El proceso de desarrollo de software

Introducción

Nos proponemos explicarte el proceso de desarrollo de software. Te contaremos **qué es el ciclo de vida** y te presentaremos los **principales modelos de desarrollo que se utilizan actualmente**. También te contaremos las disciplinas que intervienen en la Ingeniería de software.
Finalmente te contaremos **qué es la configuración de software** y cómo planear y realizar las **pruebas del software**.

Como hemos visto en el módulo **Técnicas de Programación**, un sistema puede llegar a ser muy complejo en funcionalidad y en cantidad de código fuente. Para poder administrar la complejidad de tales sistemas, es necesario contar con modelos de procesos y tecnologías de software apropiadas.
Te presentaremos distintos modelos de procesos y veremos cuáles son sus características, ventajas y desventajas.

El proceso de desarrollo de software

Seguramente habrás oído hablar muchas veces de proceso histórico, de proceso productivo, proceso lógico, proceso natural, etc.

¿Qué es para vos un proceso?
Habrá muchas formas de definir este concepto, pero seguramente tu definición se asemejará a ésta que te proponemos:

"Un proceso es un conjunto de actividades planificadas que implican la participación de un número de personas y de recursos materiales coordinados para conseguir un objetivo previamente identificado".

Un proceso entonces define **quién hace qué, cuándo y cómo** lo hace para poder alcanzar un objetivo.

Estudiaremos el proceso de desarrollo ya que, en general, el éxito de las organizaciones depende en gran medida de la definición y el seguimiento adecuado de sus procesos.

En nuestro caso, será de interés una empresa que se dedique al desarrollo de software y para eso requerirá procesos especializados que abarquen desde la creación hasta la administración de un sistema de software.

El Ciclo de Vida de Desarrollo de Software

Como te comentamos en la Unidad 1 de la materia Técnicas de Programación, el desarrollo de un sistema se realiza durante todo el **ciclo de vida**, que es el período de tiempo que se extiende desde la idea original del problema a resolver hasta el mantenimiento y desarrollo de las mejoras.

Análisis del problema

En esta etapa se debe determinar cuál es el problema a resolver y los límites y alcances que tendrá el software que lo resolverá. Es el momento de reunirse con quien nos solicita el programa para saber cuáles son los requerimientos.

Especificación del software

En este momento los profesionales de sistemas se encargan de definir las entradas y las salidas del software y, qué restricciones tendrán los datos. También se describen los componentes que se deberán desarrollar, qué características y comportamiento tendrán y cómo estarán relacionados.

Desarrollo del software

Corresponde al proceso de construcción de software propiamente dicho. Es en esta etapa en donde los programadores escriben el código fuente utilizando algún lenguaje de programación.

Verificación del software

Una vez que el software está desarrollado se debe probar para verificar que responde a las definiciones y no tiene errores.

Mantenimiento del software

Todo sistema debería tener mantenimiento ya que siempre habrá que realizar alguna modificación, agregando nueva funcionalidad o bien cambiando alguna característica porque se ha modificado alguna especificación.

El siguiente esquema nos recuerda el ciclo de vida del software:

Modelos de Desarrollo de Software

Como seguramente recordarás, no existe un único modelo de **Ciclo de Vida**. Más allá de que los distintos modelos tendrán diferentes características, todo proceso debe cubrir los siguientes objetivos.

- Definir las actividades a realizar y en qué orden.
- Establecer criterios de transición para pasar a la fase siguiente.
- Proporcionar puntos de control para la gestión del proyecto.
- Asegurar la consistencia con el resto de los sistemas de información.

A continuación, te presentamos algunos de los modelos de desarrollo más importantes.

1. Modelo de cascada

El **modelo de cascada** original se desarrolló entre las décadas de los años 60 y 70. Se define como una secuencia de actividades o etapas, donde la estrategia principal es seguir el progreso del desarrollo de software hacia puntos de revisión bien definidos mediante entregas programadas con fechas precisas. En este modelo, no se muestra una etapa específica de documentación, ya que ésta se lleva durante todo el proceso de desarrollo.

> En el modelo original se planteaba que cada actividad debía completarse antes de poder continuar con la siguiente actividad. Sin embargo, en una revisión posterior, se extendió el modelo permitiendo regresar a actividades anteriores.

Este modelo fue muy aceptado debido a que las etapas son lógicas y se comprenden fácilmente por los usuarios no técnicos. Pero este modelo no explica cómo modificar un resultado, en especial teniendo en cuenta lo difícil que puede llegar a especificar toda la funcionalidad, y los requisitos de un sistema desde el comienzo y que estos se mantengan estables durante todo el proceso.

Otra **desventaja** es que toma demasiado tiempo en tener un resultado tangible para el usuario y retrasa la detección de errores hacia las etapas finales del desarrollo.

El modelo establece las siguientes **etapas para el desarrollo de software**.

Análisis de requisitos

Consiste en la recopilación de los requisitos del software. Se debe comprender el ámbito de información del software, así como la función, el rendimiento y las interfaces requeridas. Estos requisitos se deben documentar y revisar de tal manera que los entiendan tanto los usuarios como el equipo de desarrollo del software. En esta fase se desarrollará el documento de requisitos del software que consistirá en una especificación precisa y completa de lo que debe hacer el sistema.

Diseño

Consiste en descomponer y organizar el sistema en elementos componentes que puedan ser desarrollados por separado. El resultado del diseño es la colección de especificaciones de cada elemento componente. En esta fase se desarrollará el Documento del diseño del Software que será una descripción de la estructura global del sistema.

Implementación

En esta fase se traduce el diseño a un lenguaje legible para una computadora. También se harán las pruebas o ensayos necesarios para garantizar que dicho código funciona correctamente. La documentación de esta fase será el Código fuente.

Verificación

Consiste en probar el sistema completo para garantizar el funcionamiento correcto del conjunto antes de ser puesto en producción. Aquí tendremos el Sistema Software ejecutable.

Mantenimiento

Puede ocurrir que durante el uso del sistema sea necesario realizar cambios para corregir errores que no han sido detectados en las fases anteriores o bien para introducir mejoras. Tendremos que hacer un Documento de cambios ante cualquier modificación. En todas estas fases la verificación y validación se han de tener en cuenta. La **verificación** consiste en comprobar que el software que se está desarrollando cumple los requisitos y la **validación** lo que hace es comprobar que las funciones del software son las que el usuario desea.

2. Modelo incremental

El **modelo incremental** consiste en el desarrollo inicial de la arquitectura completa del sistema, seguida de incrementos y versiones parciales. Cada incremento tiene su propio ciclo de vida, típicamente siguiendo el modelo de cascada. Los incrementos pueden construirse de manera serial o paralela dependiendo de la naturaleza de la dependencia entre versiones y recursos.

Cada incremento agrega funcionalidad adicional o mejorada sobre el sistema. Conforme se completa cada etapa, se verifica e integra la última versión con las demás versiones ya completadas del sistema. Durante cada incremento, el sistema se evalúa con respecto al desarrollo de versiones futuras.

Las actividades se dividen en procesos y subprocesos, dando lugar al término fábrica de software (en inglés, **software factory**).

¿Ventajas?

Este modelo tiene como ventaja que permite que la administración del proyecto sea más simple en incrementos pequeños. Además, al tener una menor funcionalidad por incremento, cada etapa es más simple de comprender y de probar. Otra ventaja es que el usuario tendrá una versión más temprana del sistema, ya que se encontrará con una pequeña funcionalidad para ir probando y verificando.

Con respecto a los cambios, este modelo permite resolverlos en un mismo período de tiempo. De esta manera el modelo es tolerante a modificaciones en los requerimientos, algo que hoy en día sucede con mucha frecuencia.

3. Modelo evolutivo

Ahora te presentamos el **modelo evolutivo**, el cual es una extensión del modelo incremental. En este modelo, los incrementos se hacen de manera secuencial, en lugar de en paralelo.

Desde el punto de vista del cliente, el sistema evoluciona según vayan siendo entregados los incrementos. Desde el punto de vista del desarrollador, los requerimientos que son claros al principio del proyecto determinan el incremento inicial, mientras que los incrementos para cada uno de los siguientes ciclos de desarrollo se definirán a través de la experiencia de los incrementos anteriores. Este modelo considera que el desarrollo de sistemas es un proceso de cambios progresivos mediante deltas (cambios) de especificación de requerimientos.

Este modelo también es conocido como **Desarrollo rápido de aplicaciones** (RAD en inglés, **rapid application development**) basado en prototipos.

Te contamos que **un prototipo de software** es un medio para especificar los requerimientos y comunicarlos al usuario. **El prototipo** será una representación limitada de un producto (en nuestro caso, el software) que permitirá que el usuario lo pueda probar en situaciones reales. De esta manera se crea un proceso de diseño de iteración que genera calidad.

Un prototipo puede ser cualquier cosa, desde un trozo de papel con sencillos dibujos a un módulo de software.

Los prototipos apoyan la evaluación del producto, clarifican los requisitos del usuario y definen alternativas.

Como **ventajas** te podemos decir que:

- Los requisitos de los usuarios son más fáciles de determinar y la Implementación del sistema será más sencilla debido a que los usuarios conocen lo que esperan.
- Los sistemas se desarrollan más rápidamente
- Facilita la comunicación con los usuarios ya que los prototipos son comprendidos tanto por el usuario como por el analista.

Dentro de los **inconvenientes** te podemos mencionar que:

- Puede crear falsas expectativas en el usuario ya que puede ver el prototipo como si fuera el producto final.
- Puede darse una fuerte intromisión de los usuarios finales en el momento del desarrollo.
- Pueden producirse inconsistencias entre el prototipo y el sistema final.
- El paradigma de prototipos no es apropiado para proyectos grandes y de larga duración ni para aplicaciones pequeñas (menos de un mes), siendo óptimo en aplicaciones y proyectos cuya duración esté fijada entre 3 y 5 meses.

4. Modelo espiral

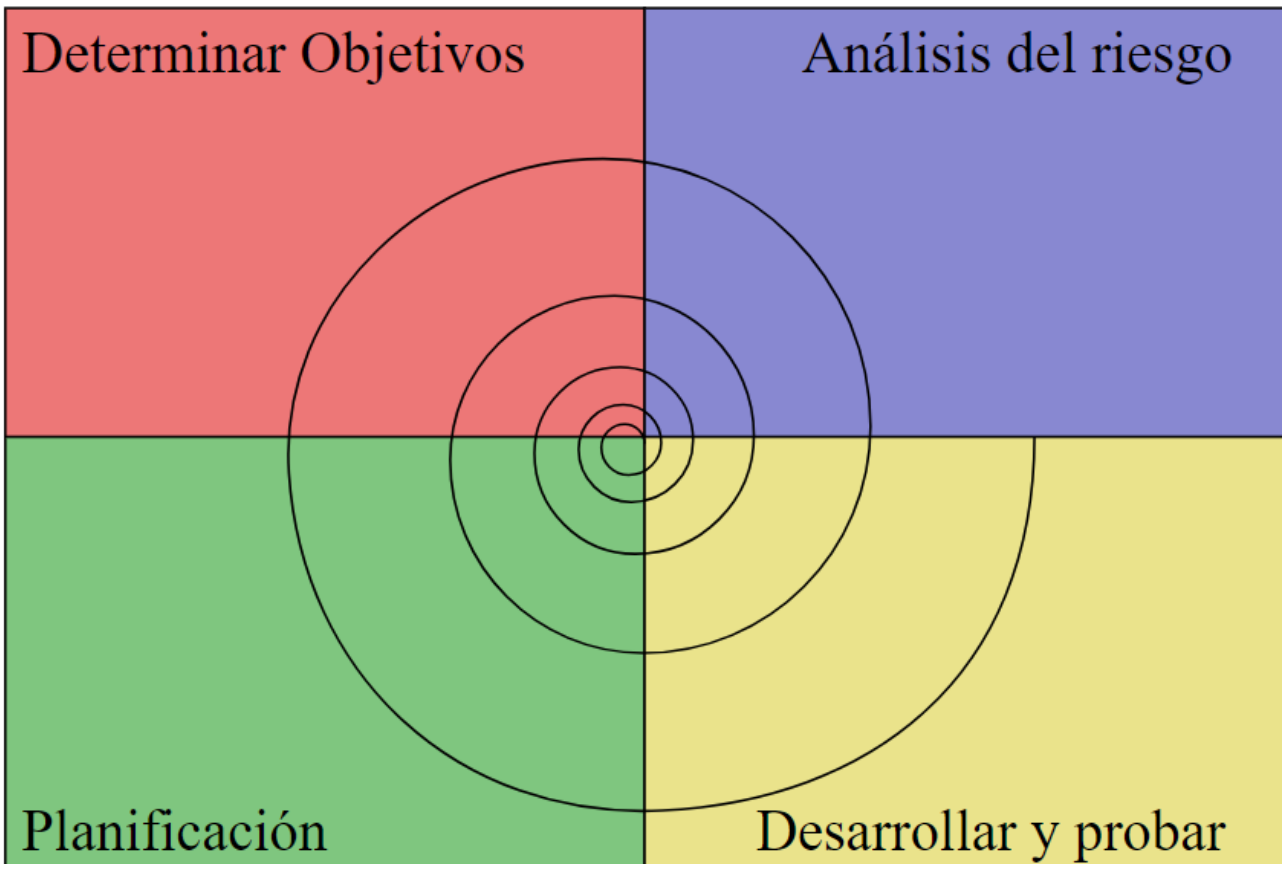
Este modelo fue desarrollado en la década del 80 y surgió como una extensión del modelo de cascada. El **modelo espiral** se basa en una estrategia para reducir el riesgo del proyecto en áreas de incertidumbre, como por ejemplo, tener requerimientos iniciales incompletos e inestables.

El modelo enfatiza ciclos de trabajo, cada uno de los cuales estudia el riesgo antes de proceder al siguiente ciclo. Cada ciclo comienza con la identificación de los objetivos, soluciones alternas y restricciones asociadas con cada alternativa y, finalmente, se procede a su evaluación.

Cuando se encuentra que existe cierta incertidumbre, se utilizan diversas técnicas para reducir el riesgo de las distintas alternativas. Cada ciclo del modelo espiral termina con una revisión que discute los logros actuales y los planes para el siguiente ciclo.

Esta secuencia de pasos se compone de **4 actividades**:

1_ Determinar Objetivos.
2_ Análisis del riesgo.
3_ Desarrollar y probar.
4_ Planificación.



Los **ciclos se repiten en forma de espiral**, comenzando desde el centro. Se suele interpretar como que dentro de cada ciclo de la espiral se sigue un Modelo Cascada, pero no necesariamente debe ser así.

Como **ventajas** te podemos decir que

- Este modelo se puede ajustar a un amplio rango de opciones.
- Su orientación al riesgo evita, y hasta elimina, muchas de las posibles dificultades.
- Se centra en la eliminación de errores.
- Permite incorporar objetivos de calidad en el desarrollo software.
- Se adapta bien al diseño Orientado a Objetos.

Dentro de los **inconvenientes** podemos mencionar que

- Depende de manera excesiva de la experiencia que se tenga en identificar y evaluar riesgos.
- Es difícil adaptarlo al software contratado debido a la poca flexibilidad y libertad que posee un software de estas características.

Disciplinas que conforman la Ingeniería de Software

Durante el proceso de desarrollo de software, como te habrás dado cuenta, intervienen muchas personas con distintos roles que realizan actividades diferentes.

¿Qué son las disciplinas entonces? ¿Por qué hablamos de disciplinas que conforman el desarrollo de Software?

Las **disciplinas** son precisamente eso: u**n conjunto de actividades relacionadas con un área de atención dentro de todo el proyecto.**

De esta manera podemos observar que durante el ciclo de desarrollo intervienen más de una disciplina. Entre las principales disciplinas que intervienen podemos mencionarte las siguientes:

Modelado de Negocios

El propósito de esta disciplina es entender el contexto del cliente, y la aplicación que tendrá el sistema dentro del mismo. Las actividades del modelado de negocios más comunes son la realización de:

- Modelado de contexto. Esto es, mostrar cómo tiene cabida tu sistema dentro del ambiente general.
- Modelado de requerimientos de negocio de alto nivel (casos de uso).
- Glosario definiendo los términos críticos del negocio.
- Modelado de dominio, describiendo las entidades principales del negocio.

Requerimientos
Aquí se aplican métodos de ingeniería a los requerimientos del proyecto, incluyendo la identificación, modelado y documentación de dichos requerimientos. El entregable principal de esta disciplina es el documento donde se especifican los Requerimientos del Software.

El rol principal lo cubre el Analista de Sistemas, quien se encarga de descubrir todos los casos de uso de requerimientos.

Análisis y Diseño
El propósito de esta disciplina es crear una arquitectura robusta para el sistema basada en los requerimientos del cliente, transformar dichos requerimientos en un diseño, y asegurarse de que los problemas del ambiente dentro del cual se implementará el sistema se reflejen en el diseño.

En esta disciplina interviene el Arquitecto de Software, quien decide las tecnologías para toda la solución. Y el Diseñador, quien detalla el análisis y diseño de un grupo de casos de uso.

Implementación
Esta disciplina define la organización del código, implementa el diseño de elementos, prueba los componentes desarrollados como unidades e integra los resultados individuales en un sistema ejecutable. En esta disciplina es donde aparece el Desarrollador, quien tiene el plan de construcción que muestra qué entidades o clases se integrarán unas a otras. También se encargará de codificar un grupo de clases u operaciones de clases.

Pruebas
Esta disciplina es la encargada de evaluar todos los componentes del sistema y de asegurar la calidad del producto desarrollado. Las actividades que se llevan a cabo son:

- Encontrar fallas de calidad en software y documentarlas.
- Dar retroalimentación sobre la calidad percibida en el software.
- Validar y probar que se han cubierto los requerimientos y el diseño del sistema.

Es acá donde interviene el Administrador de Pruebas, quien se asegura de que las pruebas se completen y sean realizadas bajo las especificaciones correctas. También interviene el Analista de Pruebas, quien selecciona qué probar; el Diseñador de Pruebas, quien decide qué pruebas deben ser automatizadas y cuáles manuales; y el tester, quien ejecuta una prueba específica.

Transición
En esta disciplina se describen las actividades asociadas con el aseguramiento de la entrega y disponibilidad del producto de software hacia el usuario final.

Interviene el Administrador de transición, quien se encarga de prever que todas las unidades del sistema tengan una transición exitosa. Además, intervienen otros roles como Escritor Técnico, Desarrollador de Curso, Artista Gráfico, quienes crean los materiales detallados para asegurarse de una transición exitosa.

Administración y configuración del cambio
Esta disciplina consiste en controlar los cambios y mantener la integridad de los productos que incluye el proyecto. Interviene el Administrador de Configuración, quien prepara el ambiente, las políticas; el Administrador de Control de Cambio, quien establece un proceso de

control de cambio; y el Administrador de Control de Cambio, quien revisa y administra las solicitudes de cambios.

Administración de Proyectos

Es la disciplina que provee un marco de trabajo para administrar los proyectos intensivos de software, además de guías prácticas para el desarrollo del mismo. El Administrador del Proyecto es quien crea los casos de negocio y un plan general para todo el proyecto. También planea, rastrea y administra los riesgos para cada iteración.

Configuración de Software

Evolución del software

Con la experiencia que habrás tenido en el cursado de los otros módulos, habrás notado que el software es un producto que se realiza en base a una especificación y que luego puede ser modificado, ya sea porque esas especificaciones cambian o bien por algún otro motivo incluso ajeno al cliente.

De esta manera, debemos agregar una **nueva etapa de Mantenimiento al proyecto** una vez que el producto ha sido terminado. Esta **Fase de Mantenimiento** consiste en repetir o realizar parte de las actividades de las fases anteriores para introducir cambios en una aplicación del software ya entregada al cliente.

El mantenimiento lo podemos clasificar en **tres tipos distintos**:

Mantenimiento Correctivo

Tiene como finalidad corregir errores en el software que no han sido detectados y eliminados durante el desarrollo inicial del mismo.

Mantenimiento Adaptativo

Se produce en aplicaciones que se están utilizando desde hace mucho tiempo, de manera que los elementos básicos HW y SW que constituyen la plataforma o entorno en que se ejecutan evolucionan, modificándose parcialmente la interfaz ofrecida a las aplicaciones que corren sobre ellos.

Mantenimiento Perfectivo

Es necesario para obtener versiones mejoradas del producto que permitan mantener o aumentar su éxito.

El **mantenimiento**, por lo tanto, es una tarea que puede durar mucho tiempo y abarcar distintas etapas, es por eso que se necesita una buena gestión sobre los cambios y el testing.

Gestión de la configuración

Seguramente te estarás preguntando **qué es configurar software.**
La configuración es la manera en que diversos elementos se combinan para construir un software bien organizado.

Para mantener bajo control la configuración del software nos vamos a apoyar en dos técnicas que son:

Control de versiones: consiste en almacenar de forma organizada las sucesivas versiones de cada elemento de la configuración.

Control de cambios: consiste en garantizar que las diferentes configuraciones del software se componen de elementos compatibles entre sí.

Recordemos que estamos trabajando bajo una metodología de desarrollo. Debido a eso es preciso llevar un control y registro de los cambios con el fin de reducir errores, aumentar la calidad y productividad.

El objetivo de la **gestión de la configuración** es el de mantener la integridad de los productos que se obtienen a lo largo del desarrollo del sistema, garantizando que no se realizan cambios incontrolados y que todos los participantes en el desarrollo del sistema disponen de la versión adecuada de los productos que manejan.

Esta **gestión de la configuración se realiza** durante todas las actividades asociadas al desarrollo del sistema.

Desde el punto de vista del usuario y del desarrollador se consideran como **elementos componentes de la configuración** todos los que intervienen en el desarrollo. Estos elementos serán, por tanto:

- **Documentos del desarrollo.**
- **Código fuente de los modelos.**
- **Programas, datos y resultados de las pruebas.**
- **Manuales del usuario.**
- **Documentos de mantenimiento.**
- **Prototipos.**
- **Código fuente de los modelos.**
- **Programas, datos y resultados de las pruebas.**
- **Manuales del usuario.**
- **Documentos de mantenimiento.**
- **Prototipos.**

Gestión de cambios

Más allá del objetivo concreto del mantenimiento, las actividades a realizar durante el mismo son básicamente la realización de cambios significativos sobre el software ya realizado.

Podemos distinguir **2 enfoques diferentes** en cuanto a la gestión de cambios, en función del mayor o menor grado de modificación del producto. De esta manera tenemos:

- Si el cambio a realizar afecta a la mayoría de los componentes de software se puede plantear como un nuevo desarrollo y aplicar un nuevo ciclo de vida utilizando lo ya desarrollado (reutilización).

- Si el cambio afecta a una parte localizada del producto entonces se puede organizar como una simple modificación de algunos elementos.

Normas y estándares

Al igual que muchas otras actividades de ingenierías que están reguladas, en la ingeniería de software también hay normas, algunas han sido recogidas por organizaciones internacionales y establecidas como estándares.

Te mostramos ejemplos:

- **ISO:** Son las siglas del organismo internacional de normalización. El organismo español que lo integra es AENOR. Entre sus normas de Ingeniería del software de mayor nivel se encuentran las **ISO-9001** que establecen los criterios que han de cumplir las empresas que desarrollen software para obtener certificaciones de determinados niveles de garantía de calidad para su producción.

- **MÉTRICA 3:** Ofrece a los organismos oficiales un instrumento útil para la sistematización de las actividades que dan soporte al ciclo de vida del software.

Testing de Software

La calidad de un software se determina por la etapa que sigue a su desarrollo. Un sistema no finaliza cuando termina de desarrollarse, ya que, en el modelo de ciclo de vida, las actividades de revisión y pruebas tienen como objetivo controlar la calidad del producto.

Factores de calidad

Existe un esquema de mediciones de la calidad del software establecido en 1977. Este modelo se utiliza para especificar software de calidad y está basado en valoraciones a tres niveles diferentes.

Factores. Constituyen el nivel superior y son la valoración propiamente dicha. Describen la visión externa del software, es decir, cómo es visto por el usuario.

Criterios. Describen la visión interna del software, es decir, cómo es visto por el desarrollador.

Métricas. Están en el nivel inferior, son mediciones puntuales de determinados atributos o características del producto.

Entre los **factores de calidad** te podemos mencionar los siguientes:

• **Corrección**: Es el grado en que un software cumple con sus especificaciones. Se puede definir como el porcentaje de requisitos que se cumplen adecuadamente.

• **Fiabilidad**: Es el grado de ausencia de fallos durante la operación del software; puede estimarse como la cantidad de fallos producidos o tiempo durante el que permanece inutilizable.

• **Eficiencia**: Relación entre la cantidad de resultados suministrados y los recursos requeridos durante la operación.

• **Seguridad**: Dificultad para el acceso a los datos u operaciones por parte de personal no autorizado

• **Facilidad de uso**: Es la inversa del esfuerzo requerido para aprender a usar un producto software y poder utilizarlo adecuadamente.

• **Mantenibilidad**: Facilidad para corregir el software.

• **Flexibilidad**: Facilidad para modificar el software.

• **Facilidad de prueba**: Es la inversa del esfuerzo requerido para probar un software y comprobar su corrección o fiabilidad

• **Transportabilidad**: Facilidad para adaptar el software a una plataforma de hardware y Sistema Operativo diferente a aquella para la que fue desarrollado inicialmente.

• **Reusabilidad**: Facilidad para emplear parte del software en otros desarrollos posteriores.

• **Interoperatividad**: Facilidad del software de trabajar en combinación con otros productos.

Plan de garantía de la calidad

La calidad del software se consigue mediante una buena organización del desarrollo. Como parte del proceso de desarrollo se deberá generar un documento llamado **Plan de garantía de calidad del software**. Este documento deberá contemplar los siguientes aspectos.

- Organización de los equipos de personas y la dirección y seguimiento del desarrollo.
- El modelo de ciclo de vida a seguir.
- Documentación requerida.
- Revisiones y auditorías que se llevarán a cabo durante el desarrollo para garantizar que las actividades y documentos son correctos y aceptables.
- Organización de las pruebas que se realizarán sobre el producto software a distintos niveles.
- Organización de las etapas de mantenimiento.

Revisiones

Te contamos que una **revisión** consiste en inspeccionar el resultado de una actividad para determinar si es aceptable o no.

Te mencionamos algunas recomendaciones que podrás seguir para formalizar las revisiones.

- Deben ser realizadas por un grupo de personas, no por una sola
- El grupo que realice la revisión debe ser reducido (3-5 personas).
- No deben ser realizadas por los autores del software y así poder garantizar la imparcialidad del criterio.
- Se debe revisar el producto, pero no al productor ni al proceso de producción.
- Si la revisión tiene que decidir la aceptación o no del producto, se debe establecer previamente una lista formal de comprobaciones a realizar
- Debe levantarse acta o minuta de la reunión de la revisión. Este documento puede considerarse como el producto de la revisión.

Pruebas

Las **pruebas** consisten en hacer funcionar un software o una parte de él en condiciones determinadas y comprobar si los resultados son los correctos. Por lo tanto, el **objetivo** de las pruebas será descubrir los errores que pueda contener el software probado.

Es importante que comprendas que las pruebas **no** permiten garantizar la calidad de un producto, **sino** la funcionalidad que se está probando.

Síntesis de la Unidad

En esta unidad te presentamos el **PROCESO DE DESARROLLO DE SOFTWARE**, que deriva en el **CICLO DE VIDA DE SOFTWARE**. Este ciclo puede ser implementado en algún **MODELO DE DESARROLLO** (cascada, incremental, evolutivo, espiral).

También, te contamos las disciplinas que intervienen en la **INGENIERÍA DE SOFTWARE**. Analizamos las diferentes **ETAPAS DEL DESARROLLO DE SOFTWARE** a saber:

- ANÁLISIS DEL PROBLEMA
- ESPECIFICACIÓN DEL SOFTWARE
- DESARROLLO DEL SOFTWARE
- VERIFICACIÓN DEL SOFTWARE
- MANTENIMIENTO DEL MISMO.

Finalmente, te contamos que esta última etapa se ocupa de las tareas de **CONFIGURACIÓN DE SOFTWARE**, definido por la **GESTIÓN DE CONFIGURACIÓ**N y la **GESTIÓN DE CAMBIOS**; y **TESTING DE SOFTWARE**, que vimos que está definido por los **factores de calidad**, las **revisiones** y las **pruebas**.

AUTOEVALUACIÓN:

Si se va a desarrollar un sistema muy complejo, no hace falta tener un proceso de desarrollo.

○ Verdadero

◉ Falso

Para poder administrar la complejidad de tales sistemas, es necesario contar con modelos de procesos y tecnologías de software apropiadas.

La respuesta correcta es 'Falso'

¿Cuáles de los siguientes son modelos de desarrollo de software?

Seleccione una o más de una:

☑ a. Evolutivo

Es un modelo de desarrollo de software.

☑ b. Incremental

Es un modelo de desarrollo de software.

☑ c. Cascada

Es un modelo de desarrollo de software.

☑ d. Espiral

Es un modelo de desarrollo de software.

e. Seguridad

Respuesta correcta

Las respuestas correctas son: Cascada, Incremental, Evolutivo, Espiral

La disciplina de Análisis y Diseño tiene como propósito:

a. **Definir la organización del código, implementar el diseño de elementos y probar los componentes desarrollados.**

b. **Controlar los cambios y mantener la integridad de los productos que incluye el proyecto.**

c. **Proveer un marco de trabajo para administrar los proyectos de software.**

d. **Crear una arquitectura robusta para el sistema basada en los requerimientos del cliente.**

Es el propósito de la disciplina de Análisis y Diseño.

La respuesta correcta es: **Crear una arquitectura robusta para el sistema basada en los requerimientos del cliente.**

¿Qué tipo de mantenimiento tiene como finalidad corregir errores en el software que no han sido detectados y eliminados durante el desarrollo inicial del mismo?

a. Mantenimiento Perfectivo

b. Mantenimiento Correctivo

Se realiza para solucionar los errores que no han sido detectados en el momento de la implementación.

c. Mantenimiento Adaptativo

Respuesta correcta

La respuesta correcta es: Mantenimiento Correctivo

Mediante las revisiones del sistema se podrá determinar si una actividad es aceptable.

Seleccione una:

Verdadero

Falso

Se utilizan las revisiones para inspeccionar el resultado de una actividad para determinar si es aceptable o no.

La respuesta correcta es 'Verdadero'

Metodologías ágiles de gestión de proyectos

Introducción

En esta unidad nos proponemos explicarte las metodologías ágiles de gestión y desarrollo de software. Te contaremos las diferencias que existen entre los modelos tradicionales y los ágiles. También veremos las historias de usuario junto con los roles que intervienen en el modelo ágil.
Finalmente te presentaremos el modelo ágil más utilizado en estos momentos, el **Scrum**, en donde veremos sus características, roles y componentes que lo integran.

En la Unidad 1 vimos cómo es el proceso de desarrollo de software y te presentamos los principales modelos para gestionar un proyecto. Todos esos modelos se corresponden con los denominados **modelos tradicionales**, ya que fueron desarrollados al inicio de la Ingeniería de software.
Ahora bien, la experiencia de estas últimas décadas en el desarrollo de software y los cambios en los ritmos de producción han traído como consecuencia que esos modelos muchas veces no se adaptan adecuadamente al ritmo de estos tiempos. Es por eso que ha surgido un nuevo modelo, que lo podemos denominar genéricamente como **Metodología Ágil,** que te presentaremos en esta unidad.

Gestión de Proyectos de Desarrollo Tradicional vs. Ágil

Habrás notado a lo largo de la experiencia en este curso, y también por lo que te habrán contado profesores y profesionales de la ingeniería de software que es muy difícil especificar los requisitos en una única y primera etapa.

Por la complejidad de muchas de las reglas de negocio que automatizamos cuando construimos software, es muy difícil saber qué software se quiere hasta que se trabaja en su implementación y se ven las primeras versiones o prototipos. También, te pasará que muchas veces ni siquiera el propio usuario estará seguro de los requisitos y te irá pidiendo modificaciones a medida que vaya evaluando el software. Resulta también muy difícil documentar de una única vez, antes de la codificación, un diseño que especifique de manera realista y sin variación, todas las cuestiones a implementar en la programación.

Las metodologías tradicionales se basaron en los procesos industriales y de construcción de elementos físicos. De esta manera, las ingenierías clásicas o la arquitectura necesitan seguir este tipo de ciclos de vida en cascada porque precisan mucho de un diseño previo a la construcción, exhaustivo e inamovible como puede ser los planos del arquitecto antes de empezar el edificio.

Como seguramente podrás deducir, una vez realizados los cimientos de un edificio será muy difícil que se vuelva a rediseñar el plano y se cambie lo ya construido.

Con el software no pasa lo mismo ya que, aunque se pretenda emular ese modo de fabricación, el software es algo que tendrá cambios y, por eso, deberás tener muy en claro que es casi imposible cerrar un diseño en una primera etapa para pasarlo a programación sin tener

que modificarlo posteriormente.
En nuestro caso, que desarrollamos software, por su naturaleza, es más fácil de modificar. Cambiar líneas de código tiene menos impacto que cambiar las columnas de un edificio ya construido. De ahí que estas nuevas metodologías ágiles se basan en nuevas propuestas que recomiendan construir rápido una versión software y modificarla evolutivamente.

Diferenciar el cómo se construye software del cómo se construyen los productos físicos es uno de los fundamentos de las metodologías ágiles

Ciclo de vida Ágil

Como seguramente te habrás imaginado, esta nueva metodología también tiene un ciclo de vida. Podríamos decir que sería un ciclo de vida iterativo e incremental, con iteraciones cortas (semanas) y en cada iteración no es necesario que haya fases lineales tipo cascada.

De esta manera, un **proyecto ágil** se podría definir como una manera de enfocar el desarrollo de software mediante un ciclo iterativo e incremental, con equipos que trabajan de manera altamente colaborativa y autoorganizados.

Un proyecto ágil es un desarrollo iterativo realizado por equipos colaborativos y autoorganizados.

A diferencia de ciclos de vida iterativos e incrementales más "relajados", en un proyecto ágil cada iteración **no** es un "mini cascada".

Esto es así porque el objetivo de **acortar al máximo las iteraciones** (normalmente entre 1 y 4 semanas) lo hace casi imposible. Cuanto menor es el tiempo de iteración más se solapan las tareas. Esto sucede hasta tal punto que muchas veces te podrá pasar que te encuentres en un proyecto donde, se esté diseñando, programando y probando de manera no secuencial, es decir solapada, durante una misma iteración.

Esa característica de simultaneidad de etapas implicará una máxima colaboración e interacción de los miembros del equipo. Los equipos tendrán que ser multidisciplinares, es decir, que no hay roles que sólo diseñen o programen, porque todos pueden diseñar y programar. También significa autoorganización, es decir, que en la mayoría de los proyectos ágiles no hay, por ejemplo, un único jefe de proyecto responsable de asignar tareas.

A modo de resumen, te podemos decir que un **proyecto ágil** lleva la iteración al extremo mediante estos **dos pasos:**

- **Dividir las tareas del proyecto en incrementos de corta duración** (según la metodología ágil, típicamente entre 1 y 4 semanas).
- **Desarrollar en cada iteración un prototipo operativo.** Al final de cada incremento se obtiene un producto entregable que es revisado junto con el cliente, posibilitando la aparición de nuevos requisitos o la perfección de los existentes, reduciendo riesgos globales y permitiendo la adaptación rápida a los cambios.

El Manifiesto Ágil

El 12 de febrero de 2001, un grupo de 17 destacados y conocidos profesionales de la ingeniería del software escribían en Utah el **Manifiesto Ágil**.

Entre ellos estaban *Ken Schwaber* y *Jeff Sutherland*, los creadores de algunas de las metodologías ágiles más conocidas en la actualidad, entre ellas **Scrum,** metodología que veremos en las próximas secciones. Su objetivo fue establecer los valores y principios que permitirían a los equipos desarrollar software rápidamente y respondiendo a los cambios que pudieran surgir a lo largo del proyecto.

Como te contamos al inicio de esta Unidad, se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

De esta forma se establecieron **cuatro valores ágiles**

Los Valores Ágiles

Individuos e interacciones
Valorar a los individuos y las interacciones del equipo de desarrollo sobre el proceso y las herramientas. Se tendrán en cuenta las buenas prácticas de desarrollo y gestión de los participantes del proyecto (siempre dentro del marco de la metodología elegida). Esto facilita el trabajo en equipo y disminuyen los impedimentos para que realicen su trabajo. Asimismo, compromete al equipo de desarrollo y a los individuos que lo componen.

Software funcionando
Desarrollar software que funciona más que conseguir una documentación exhaustiva. No es necesario producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante. Los documentos deben ser cortos y centrados en lo fundamental. La variación de la cantidad y tipo de documentación puede ser amplia dependiendo del tipo de cliente o del proyecto. El hecho de decir que la documentación es el código fuente y seguir esa idea sin flexibilidad puede originar un caos. El problema no es la documentación sino su utilidad.

Colaboración con el cliente
La colaboración con el cliente más que la negociación de un contrato. Es necesaria una interacción constante entre el cliente y el equipo de desarrollo. De esta colaboración depende el éxito del proyecto. Este es uno de los puntos más complicados de llevar a cabo, debido a que muchas veces el cliente no está disponible. En ese caso desde dentro de la empresa existirá una persona que represente al cliente, haciendo de interlocutor y participando en las reuniones del equipo.

Respuesta ante el cambio
Responder a los cambios más que seguir estrictamente un plan. Pasamos de la anticipación y la planificación estricta sin poder volver hacia atrás a la adaptación. La flexibilidad no es total, pero existen muchos puntos (todos ellos controlados) donde se pueden adaptar las actividades.

Los Principios Ágiles

De los **cuatro valores ágiles** que te presentamos previamente, surgen los **doce principios del manifiesto ágil**. Estos principios son características que diferencian un proceso ágil de uno tradicional.

Los principios definidos en el manifiesto ágil:

- Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
- Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
- Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
- Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
- Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
- El software funcionando es la medida principal de progreso.
- Los procesos ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
- La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
- La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
- A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para, a continuación, ajustar y perfeccionar su comportamiento en consecuencia.

Product owner

Ahora te presentamos un nuevo rol que surge en la metodología ágil, el **Product Owner.** Y para eso te contamos que el "**product owner**" (o **propietario del producto**) es aquella persona con una visión muy clara del producto que se quiere desarrollar, que es capaz de transmitir esa visión al equipo de desarrollo y, además, está altamente disponible para transmitirla.

La figura del **product owner** es clave en la planificación y seguimiento de un proyecto ágil. Es una figura que cuando no realiza correctamente su función el proyecto tiene un serio riesgo, y problema, llegando incluso a dejar de ser ágil, o incluso dejando de ser proyecto.

Es importante explicarte que este rol puede ser ocupado por una persona interna o externa a la organización. Entre sus principales responsabilidades te podemos mencionar:

- Ser el representante de todas las personas interesadas en el proyecto, ya sea internas o externas.

- Definir los objetivos del producto o proyecto.
- Colaborar con el equipo para planificar, revisar y dar detalle a los objetivos de cada iteración.

Historias de usuario

Ahora te vamos a presentar un componente que se utiliza en las metodologías ágiles, las **Historias de usuario.**

Para eso, te contamos que la descripción de las necesidades se realiza a partir de las historias de usuario (**user story**) que son, principalmente, lo que el cliente o el usuario quiere que se implemente; es decir, son una descripción breve, de una funcionalidad de software tal y como la percibe el usuario.

El concepto de historia de usuario tiene sus raíces en la metodología "**eXtreme Programming**" o **programación extrema**. Esta metodología fue creada por Kent Beck y descrita por primera vez en 1999 en su libro "eXtreme Programming Explained". No obstante, las historias de usuario se adaptan de manera apropiada a la mayoría de las metodologías ágiles teniendo, por ejemplo, un papel muy importante en la metodología que veremos en la siguiente sección, denominada Scrum.

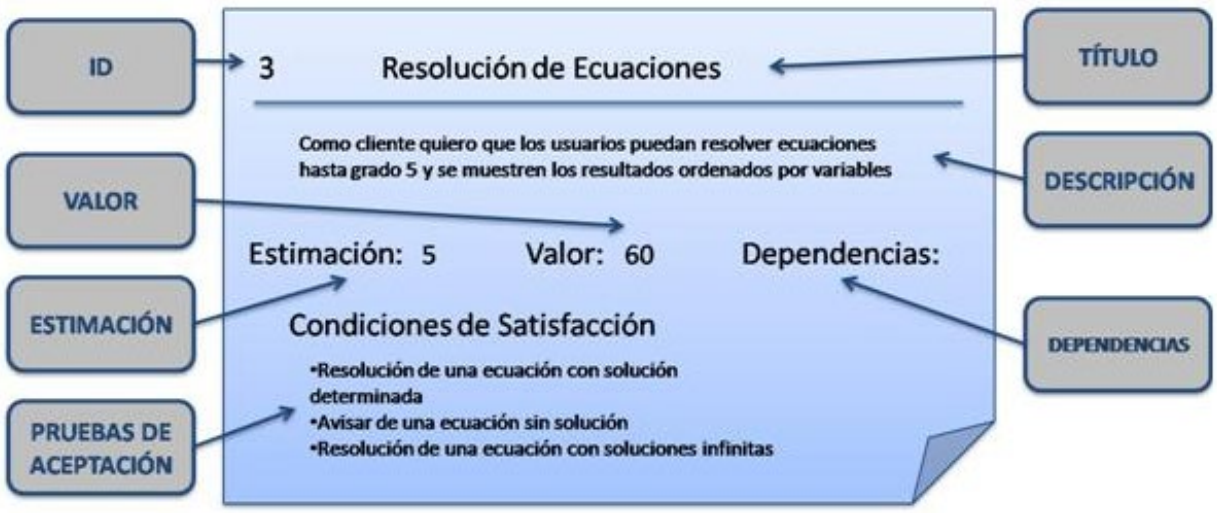
Es en este momento donde tiene una función preponderante el **Product owner,** ya que, en la mayoría de las ocasiones, el negocio o los usuarios, van proporcionando una cantidad de ideas a implementar, que se van convirtiendo en historias de usuario.

La función del **product owner** es vital, debe ser quien decida las historias de usuario que entrarán en el **product backlog (las historias que van a desarrollarse)** y cuáles no; además debe fijar la prioridad de las historias del product backlog.

Información de una historia de usuario

Ahora, te contamos aquellos campos que consideramos necesarios para describir de manera adecuada una historia de usuario. Esta es una lista orientativa ya que, dependiendo del proyecto, podrás incluir cualquier otro campo que proporcione información útil,

Estos campos los podés observar en la siguiente figura:



De esta manera, una historia de usuario está compuesta por los siguientes elementos:

- **ID**: identificador de la historia de usuario.
- **Título**: Leyenda descriptiva de la historia de usuario.
- **Descripción**: descripción sintetizada de la historia de usuario. Si bien el estilo puede ser libre, la historia de usuario debe responder a tres preguntas: ¿Quién se beneficia? ¿Qué se quiere? y ¿Cuál es el beneficio? Como ayuda te podemos recomendar que sigas el patrón: Como [rol del usuario], quiero [objetivo], para poder [beneficio]. Con este patrón se garantiza que la funcionalidad se captura a un alto nivel y que se está describiendo de una manera no demasiado extensa.
- **Estimación**: estimación del tiempo de implementación de la historia de usuario en unidades de desarrollo, conocidas como puntos de historia (estas unidades representarán el tiempo teórico de desarrollo/persona que se estipule al comienzo del proyecto).
- **Valor**: valor (normalmente numérico) que aporta la historia de usuario al cliente o usuario. El objetivo del equipo es maximizar el valor y la satisfacción percibida por el cliente o usuario en cada iteración. Este campo determinará junto con el tiempo, el orden con el que las historias de usuario van a ser implementadas.
- **Dependencias**: una historia de usuario no debería ser dependiente de otra historia, pero en ocasiones es necesario mantener la relación. En este campo se indicarían los identificadores de otras historias de las que depende.
- **Pruebas de aceptación**: pruebas consensuadas entre el cliente o usuario y el equipo, que deberán ser superadas para dar como finalizada la implementación de la historia de usuario. Este campo también se suele denominar "criterios o condiciones de aceptación".

Es importante que tengas en cuenta que, si bien las historias de usuario son lo suficientemente flexibles como para describir la funcionalidad de la mayoría de los sistemas, no son apropiadas para todo.
Si por cualquier razón, necesitas expresar alguna necesidad de una manera diferente a una historia de usuario, podrás utilizar las interfaces o pantallas de usuario.
Del mismo modo puede ocurrir con documentos de especificaciones de seguridad, normativas, etc.

No obstante, lo que sí es importante con esta documentación adicional es mantener la trazabilidad con las historias de usuario. Por ejemplo, a través de hojas de cálculo donde se lleve el control de a qué historia pertenece cada documento adicional, o especificando el identificador de la historia en algún apartado del documento, etc.

Malas interpretaciones sobre el concepto de historia de usuario

Seguramente estarás comparando las historias de usuarios con la documentación de requerimientos funcionales de la etapa de Análisis de los modelos tradicionales y estarás pensando que son lo mismo.
Estos conceptos no siempre quedan bien delimitados y nos puede llevar a la confusión, es por eso que ahora te respondemos algunas preguntas que seguramente te estarás haciendo.

***Las historias de usuario ¿equivalen a* requisitos funcionales?**

En general, se asocia el concepto de historia de usuario con el de la especificación de un requisito funcional. De hecho, muchas veces se habla de que a la hora de especificar una necesidad del cliente o del usuario, las metodologías ágiles usan la historia de usuario y las tradicionales, el requisito funcional. Sin embargo, detrás del concepto de historia de usuario hay muchos aspectos que lo diferencian de lo que es una especificación de un requisito, diferencias que muchas veces son poco conocidas y que llevan a muchos equipos a dudas y confusiones.

Una **historia de usuario** describe funcionalidad que será útil para el usuario o cliente de un software. Y aunque normalmente las historias de usuario asociadas a las *metodologías ágiles*, suelen escribirse en pósit o tarjetas, son mucho más que eso. Como te hemos comentado anteriormente, una historia no es sólo una descripción de una funcionalidad, sino también es de vital importancia para comunicar al usuario el progreso del desarrollo.

Las historias de usuario, frente a mostrar el **"cómo"**, sólo dicen el **"qué". ** Es decir, muestran funcionalidad que será desarrollada, pero no cómo se desarrollará. De ahí que aspectos como que "el software se escribirá en Java" **no** deben estar contenidos en una historia de usuario.

De esta manera, no se deben equiparar las historias de usuario con las especificaciones de requisitos ya que, por definición, **las historias de usuario no deben tener el nivel de detalle que suele tener la especificación de un requisito**.

Una **historia de usuario** debería ser **pequeña, memorizable,** y que pudiera ser desarrollada por un par de programadores en una semana. Debido a su brevedad, es imposible que una historia de usuario contenga toda la información necesaria para desarrollarla, en tan reducido espacio no se pueden describir aspectos del diseño, de las pruebas, normativas, convenciones de codificación a seguir, etc.

Para resolver el anterior problema hay que entender que el objetivo de las historias de usuario es, entre otros, lograr la interacción entre el equipo y el cliente o el usuario por encima de documentar, por lo que tanto no se deben sobrecargar de información.

...casos de uso?

Otro concepto que suele crear confusión sobre las historias de usuario son los casos de uso. Aunque hay quien ha logrado incluir casos de uso en su proceso ágil, no quiere decir que las historias de usuario sean equivalentes a los casos de uso.
Básicamente, si pensamos que una historia de usuario es el **"qué"** quiere el usuario, el caso de uso es un **"cómo"** lo quiere.

Generalmente, cuando un proyecto comienza a seguir una metodología ágil, se deberían olvidar completamente los casos de uso y el equipo debería centrarse en la realización de historias de usuario. Sin embargo, esto puede producir los siguientes problemas:

- **Las historias de usuario no proporcionan a los diseñadores un contexto desde el que trabajar.** Pueden no tener claro cuál es el objetivo en cada momento. ¿Cuándo le surgiría al cliente o usuario esta necesidad?
- **Las historias de usuario no proporcionan al equipo de trabajo ningún sentido de completitud.** Se puede dar el caso que la cantidad de historias de usuario no deje de aumentar, lo que puede provocar desmotivación en el equipo. Realmente, ¿qué tan grande es el proyecto?
- **Las historias de usuario no son un buen mecanismo para evaluar la dificultad del trabajo** que está aún por llegar.

Por lo tanto, si en un proyecto ocurre alguno de estos problemas se puede barajar la posibilidad de complementar las necesidades descritas en las historias de usuario con casos de uso, donde quede reflejado el comportamiento necesario para cumplir dichas necesidades.
En el caso de que se usen las historias de usuario y los casos de uso de manera complementaria, una historia de usuario suele dar lugar a la especificación de varios casos de uso.

Creando buenas historias de usuario

Para asegurar la calidad de una historia de usuario, en el año 2003 se definió un **método llamado INVEST.**

El método se usa para comprobar la calidad de una historia de usuario revisando que cumpla las características que te describimos a continuación:

Independient (independiente)

Es importante que cada historia de usuario pueda ser planificada e implementada en cualquier orden. Para ello debería ser totalmente independiente (lo cual facilita el trabajo posterior del equipo). Las dependencias entre historias de usuario pueden reducirse combinándolas en una o dividiéndolas de manera diferente.

Negotiable (negociable)

Una historia de usuario es una descripción corta de una necesidad que no incluye detalles. Las historias deben ser negociables ya que sus detalles serán acordados por el cliente/usuario y el equipo durante la fase de "conversación". Por tanto, se debe evitar una historia de usuario con demasiados detalles porque limitaría la conversación acerca de la misma.

Valuable (valiosa)

Una historia de usuario tiene que ser valiosa para el cliente o el usuario. Una manera de hacer una historia valiosa para el cliente o el usuario es que la escriba él mismo.

Estimable (estimable)

Una buena historia de usuario debe ser estimada con la precisión suficiente para ayudar al cliente o usuario a priorizar y planificar su implementación. La estimación generalmente será realizada por el equipo de trabajo y está directamente relacionada con el tamaño de la historia de usuario (una historia de usuario de gran tamaño es más difícil de estimar) y con el conocimiento del equipo de la necesidad expresada (en el caso de falta de conocimiento, serán necesarias más fases de conversación acerca de la misma).

Small (pequeña)

Las historias de usuario deberían englobar como mucho unas pocas semanas/persona de trabajo. Incluso hay equipos que las restringen a días/persona. Una descripción corta ayuda a disminuir el tamaño de una historia de usuario, facilitando su estimación.

Testable (comprobable)

La historia de usuario debería ser capaz de ser probada (fase "confirmación" de la historia de usuario). Si el cliente o usuario no sabe cómo probar la historia de usuario significa que no es del todo clara o que no es valiosa. Si el equipo no puede probar una historia de usuario nunca sabrá si la ha terminado o no.

Asignando valor a las historias de usuario

Hagamos un repaso, y para eso te recordamos que el **Product backlog** (también llamado **pila de producto**) es una lista de tareas que el equipo elabora en la reunión de **planificación de la iteración (Sprint planning)** como plan para completar los objetivos y requisitos seleccionados para la iteración.

Los ítems del **Product backlog,** deben estar priorizados, es decir, deben tener asignados un valor. Dicho valor es asignado por el ***Product Owner***, y pondera básicamente las siguientes variables:

- Beneficios de implementar una funcionalidad.
- Pérdida o costo que cause posponer la implementación de una funcionalidad.
- Riesgos de implementarla.
- Coherencia con los intereses del negocio.
- Valor diferencial con respecto a productos de la competencia.

Uno de los aspectos más importantes aquí es que la definición de **"valor"** para cada cliente puede ser distinta. Por lo tanto, te recomendamos que incluyas algún tipo de escala cualitativa.

Una manera rápida de empezar a asignar valor a las historias es dividirlas en 3 grupos, según sean **imperativas**, **importantes** o **prescindibles**. Dentro de cada grupo te resultará más fácil realizar una ordenación relativa por valor numérico y después asignarlo. Todo ello servirá para que en cada iteración se entregue el producto al cliente maximizando su valor.

Existen otro tipo de ponderaciones, por ejemplo, la **técnica MoSCoW.** Esta técnica fue definida en el año 2004. Su fin es obtener el entendimiento común entre cliente y el equipo del proyecto sobre la importancia de cada requisito o historia de usuario. La clasificación es la siguiente:

M - MUST -> Se debe tener la funcionalidad. En caso de que no exista la solución a construir fallará**.**

S - SHOULD -> Se debería tener la funcionalidad. La funcionalidad es importante pero la solución no fallará si no existe.

C - COULD -> Sería conveniente tener esta funcionalidad. Es en realidad un deseo.

W - WON'T -> No está en los planes tener esta funcionalidad en este momento. Posteriormente puede pasar a alguno de los estados anteriores.

Te presentamos un ejemplo de cómo podes organizar tu lista de historias de usuarios del **product backlog**:

Requisito	Tarea	Quien	Estado (No iniciada / en progreso / completada)	Dia:	1	2	3	4	5	6	7	8	9	10
				Horas pendientes	1120	1088	1076	1048	1040	1032	1020	1008	992	972
Requisito A	Tarea 1	Joao	Completada		16	8								
Requisito A	Tarea 4	Laura	Completada		4									
Requisito A	Tarea 5	Laura	Completada		4									
Requisito A	Tarea 3	Gabri	Completada		8									
Requisito A	Tarea 2	Laura	Completada		16	8	4							
Requisito A	Tarea 6	Gabri	Completada		8	8	8							
Requisito A	Tarea 7	Joao	Completada		16	16	16	8						
Requisito A	Tarea 8	Laura	Completada		8	8	8							
Requisito A	Tarea 9	Laura	Completada		8	8	8	8	8					
Requisito A	Tarea 10	Laura	Completada		8	8	8	8	8	8	4			
Requisito A	Tarea 11	Joao	Completada		16	16	16	16	16	16	8			
Requisito B	Tarea 12	Gabri	Completada		16	16	16	16	16	16	16	16	8	
Requisito B	Tarea 13	Laura	Completada		16	16	16	16	16	16	16	16	8	
Requisito B	Tarea 14	Joao	En progreso		8	8	8	8	8	8	8	8	8	4
Requisito B	Tarea 15	Gabri	En progreso		8	8	8	8	8	8	8	8	8	8
Requisito B	Tarea 16	Laura	En progreso		8	8	8	8	8	8	8	8	8	8
Requisito C	Tarea 17	Joao	No iniciada		4	4	4	4	4	4	4	4	4	4
Requisito C	Tarea 18	Gabri	No iniciada		8	8	8	8	8	8	8	8	8	8
Requisito C	Tarea 19	Laura	No iniciada		16	16	16	16	16	16	16	16	16	16
Requisito C	Tarea 20	Joao	No iniciada		8	8	8	8	8	8	8	8	8	8

Scrum

Scrum es una metodología ágil que proporciona un marco para la gestión de proyectos. Podríamos decir que hoy en día es la metodología ágil más popular, y, de hecho, se ha utilizado para desarrollar software desde principios de la década de los 90.

El conjunto de buenas prácticas de Scrum se aplica esencialmente a la gestión de proyectos. Y para eso esta metodología se basa en **tres pilares fundamentales**, que te pasamos a comentar:

Transparencia

Todos los aspectos del proceso que afectan al resultado son visibles para todos aquellos que administran dicho resultado. Por ejemplo, se utilizan pizarras y otros mecanismos o técnicas colaborativas para mejorar la comunicación.

Inspección

Se debe controlar los diversos aspectos del proceso con la frecuencia suficiente para que se le puedan detectar variaciones o desvíos.

Revisión

El producto debe estar dentro de los límites aceptables. En caso de desviación se procederá a una adaptación del proceso y el material procesado.

El equipo en Scrum

Uno de los aspectos más importantes en cualquier proyecto, y por lo tanto también en los proyectos ágiles, es el establecimiento del equipo. Los roles y responsabilidades deben ser claros y conocidos por todos los integrantes del mismo.

Cada **equipo Scrum tiene tres roles**:

Scrum Master

Es el responsable de asegurar que el equipo Scrum siga las prácticas de Scrum. Sus principales funciones son:

- Ayudar a que el equipo Scrum y la organización adopten Scrum.
- Liderar el equipo Scrum, buscando la mejora en la productividad y calidad de los entregables.
- Ayudar a la autogestión del equipo.
- Gestionar e intentar resolver los impedimentos con los que el equipo se encuentra para cumplir con las tareas del proyecto.

Propietario del Producto (Product Owner)

Es la persona responsable de gestionar las necesidades que serán satisfechas por el proyecto y asegurar el valor del trabajo que el equipo lleva a cabo. Su aportación al equipo se basa en:

- Recolectar las necesidades o historias de usuario.
- Gestionar y ordenar las necesidades representadas por las historias de usuario, las que te describimos en la sección 5.
- Aceptar el software al finalizar cada iteración.
- Maximizar el retorno de inversión del proyecto.

Equipo de desarrollo

El equipo está formado por los desarrolladores, que convertirán las necesidades del Product Owner en un conjunto de nuevas funcionalidades, modificaciones o incrementos del software final. El equipo de desarrollo tiene características especiales, te contamos algunas de las más importantes:

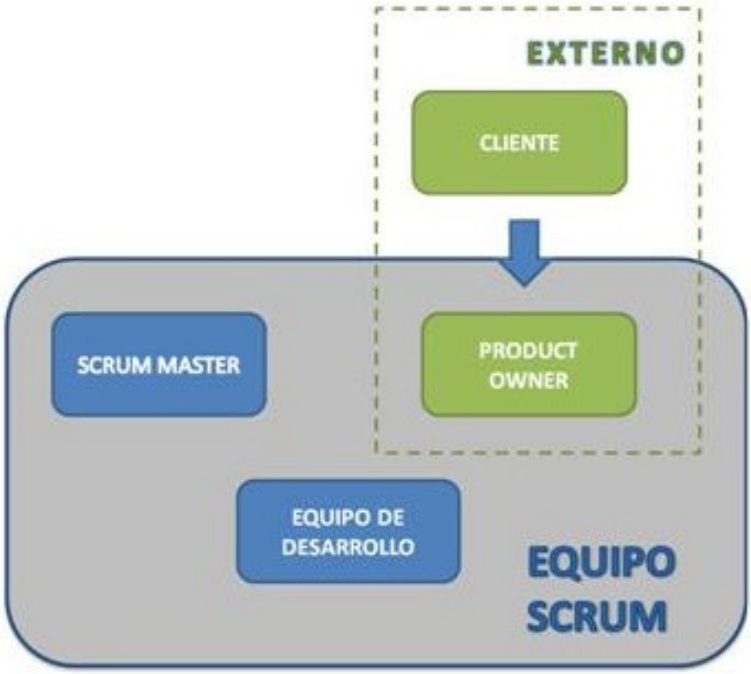
o **Auto-gestionado:** el mismo equipo supervisa su trabajo. En Scrum se potenciarán las reuniones del equipo, aumentando la comunicación. No existe el rol clásico de jefe de proyecto, ya que, como te comentamos antes, el Scrum Master tiene otras responsabilidades.

o **Multifuncional:** no existen compartimientos estancos o especialistas, cada integrante del equipo puede encargarse de tareas de programación, pruebas, despliegue, etc. Asimismo, las personas pueden tener capacidades diferentes o conocimientos más profundos en diferentes áreas. Lo importante es que cualquier integrante del equipo sea capaz de realizar cualquier función.

o **No distribuidos:** es conveniente que el equipo se encuentre en el mismo lugar físico. Esto facilita la comunicación y la autogestión que nace del mismo equipo. Sin embargo, hoy en día con las facilidades en las comunicaciones, se pueden realizar proyectos Scrum con equipos distribuidos gracias a herramientas de trabajo colaborativo.

o **Tamaño óptimo:** un equipo de desarrollo Scrum (sin tener en cuenta al Product Owner y al Scrum Master) estaría compuesto por al menos tres personas. Con menos de tres personas la interacción decae y con ella la productividad del equipo. Como límite superior, con más de nueve personas la interacción hace que la autogestión sea muy difícil de alcanzar.

En la siguiente imagen te mostramos los roles que intervienen en el Equipo Scrum:



El Product Backlog

Ahora te proponemos volver a repasar lo que se conoce como la pila de producto o **Product Backlog,** ahora aplicado a Scrum. En esta metodología es uno de los elementos fundamentales.

El **Product Backlog** consiste en un listado de historias del usuario que se incorporarán al software a partir de incrementos sucesivos. Es decir, sería similar a un listado de requisitos de usuario y representa lo que el cliente espera. Una de las principales diferencias respecto de un proceso tradicional es la evolución continua del **Product Backlog**, buscando aumentar el valor del producto desde el punto de vista del negocio.

A partir del **Product Backlog** se logra tener una única visión durante todo el proyecto. Y, por lo tanto, los fallos en el **Product Backlog** repercutirán profundamente en el proyecto.

Seguramente también recordarás que este listado estará ordenado. Aunque no existe un criterio preestablecido en Scrum para ordenar las historias de usuario, el más aceptado es partir del valor que aporta al negocio la implementación de la historia de usuario.
El responsable de ordenar el Product Backlog es el **Product Owner,** aunque también puede ser ayudado o recibir asesoramiento de otros roles como, por ejemplo, el Scrum Master y el equipo de desarrollo.

Un **Product Backlog** cuenta esencialmente, con cuatro cualidades: debe estar detallado de manera adecuada, estimado, emergente y priorizado. Ahora te contamos con mayor detalle cada una de ellas:

Detallado adecuadamente

El grado de detalle dependerá de la prioridad. Las historias de usuario que tengan una mayor prioridad se describen con más detalle. De esta manera las siguientes funcionalidades a ser implementadas se encuentran descritas correctamente y son viables. Como consecuencia de esto, las necesidades se descubren, se descomponen, y perfeccionan a lo largo de todo el proyecto.

Estimado

Las estimaciones a menudo se expresan en días ideales o en términos abstractos. Saber el tamaño de los elementos del Product Backlog te ayudará a darle prioridad y a planificar los siguientes pasos.

Emergente

Las necesidades se van desarrollando y sus contenidos cambian con frecuencia. Los nuevos elementos se descubren y se agregan a la lista teniendo en cuenta los comentarios de los clientes y usuarios. Así mismo, otros elementos podrán ser modificados o eliminados.

Priorizado

Los elementos del Product Backlog se priorizan. Los elementos más importantes y de mayor prioridad aparecen en la parte superior de la lista. Puede no ser necesario priorizar todos los elementos en un primer momento, sin embargo, sí es conveniente identificar los 15-20 elementos más prioritarios.

El Sprint

Una de las bases de los proyectos ágiles es el desarrollo mediante las iteraciones incrementales.

En **Scrum a cada iteración se le denomina Sprint.**

Scrum recomienda iteraciones cortas, por lo que cada Sprint durará entre 1 y 4 semanas. Y como resultado se creará un producto potencialmente entregable, un prototipo operativo. Las características que van a implementarse en el Sprint provienen del Product Backlog.

El equipo de desarrollo selecciona las historias de usuario que se van a desarrollar en el Sprint conformando la pila de Sprint (Sprint Backlog). La definición de cómo descomponer, analizar o desarrollar este Sprint Backlog queda a criterio del equipo de desarrollo.

Algo importante que tenés que saber es que, aunque todos los Sprints dan como resultado un incremento del software, no todos implican un paso a producción. Es responsabilidad del Product Owner y los clientes decidir el momento en el que los incrementos son puestos en producción. Una posibilidad para realizar esa puesta en producción es con los denominados

"**Sprints de Release**". Estos Sprints contendrán, en general, tareas solamente relacionadas con el despliegue, instalación y puesta en producción del sistema. Es decir, que no existen tareas donde se agreguen nueva funcionalidad.

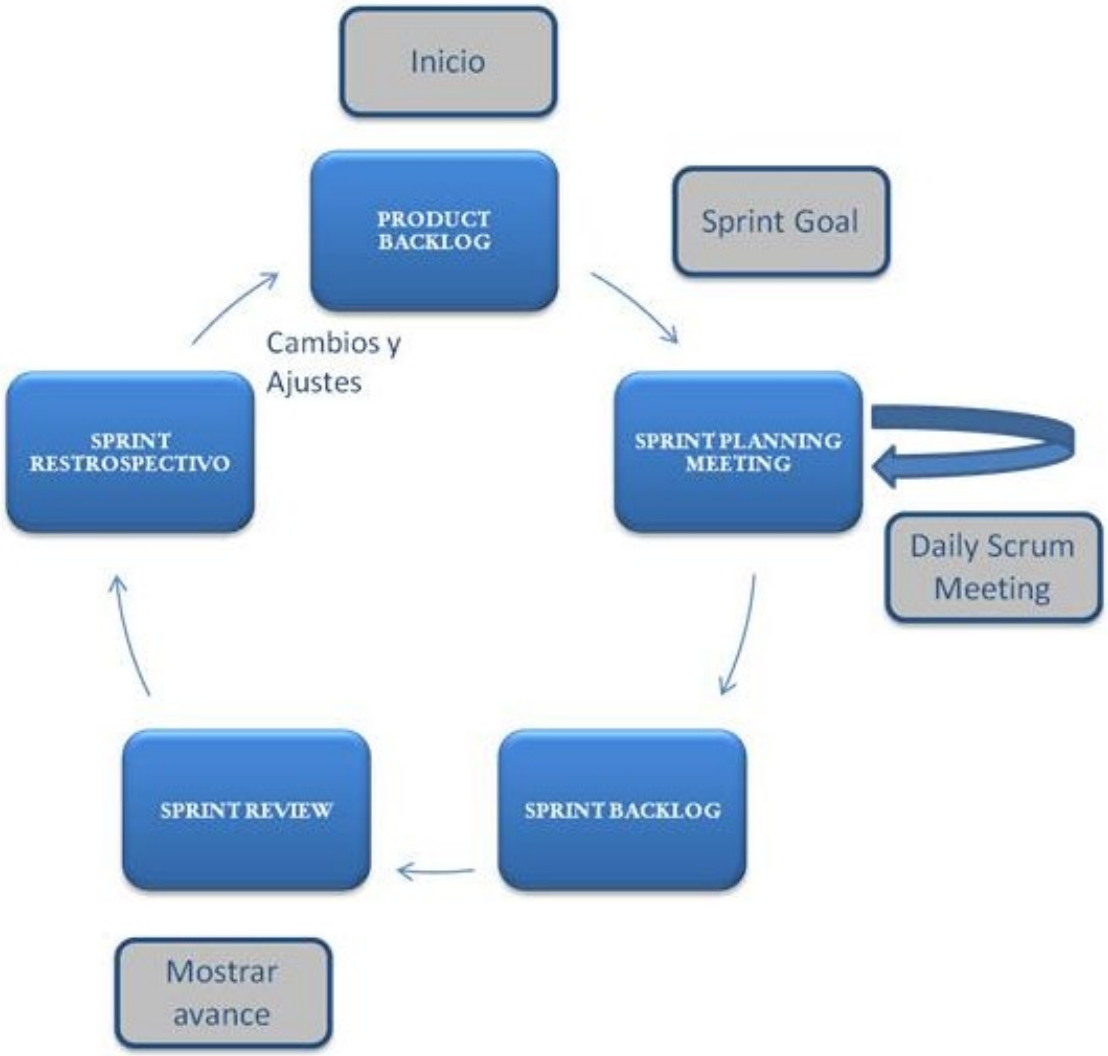
En **Scrum el Sprint Backlog** indica solamente lo que el equipo realizará durante la iteración. El **Product Backlog,** por el contrario, es una lista de características que el Product Owner quiere que se realicen en futuros Sprints.

El **Product Owner** puede visualizar, pero **no** puede modificar el Sprint Backlog. En cambio, puede modificar el ProductBacklog cuantas veces quiera con la única restricción de que los cambios tendrán efecto una vez finalizado el Sprint.

Para mejorar la gestión de las historias de usuario y las tareas de cada Sprint usualmente se utilizan pizarras u otros mecanismos que brinden información inmediata al equipo.

Reuniones

Las **reuniones** son un pilar importante dentro de Scrum. Se realizan a lo largo de todo el Sprint como muestra la siguiente figura. Estas reuniones están representadas en los cuadros con color gris. Se definen diversos tipos de reuniones:



Reunión de planificación del Sprint

Se lleva a cabo al principio de cada Sprint, definiendo en ella que se va a realizar en ese Sprint. Esta reunión da lugar al **Sprint Backlog**. En esta reunión participan todos los roles. El **Product Owner** presenta el conjunto de historias de usuario en el Product Backlog y el equipo de desarrollo selecciona las historias de usuario sobre las que se trabajará. Como resultado de la reunión, el equipo de desarrollo hace una previsión del trabajo que será completada durante el Sprint.

Reunión diaria

Con una duración de no más de 15 minutos, participan el equipo de desarrollo y el Scrum Master. En esta reunión cada miembro del equipo presenta lo que hizo el día anterior, lo que va a hacer hoy y los impedimentos que se han encontrado.

Reunión de revisión del Sprint

Se realiza al final del Sprint. Participan el equipo de desarrollo, el Scrum Master y el Product Owner. Durante la misma se indica qué ha podido completarse y qué no, presentando el trabajo realizado al Product Owner. Por su parte el Product Owner (y demás interesados) verifican el incremento del producto y obtienen información necesaria para actualizar el Product Backlog con nuevas historias de usuario. No debe durar más de 4 horas.

Retrospectiva del Sprint

También al final del Sprint (aunque puede que no se realice al final de todos los Sprints), sirve para que los integrantes del equipo Scrum y el Scrum Master den sus impresiones sobre el Sprint que acaba de terminar. Se utiliza para la mejora del proceso y normalmente se trabaja con dos columnas, con los aspectos positivos y negativos del Sprint. Esta reunión no debería durar más de 4 horas.

Beneficios del Scrum

La **implementación de las metodologías ágiles**, y, por lo tanto, de los principios ágiles, aporta una serie de beneficios como el **aumento de la transparencia** a lo largo de la gestión del proyecto, la **mejora de la comunicación** y la **autogestión del equipo** de desarrollo. Así mismo, existen otras ventajas que se obtienen al utilizar Scrum, entre las cuales te podemos comentar las siguientes:

Entrega periódica de resultados

El **Product Owner** establece sus expectativas indicando el valor que le aporta cada historia de usuario y cuándo espera que esté completado. Por otra parte, comprueba de manera regular si se van cumpliendo sus expectativas.

Entregas parciales

El cliente puede utilizar las primeras funcionalidades de la aplicación que se está construyendo antes de que esté finalizada por completo. Por lo tanto, el cliente puede empezar antes a recuperar su inversión. Por ejemplo, puede utilizar un producto al que sólo le faltan características poco relevantes, puede introducir en el mercado un producto antes que su competidor, puede hacer frente a nuevas peticiones de clientes, etc.

Flexibilidad y adaptación respecto a las necesidades del cliente

De manera regular el Product Owner redirige el proyecto en función de sus nuevas prioridades, de los cambios en el mercado, de los requisitos completados que le permiten entender mejor el producto, de la velocidad real de desarrollo, etc. Al final de cada iteración el Product Owner puede aprovechar la parte de producto completada hasta ese momento para hacer pruebas de concepto con usuarios o consumidores y tomar decisiones en función del resultado obtenido.

Mejores estimaciones

La estimación del esfuerzo y la optimización de tareas es mejor si la realizan las personas que van a desarrollar la historia de usuario, dadas sus diferentes especializaciones, experiencias y puntos de vista. De la misma manera, con iteraciones cortas la precisión de las estimaciones aumenta.

A continuación, te presentamos, a modo de resumen gráfico, el ciclo de la metodología Scrum: