realizar. Esto se puede hacer contando **funcionalidades, requerimientos, User Stories, Casos de Uso u otras métricas** de conteo. La estimación del tamaño se basa en la información disponible y el nivel de incertidumbre asociado.

Además de estimar el tamaño, se realiza una **estimación del esfuerzo**, que se refiere a la cantidad de **horas lineales** necesarias para desarrollar el software. Esta estimación no especifica quién realizará el trabajo ni el calendario, se enfoca solo en cuantificar las horas requeridas.

Una vez estimado el esfuerzo, se procede a la **estimación del calendario,** que implica **planificar y asignar fechas de entrega.** A partir de estas estimaciones, se pueden abordar los **costos y presupuestos** del proyecto. Estos cálculos son **fundamentales para la gestión y planificación** efectiva de proyectos de software.

TÉCNICAS FUNDAMENTALES DE ESTIMACIÓN

Existen diferentes métodos de estimación en proyectos de software, cada uno basado en enfoques y técnicas específicas:

1. Métodos basados en la experiencia: Estos métodos se basan en la experiencia previa del equipo en proyectos similares para estimar el esfuerzo y el tiempo requerido para un nuevo proyecto. Utilizan el conocimiento acumulado a lo largo del tiempo.

2. **Métodos basados exclusivamente en recursos**: Estos métodos se centran en la cantidad y el tipo de recursos necesarios (personas, tiempo, herramientas) para llevar a cabo un proyecto. La estimación se realiza en función de la disponibilidad y el costo de estos recursos.
3. **Métodos basados exclusivamente en el mercado:** Estos métodos se basan en el análisis de costos de proyectos similares en el mercado para estimar el esfuerzo y el tiempo requerido para un nuevo proyecto. Son comunes en proyectos para clientes externos.
4. **Métodos basados en los componentes del producto o en el proceso de desarrollo:** Estos métodos descomponen el proyecto en sus componentes individuales, como módulos de software o actividades de desarrollo. Se estiman individualmente y luego se suman para obtener una estimación total del proyecto.
5. **Métodos algorítmicos:** Estos métodos utilizan modelos matemáticos y estadísticos para estimar el esfuerzo y el tiempo requeridos. Consideran diversas variables como el tamaño del software, la complejidad y la productividad del equipo.

Dentro de los métodos basados en la experiencia tenemos las técnicas de estimación de:

1. **DATOS HISTORICOS**: Implica comparar un proyecto nuevo con uno anterior que me nutra y me permita hacer la estimación de mi proyecto. Los datos básicos históricos concretos que voy a necesitar para tomar como **entrada** a mis estimaciones son el **tamaño, el esfuerzo, el tiempo y los defectos**. Necesito contar con información completa de estos datos de manera que se tomen siempre de la misma manera para los proyectos que analizo. *Problema:* Los dominios analizados pueden ser muy distintos.
2. **JUICIO EXPERTO (más utilizado):** Esta técnica se basa en la **experiencia y conocimiento de expertos** en la materia para determinar la estimación de esfuerzo y tiempo requerido para un proyecto de software. Una vez identificados los expertos, que es importante identificarlos bien, se les presenta la descripción detallada del proyecto y se les solicita que proporcionen estimaciones basadas en su experiencia y conocimiento. Es importante tener en cuenta que la técnica de juicio de experto **se basa en la subjetividad y experiencia** de los expertos, y **las estimaciones pueden variar significativamente** dependiendo de la experiencia y conocimiento de los expertos involucrados.
   Algunos criterios que pueden seguir los expertos para lograr una buena estimación y lograr una estructuración del juicio de experto son:
   - ✓ Estimar solo tareas con granularidad aceptable (no muy alta)
   - ✓ Usar el método **Optimista, Pesimista y Habitual** que tiene que ver con estimar según 3 características y luego aplicar esto a una formula. (O+4H+P)/6.
     (4 veces el tiempo habitual + el tiempo optimista + el pesimista y todo dividido 6)
     No es para seguirlo a rajatabla. Yo defino cuanto me va a llevar una determinada actividad, después planteó la fórmula y es el tiempo que me llevará.
   - ✓ Usar un checklist y un criterio definido para asegurar cobertura.

   PURO: Un experto estudia las especificaciones y hace su estimación. Se basa fundamentalmente en los conocimientos del experto.
   Problema: Si desaparece el experto, la empresa deja de estimar
3. **WIDEBAND DELPHI:** Un grupo de personas son informadas y tratan de adivinar lo que costara el desarrollo tanto en esfuerzo como en duración. Las estimaciones en grupo suelen ser mejores que las individuales. Las estimaciones individuales se recopilan de forma anónima y se envían a los expertos para que puedan revisarlas y ajustarlas en rondas sucesivas.
   Problema: Requiere más tiempo y recursos que el juicio de experto puro, ya que implica múltiples rondas de estimaciones y análisis de datos hasta que la estimación converja de forma razonable.
4. **ANALOGÍA:** También se basa en datos históricos, pero se seleccionan específicamente aquellos proyectos que son similares al proyecto actual para realizar la comparación y estimación.

Las estimaciones en proyectos de software a menudo omiten actividades importantes y tienden a ser **optimistas** debido a la falta de experiencia y a la incertidumbre inherente. Las omisiones incluyen requerimientos faltantes, actividades de desarrollo no consideradas inicialmente y actividades generales. **Las estimaciones deben ajustarse a medida que se obtiene más información y experiencia en el proyecto.**

ESTIMACIÓN EN AMBIENTES AGILES

En estimaciones ágiles, se aplican los mismos conceptos que en las estimaciones tradicionales, pero con un enfoque **pragmático y flexible**. Las estimaciones no son planes ni compromisos, sino pronósticos que se ajustan a medida que se obtiene más información. En **entornos ágiles**, se trabajan con **features** o **historias de usuario** para estimar el trabajo a realizar.

Las historias de usuario se estiman en **Story Points (SP),** que representan una **medida relativa** basada en la combinación de incertidumbre, esfuerzo y complejidad de la historia. Esta estimación **separa el esfuerzo del proyecto de su duración y es específica del equipo**, no siendo comparable entre equipos ni basada en el tiempo. Se utiliza una **historia de usuario canónica** como **referencia** para definir los Story Points, promoviendo un enfoque centrado en el equipo y mejorando la eficiencia en el análisis de historias.

Para estimar una historia de usuario, debemos considerar su **tamaño, tiempo y esfuerzo**. **El tamaño** refleja la cantidad de trabajo y su complejidad. Las estimaciones basadas en el **tiempo** son propensas a errores debido a factores externos. El **esfuerzo** se relaciona con las horas lineales necesarias. Es importante distinguir entre tamaño y esfuerzo. Para medir el progreso del equipo durante el proyecto e iteraciones, se utiliza:

VELOCITY

La velocidad (Velocity) es una métrica que mide el **progreso del equipo**. Se calcula **sumando los Story Points** de las **User Stories completadas al 100%** en una iteración. No se incluyen los SP de las historias parcialmente completas. Esta métrica permite corregir estimaciones y determinar la duración del proyecto al dividir el total de Story Points por la velocidad del equipo. La velocidad también ayuda en la planificación del proyecto.

POSIBLES MÉTODOS DE ESTIMACIÓN

POKER ESTIMATION

En la estimación ágil se utiliza una combinación de juicio de experto, analogía y el método Wideband Delphi. Se basa en la Serie de Fibonacci para asignar valores de complejidad a las User Stories, lo que permite una estimación más precisa. La secuencia de Fibonacci comienza en 1, y cada número es la suma de los dos anteriores (1, 1, 2, 3, 5, 8, 13, etc.). Se establece un límite de 8 para los Story Points, y una User Story con 0 indica desconocimiento y necesita más detalle.
El proceso de estimación se llama "Poker Planning" y los participantes son desarrolladores competentes en la tarea. Cada miembro del equipo elige una carta con un valor numérico de 0 a 100 y la coloca boca abajo. Después, las cartas se voltean simultáneamente, y los miembros que eligieron los valores más altos y bajos explican sus razones. Se repite el proceso hasta llegar a un consenso en los Story Points asignados a cada User Story.
En Agile, se enfatiza el empirismo y la retroalimentación, por lo que las estimaciones se ajustan a medida que se ejecuta, inspecciona y adapta el proyecto.

GESTIÓN DE PRODUCTOS

La **gestión de productos** implica tener una **visión clara** de por qué creamos productos de software, y esto va más allá de la **satisfacción del cliente**, la **generación de ingresos** o **la popularidad**. También puede estar relacionado con la ambición de **cambiar el mundo** o **mejorar nuestra diaria**. Sin embargo, a menudo invertimos tiempo en construir características que no se utilizan o se utilizan raramente, lo que implica un esfuerzo desperdiciado en algo que no aporta valor. Por lo tanto, es crucial eliminar el desperdicio y centrarse en lo que realmente es útil.

EVOLUCIÓN DE LOS PRODUCTOS DE SOFTWARE

Podemos organizar la **evolución de los productos de software** en una especie de **pirámide** con respecto a sus características. En la **base** de la pirámide se encuentran las características fundamentales que deben existir para que el software **funcione correctamente**, sea **confiable** y tenga cierta **usabilidad básica.** Estos **aspectos son esenciales** para lograr el propósito esperado y son lo que todos esperamos de la funcionalidad del software.

En **la cima** de la pirámide se ubican otras características relacionadas con la **visión del producto** que son más difíciles de alcanzar. Estas incluyen la **conveniencia**, la **satisfacción** del usuario y la **significatividad**

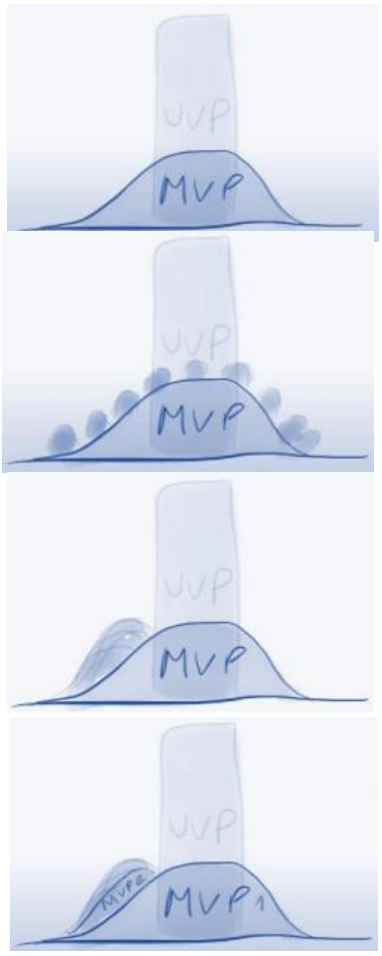
del producto. Sin embargo, muchas veces los productos se quedan en la base de la pirámide y no logran abarcar estos aspectos más elevados.

El desafío en la gestión de productos de software radica en **cómo eliminar el desperdicio y lograr abordar todos los aspectos que se persiguen al crear un producto**, ya sea satisfacer a un cliente, generar ventas, o cumplir con la visión del producto.

El enfoque aquí se centra en desarrollar un **producto mínimo** o una característica mínima para **probar si cumple** con las **expectativas del usuario** o las propias al desarrollar el producto. Se parte de una **hipótesis** sobre el valor que el producto puede ofrecer y se busca validarla con un esfuerzo mínimo, **evitando gastar recursos** en algo que no aportará valor al producto.

La técnica utilizada es la **UVP (User Value Proposition o Propuesta de Valor para el Usuario),** que se basa en **comprender las necesidades y problemas de los usuarios** para crear soluciones efectivas que las satisfagan. En lugar de desarrollar un producto completo con muchas funcionalidades desde el principio, se busca **minimizar el esfuerzo** para construir un producto que los usuarios realmente demanden, si la hipótesis inicial resulta ser válida. Esto **ayuda a evitar el desperdicio** de recursos en un producto que no cumple con las expectativas del mercado.

MVP (Mínimum Viable Product – Producto Mínimo Viable)



El **MVP** (Minimum Viable Product) es un concepto de **Lean Startup** que se centra en **desarrollar un producto con la cantidad mínima de funcionalidades** necesarias para que los usuarios potenciales puedan **evaluarlo y proporcionar retroalimentación**. El objetivo principal del MVP es **validar la hipótesis** sobre el producto y aprender de la participación de los usuarios para determinar si se deben hacer ajustes en el producto. No se trata de vender el MVP, sino de comprobar si el producto cumple con las expectativas de los usuarios.

Pueden ocurrir **dos situaciones**:

1. Los usuarios **encuentran el MVP satisfactorio** y expresan que lo necesitan tal como está. En este caso, la hipótesis original se valida, y el producto puede continuar desarrollándose en la misma dirección.
2. **Cada usuario** que prueba el MVP **proporciona retroalimentación** y **solicita funcionalidades** diferentes. Esto indica que el **mercado** y las **necesidades** de los usuarios **no están claros**, y la visión del producto **necesita ajustes** para encontrar un enfoque más adecuado.

En el desarrollo de un producto, la **hipótesis inicial puede evolucionar o modificarse** en función de las demandas y necesidades del mercado. En este proceso, el **MVP se ajusta y se mueve** hacia el enfoque que atrae más a los **clientes**. Aunque la hipótesis inicial puede ser acertada en su enfoque general, algunas de sus características pueden requerir ajustes o cambios para satisfacer mejor las expectativas de los usuarios y adaptarse al mercado.

Cuando la hipótesis inicial necesita modificaciones o ajustes, es necesario redefinir el MVP en función del conocimiento adquirido. Esto implica la creación de un nuevo MVP, que podríamos llamar MVP2. El proceso de retroalimentación se repite con este nuevo MVP, y este ciclo puede ocurrir varias veces hasta encontrar un feedback exitoso y llegar a la versión final del producto que queremos construir.

Una vez que encontramos el MVP exitoso, es decir que el feedback y la investigación son exitosas, surge el concepto de:

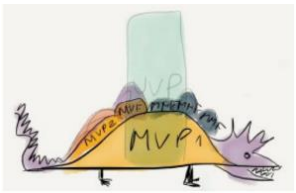
MMF (Mínimum Marketable Feature – Característica Mínima Comercializable)

El concepto de **MMF** (Mínimo Producto Comercializable) se enfoca en determinar los **elementos esenciales** que deben estar presentes en un producto o servicio **para que sea viable comercialmente**. A diferencia del MVP, el MMF se centra en la comercialización del producto en lugar de validar una hipótesis. El objetivo es identificar la **pieza mínima** que se debe **construir y comercializar** para que el producto sea rentable. Esto permite **evitar la construcción innecesaria** de características antes de verificar si el producto es comercializable. El MMF se refiere a una **"Feature" o funcionalidad extremadamente pequeña que representa lo mínimo que se puede lanzar al mercado** para evaluar su viabilidad.

Para combinar el MVP con la MMF, usamos la llamada:

MVF(Minimum Viable Feature - Característica Mínima Viable)

El concepto de **MVF** (Minimum Viable Feature - Característica Mínima Viable) se refiere a la **característica más pequeña y esencial** que se puede construir y lanzar rápidamente con recursos mínimos para **probar su utilidad y valor** en el mercado. El objetivo es **validar** cuál es la **característica** que realmente **hace la diferencia y atraerá** a los usuarios para comprar el producto. Si la MVF resulta exitosa, se pueden desarrollar más MMF en la misma área para aprovechar su potencial. La MVF es **una versión reducida del MVP** y se combina con MMF y MVP según las necesidades y la incertidumbre del negocio, creando así un producto más completo y adaptado al mercado.



MRF (Minimum Release Feature - Características Mínimas del Release)

La **MRF (Minimum Release Feature - Características Mínimas del Release)** se refiere al **conjunto más pequeño de características** que se pueden **incluir en un lanzamiento** del producto. Este conjunto representa el incremento más pequeño que proporciona un valor nuevo a los usuarios y satisface sus necesidades actuales. El **MMP** es equivalente al **MRF inicial**, pero es posible que se realicen **más lanzamientos** posteriores para mejorar el producto. El **MRF se logra en varias iteraciones**, **acumulando características** a lo largo del tiempo hasta que haya suficientes para un lanzamiento.

RELACIONES CONCRETAS

MVP
- Versión de un **nuevo producto** creado con el **menor esfuerzo posible**
- Dirigido a un **subconjunto de clientes potenciales**
- Utilizado para obtener **aprendizaje validado**.
- Más cercano a los **prototipos que a una version real funcionando de un producto**.

MMF
- es la **pieza más pequeña de funcionalidad** que puede ser liberada
- tiene valor tanto para la organización como para los usuarios.
- Es parte de un MMR or MMP.

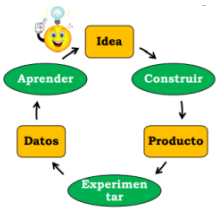
MMP
- Primer release de un MMR dirigido a **primeros usuarios** (early adopters),
- Focalizado en características clave que satisfarán a este grupo clave.

MMR
- Release de un producto que tiene el conjunto de características más pequeño posible.
- El incremento más pequeño que ofrece un valor nuevo a los usuarios y satisface sus necesidades actuales.
- **MMP = MMR1**

Los **errores más comunes** relacionados con MVP, MMF y MMP son:

1. Confundir un MVP, que se enfoca en aprender y validar hipótesis, con una Característica Comercial Mínima (MMF) o un Producto Comercializable Mínimo (MMP), que buscan obtener un retorno de inversión.
2. Correr el riesgo de entregar algo sin considerar si realmente satisface las necesidades del cliente.
3. Enfocarse excesivamente en hacer el MVP lo más mínimo posible, descuidando su viabilidad. Un MVP debe ser tanto mínimo como viable para proporcionar una evaluación precisa.
4. Entregar un producto como MVP y luego no realizar cambios, incluso si se reciben comentarios negativos. Es importante estar dispuesto a adaptar el producto en función de los comentarios y aprendizajes obtenidos.

LA FASE CONSTRUIR DEL MVP



En esta metodología, el **ciclo de Feedback** o de **Aprendizaje** es **esencial para el éxito**. No basta con entregar un producto, sino con entregar un producto o característica que el **cliente realmente utilizará**. Esto se logra alinear constantemente los esfuerzos con las necesidades reales de los clientes. **El ciclo "Construir-Experimentar-Aprender"** permite descubrir y abordar sistemáticamente las necesidades del cliente, garantizando un enfoque centrado en el usuario y la mejora continua del producto.

Cuando se construye un MVP, su complejidad puede variar, desde anuncios y explicaciones simples hasta prototipos tempranos. El objetivo es desarrollar rápidamente el MVP con el menor esfuerzo posible para validar y retroalimentar la hipótesis en el ciclo de aprendizaje. Para minimizar el desperdicio y el esfuerzo al crear el MVP, es importante simplificar en caso de duda, evitar la construcción y promesa excesiva, y reconocer que cualquier trabajo adicional más allá de lo necesario para iniciar el ciclo podría ser un desperdicio.

La fase de experimentación se vuelve esencial en el ciclo de Feedback, especialmente al trabajar en la construcción del MVP. Esto plantea varias consideraciones importantes al crear un nuevo producto.

AUDACIA DE CERO (incentivo)

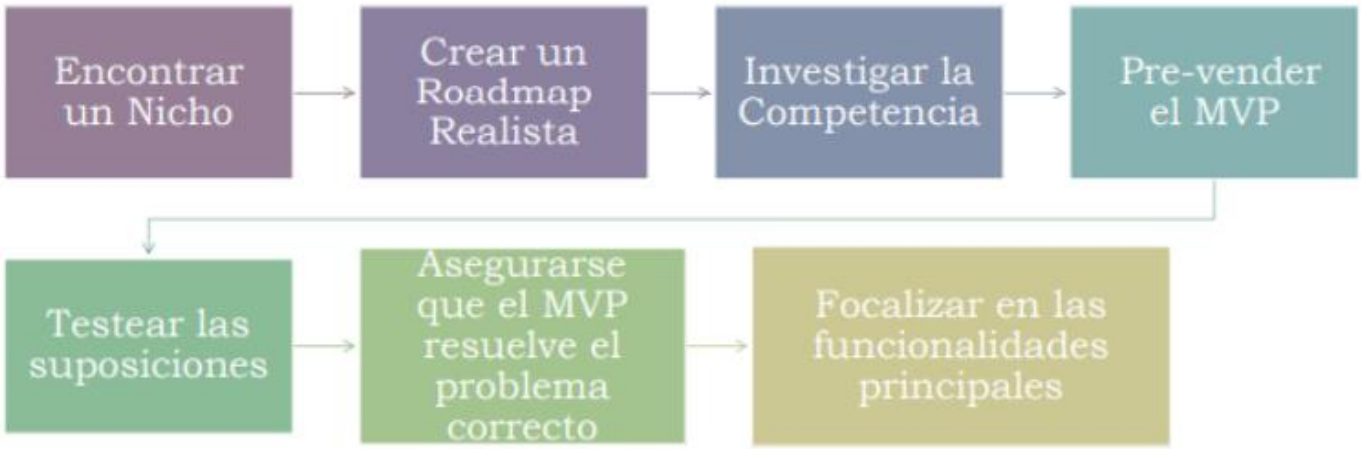
Es fundamental adoptar una **mentalidad audaz** desde el inicio del desarrollo de un producto, incluso si esto **conlleva riesgos**. Postergar la experimentación con el MVP puede resultar en una pérdida significativa de trabajo y datos esenciales, además de aumentar el riesgo de crear algo poco significativo.

El uso del MVP **permite llevar a cabo experimentos**, inicialmente de manera discreta, con los primeros usuarios en el mercado. Esto implica **verificar las hipótesis y probar todos los elementos disponibles**, comenzando por los más riesgosos.

SUPUESTOS DE "SALTOS DE FE

Se refieren a los elementos más **arriesgados** en el plan o concepto de un nuevo producto o startup. A menudo, las personas no conocen una solución específica (o incluso un problema) hasta que la experimentan, y una vez que lo hacen, no pueden imaginar vivir sin ella. Es **esencial abordar estos supuestos arriesgados** desde el principio y probarlos con el MVP.

PREPARAR UN MVP



SOFTWARE CONFIGURATION MANAGEMENT (SCM)

La **gestión de configuración de software (SCM)** es fundamental para **manejar todos los aspectos** relacionados con el software, incluyendo **herramientas, procesos y entorno**. El software se compone de **información estructurada** con **propiedades lógicas** y **funcionales,** y se crea y mantiene en diversas formas y representaciones para ser procesado por computadoras.
Los sistemas de software **experimentan cambios** durante su desarrollo y uso, lo que da lugar a la creación de nuevas **versiones** del sistema. Estas versiones **deben gestionarse y mantenerse** para mantener la **trazabilidad** de los cambios realizados en cada una de ellas. Los cambios en el software pueden originarse en **modificaciones en el negocio**, **nuevos requisitos**, **alteraciones** en productos asociados, **reorganización de prioridades**, **cambios presupuestarios**, **corrección de defectos** y **oportunidades de mejora**.
La gestión de configuración de software asegura la **integridad del producto** de software a lo largo de su ciclo de vida. Es una **actividad de soporte** que abarca todo el proyecto y que implica **dirección y monitoreo** administrativo y técnico. Su objetivo es **garantizar que el producto se mantenga íntegro** y de **alta calidad.**

ACTIVIDADES

La gestión de configuración de software (SCM) tiene como **propósito establecer y mantener la integridad** de los productos de software a lo largo de su ciclo de vida. Para lograr esto, se llevan a cabo diversas **actividades** que incluyen:

1. Identificar características técnicas y funcionales de ítems de configuración.
2. Documentar estas características técnicas y funcionales de los ítems de configuración.
3. Controlar los cambios que afectan a las características identificadas y documentadas.
4. Registrar y reportar los cambios realizados.
5. Verificar la correspondencia de los cambios con los requerimientos, manteniendo la trazabilidad.

APLICACIONES

La SCM encuentra aplicaciones en diversas disciplinas, como el **control de calidad de proceso**, el **control de calidad de producto** y la **prueba de software**. Su objetivo es resolver problemas de diferente índole mediante el establecimiento y el mantenimiento de la integridad del producto de software en todas sus etapas.

PROBLEMAS

Algunos de los **problemas** que la SCM ayuda a abordar incluyen:

- **pérdida de componentes**
- **pérdida de cambios** (cuando no se mantiene la última versión del componente)
- **regresión de fallas**, **doble mantenimiento** (realizar cambios en varios lugares)
- **superposición de cambios** (varios cambios realizados simultáneamente sin validación).

INTEGRIDAD

La integridad asegura la calidad y la confiabilidad del producto de software. La calidad es subjetiva y difícil de medir debido a la naturaleza intangible del software. La integridad del producto implica hacer explícitas las expectativas del cliente sobre el software.

La integridad de un producto de software se mantiene cuando cumple con los siguientes aspectos:

1. **Satisface las necesidades del usuario** y realiza las funciones esperadas.
2. **Permite la trazabilidad durante todo su ciclo de vida**, lo que implica mantener vínculos entre los ítems de configuración para analizar el impacto de los cambios.
3. **Cumple con criterios de rendimiento y requisitos no funcionales**.
4. **Satisface las expectativas de costos**, incluyendo la rentabilidad del desarrollo del producto y la gestión de configuración que es responsabilidad de todo el equipo, con un enfoque adicional del Gestor de Configuración en tareas específicas.

CONCEPTOS GENERALES

ÍTEM DE CONFIGURACIÓN

Los ítems de configuración son elementos esenciales en un proyecto o producto de software que pueden tener diferentes tipos, como código fuente, documentos, datos de prueba, etc. Estos ítems se almacenan en un repositorio y pueden cambiar con el tiempo. Su objetivo es mantener un registro de su estado y evolución a lo largo del ciclo de vida del proyecto o producto. Ejemplos de ítems de configuración incluyen prototipos de interfaz, manuales de usuario, código fuente y más. Su duración puede variar según el proyecto o sprint. Son fundamentales para la gestión y control de la configuración del software.

REPOSITORIO

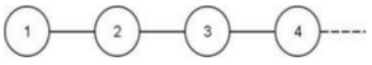
Un repositorio es un lugar donde se almacenan ítems de configuración junto con su historial, atributos y relaciones. Puede ser una base de datos o varias bases de datos organizadas con una estructura que asegura orden e integridad. Hay dos tipos principales de repositorios:

1. **Repositorio Centralizado:** Un servidor contiene todos los archivos con sus versiones. Proporciona un mayor control a los administradores, pero si el servidor falla, puede haber problemas.

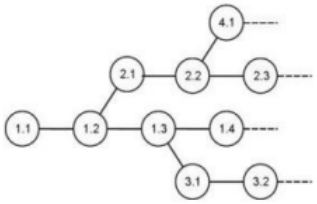
2. **Repositorio Descentralizado:** Cada cliente tiene una copia idéntica del repositorio completo. Resuelve problemas de servidores centralizados y permite ciertos flujos de trabajo no disponibles en el modelo centralizado. Sin embargo, puede ser más difícil de controlar.

VERSIÓN

Una versión es una instancia de un ítem de configuración que se diferencia de otras instancias de ese mismo ítem. Cada versión se identifica de manera única y representa un punto específico en el tiempo o contexto de ese ítem de configuración. La gestión de configuración permite a los usuarios seleccionar las versiones adecuadas para especificar configuraciones alternativas del sistema de software. Cada versión de software comprende una colección de elementos de configuración, como código fuente, documentos y datos. El **control de versiones** se enfoca en la evolución de un ítem de configuración particular y puede representarse gráficamente en forma de grafo para visualizar los cambios a lo largo del tiempo.



VARIANTE



Una variante es una versión separada de un ítem de configuración o de la configuración en sí misma. Estas variantes representan configuraciones alternativas que permiten que un producto de software pueda adaptarse a diferentes contextos, como plataformas de hardware y sistemas operativos específicos, así como funciones opcionales que pueden activarse o desactivarse según las necesidades.

CONFIGURACIÓN DE SOFTWARE

Una configuración de software es un conjunto que incluye todos los ítems de configuración con sus versiones específicas en un momento particular. Esta configuración representa una instantánea que abarca todos los componentes fuentes, documentos y la información estructural que definen una versión específica del producto en un punto en el tiempo.

LINEA BASE

Una línea base en la gestión de configuración de software es un conjunto de ítems de configuración que han sido construidos y revisados formalmente. Estos ítems se consideran como referencia para demostrar que han alcanzado un cierto nivel de madurez y sirven como base para desarrollos futuros. Las líneas base deben tener una referencia única, generalmente a través de "tags".

El propósito principal de las líneas base es proporcionar un punto de referencia sólido en el ciclo de vida del software. Además, se utilizan para realizar "Rollback" en caso de necesidad, es decir, volver a una versión anterior del software. También son útiles para determinar qué versión se va a poner en producción. Permiten rastrear la evolución del software a lo largo del tiempo y reproducir el entorno de desarrollo en momentos específicos del proyecto.

Existen dos tipos de líneas base: **las de especificación**, que incluyen requerimientos y diseño, y **las operacionales**, que contienen una versión ejecutable del producto que ha pasado por un control de calidad previo.

La primera línea base operacional corresponde a la primera ***"Release",*** que es una entrega del sistema que se libera para su uso por parte de los clientes u otros usuarios de la organización.

RAMA (BRANCH)

Las ramas en la gestión de configuración de software son conjuntos de ítems de configuración con sus respectivas versiones que permiten dividir el desarrollo de un software en diferentes direcciones. Esto puede hacerse por varias razones, como experimentación, corrección de errores en el desarrollo o para adaptar el software a diferentes plataformas.

La integración de ramas o "merge" se produce cuando se combinan dos ramas para fusionar la configuración de los ítems que las componen, incluyendo sus respectivas versiones. Es una buena práctica mantener la rama principal como la versión estable y fusionar los cambios hacia ella. En caso de que no se integren en la rama principal, se recomienda descartar las ramas para mantener la organización y evitar confusiones.

ACTIVIDADES RELACIONADAS A LA SCM

Todas las actividades que vamos a mencionar se realizan en ambas metodologías (tradicional y ágil), menos las auditorías, ya que el concepto de ser auditado y controlado por alguien externo no es compatible con una metodología ágil pura.

IDENTIFICACIÓN DE ÍTEMS DE CONFIGURACIÓN

La identificación de ítems de configuración es un proceso que implica **asignar una identificación única a cada ítem** y establecer **reglas de nomenclatura** y **versionado** en el equipo. También incluye **definir la estructura del repositorio** y dónde se ubicarán los ítems de configuración dentro de esa estructura.
Este proceso es esencial porque **proporciona una conexión** a lo largo de todas las etapas del **ciclo de vida del software**, lo que permite a los desarrolladores mantener el control y la integridad del producto, al tiempo que permite a los clientes evaluar esa integridad.
Es importante considerar **la duración de la integridad** de los ítems al identificarlos, ya que **varía** según el tiempo que sea necesario mantenerlos. Los ítems de configuración se clasifican en tres **categorías**:

1. Ítems de producto: Tienen un **ciclo de vida más largo** y se mantienen mientras el producto exista. Ejemplos incluyen documentos de arquitectura, casos de uso, código fuente, manuales de usuario, casos de prueba y bases de datos.
2. Ítems de proyecto: Tienen un **ciclo de vida que coincide con el proyecto** y se mantienen durante su duración. Ejemplos son el plan de proyecto y listados de defectos encontrados.

3. Ítems de iteración: Su **ciclo de vida corresponde a una iteración** específica del proyecto. Ejemplos incluyen planes de iteración, cronogramas de iteración y reportes de defectos.

CONTROL DE CAMBIOS

El control de cambios es un **procedimiento formal** que surge a partir de un **requerimiento de modificación** en uno o varios ítems de configuración que se encuentran en una **línea base establecida.** Una vez que se define una línea base, no se pueden realizar cambios en ella sin seguir un proceso formal de control de cambios. Este control se aplica a los ítems de configuración que forman parte de la línea base, ya que todos los involucrados en el desarrollo de software los utilizan como referencia.
El proceso de control de cambios es llevado a cabo por un **comité de control de cambios**, compuesto por diversos actores, que evalúa y decide si autoriza o no los cambios propuestos. Este comité puede incluir al **líder del proyecto, analistas, arquitectos e incluso el cliente**, dependiendo de la naturaleza del cambio propuesto.
Las **etapas del proceso de control de cambios** incluyen:

1. **Recepción de la propuesta** de cambio, que se refiere a una línea base en lugar de un ítem específico.
2. **Análisis de impacto** del cambio por parte del comité, **evaluando aspectos técnicos**, **gestión de recursos**, **efectos secundarios** y el **impacto global** en la funcionalidad y arquitectura del producto.
3. En caso de autorizarse el cambio, se **genera una orden de cambio** que detalla lo que se debe realizar, las restricciones a considerar y los criterios para revisar y auditar.
4. Después de **aplicar el cambio**, el comité vuelve a intervenir para **aprobar la modificación** de la línea base y marcarla nuevamente como línea base, realizando una revisión de partes.
5. **Se notifica a los interesados** sobre los cambios realizados en la línea base.

El control de cambios proporciona trazabilidad entre los ítems de configuración, lo que permite identificar qué ítems se ven afectados por un cambio y garantiza que los cambios sean gestionados de manera formal y documentada.

AUDITORÍAS DE CONFIGURACIÓN

La auditoría de configuración es un **conjunto de controles autorizados** que se realizan en un momento específico durante el desarrollo del software. En este proceso, **un auditor externo e imparcial** revisa las líneas base para evaluar el estado del software y asegurarse de que se cumplan las normas y directrices establecidas por la gestión de configuración.
La auditoría de configuración **es objetiva** cuando el auditor es independiente de cualquier influencia o prejuicio con respecto a lo que está auditando. Además, **es independiente** en el sentido de que el auditor no tiene ninguna relación jerárquica, funcional o de empleo con el equipo o el elemento que está siendo auditado.
Este proceso es importante para garantizar la calidad e integridad del producto mientras se encuentra en construcción, y se considera una medida preventiva, ya que es más eficiente y económico identificar y corregir problemas en esta etapa. La auditoría de configuración complementa las revisiones técnicas y verifica si las tareas se están realizando según lo planeado en el plan de gestión de configuración.
Existen dos tipos de auditorías de configuración: **la auditoría física de configuración (PCA),** que se asegura de que lo que se establece en la línea base se haya logrado de manera efectiva, y **la auditoría funcional de configuración (FCA),** que evalúa la funcionalidad y el rendimiento de los elementos de configuración para garantizar que coincidan con los requisitos especificados.
La auditoría de configuración sirve a **dos procesos** fundamentales: **validación y verificación**. **La validación** se asegura de que cada elemento de configuración resuelva el problema adecuado según las necesidades del cliente, mientras que **la verificación** verifica que un producto cumple con los objetivos definidos en la documentación de las líneas base.

REPORTES E INFORMES DE ESTADO

Los **informes de gestión de configuración** proporcionan un mecanismo para **rastrear la evolución del sistema y la ubicación del software en comparación con la última versión publicada en la línea base**. Esto garantiza que todo el equipo trabaje con la versión más reciente y no con una versión obsoleta.
Estos informes **registran todos los cambios realizados** en las líneas base a lo largo del ciclo de vida del software. Dado que estos informes suelen contener grandes cantidades de datos, **se utilizan herramientas automatizadas para generarlos**. El objetivo principal es asegurarse de que la información sobre los cambios llegue a todos los involucrados en el proyecto.
Uno de los informes más comunes es el **informe de inventario**, que proporciona una **copia del contenido del repositorio** con la estructura de directorios. Estos informes **permiten responder preguntas** como el estado actual de un ítem, si una propuesta de cambio fue aceptada o rechazada por el comité, qué versión de un ítem implementa un cambio aprobado y cuál es la diferencia entre dos versiones de este ítem.

PLANIFICACIÓN DE LA SCM

El **plan de gestión de configuración** se elabora al comienzo de un proyecto y **sigue estándares que incluyen**:

- **Reglas para nombrar** ítems de configuración.
- **Selección de herramientas** SCM.
- **Definición de roles** y miembros del Comité de Control de Cambios.
- **Procedimientos formales** para gestionar cambios.
- **Procesos de auditoría**.
- **Estructura del repositorio**.
- **Consideración de SCM para software externo** (opcional).
- **Métodos para el control de cambios**.
- **Registros a mantener.**
- **Tipos de documentos a administrar**.

El plan se elabora temprano en el proyecto, asegurando que ningún producto del proceso quede sin administrar.

EVOLUCIÓN DE LA GESTIÓN DE CONFIGURACIÓN DE SOFTWARE

CONTINUOUS INTEGRATION

La **Integración Continua (CI)** es una práctica de desarrollo que implica integrar el código en un repositorio compartido varias veces al día y verificarlo automáticamente con pruebas. Ayuda a detectar problemas tempranamente. La integración continua es asegurar que el software pueda ser desplegado en cualquier momento, es decir, que el código compile y que sea de calidad. Cada desarrollador en su entorno de trabajo realiza pruebas unitarias (en la medida de lo posible, automatizadas) desarrollando con algo que se llama TDD (desarrollo conducido por pruebas) y cuando terminó ese componente de código y lo probó y sabe que funciona, lo sube a un repositorio de integración.

CONTINUOUS DELIVERY

La **Entrega Continua (CD)** asegura que el software pueda ser desplegado en cualquier momento y se enfoca en la automatización de pruebas de aceptación. El software está siempre listo para ser puesto en producción, pero la decisión de liberarlo depende de un responsable de negocio (PO). Hay que asegurarse de tener un despliegue automatizado, para que cada puesta en producción no lleve demasiado tiempo, lo que hará que las puestas puedan llegar a hacerse más seguidas, generando que lo que se despliegue no tenga demasiados cambios y el riesgo de que algo se rompa disminuya. El software debe estar siempre en un estado de “entregable”,

CONTINUOUS DEPLOYMENT

El **Despliegue Continuo (CD)** va un paso más allá y pone automáticamente el software en el ambiente de producción sin intervención humana. Se utiliza para que los cambios lleguen rápidamente a los usuarios finales y se minimice el tiempo entre actualizaciones.
Entre las estrategias de deployments, se encuentran Canary Deployment (lanzamiento de prueba) y Blue/Green Deployment (despliegue azul/verde), que son enfoques específicos para gestionar el despliegue de nuevas versiones de software de manera controlada y segura.

SCM EN AMBIENTES ÁGILES

En metodologías ágiles, **la gestión de configuración se centra en el seguimiento y coordinación del desarrollo** en lugar de controlar a los desarrolladores. **Se adapta a equipos auto organizados y ágiles**, por lo que las auditorías de configuración y otros procesos tradicionales se simplifican o relativizan.

Las responsabilidades de SCM en Agile incluyen hacer **seguimiento y coordinación, responder a los cambios en lugar de evitarlos**, **automatizar donde sea posible** (como en Integración Continua), **eliminar el desperdicio**, **proporcionar documentación Lean y trazabilidad**, y **ofrecer retroalimentación continua** sobre calidad, estabilidad e integridad.

En Agile, estas tareas son responsabilidad de todo el equipo y se integran en las actividades necesarias para lograr los objetivos del sprint.