PROCESOS DE SOFTWARE Y NOTAS DEL

CURSO DE

PRÁCTICAS

ÁGILES

Índice de contenidos

Resumen del curso

Al finalizar este curso, serás capaz de:

- Explicar diferentes modelos de proceso para estructurar el desarrollo de software.
- Determine qué modelo de proceso funciona mejor para los diferentes proyectos de desarrollo de software.
- Aplicar los fundamentos de las prácticas de desarrollo de software ágil, como Extreme Programming (XP) y Scrum.

Este curso examina **los procesos de** desarrollo de software y las **prácticas** ágiles. El material comienza con los conceptos clave del proceso y luego pasa a los procesos y prácticas establecidos que se utilizan con éxito en la industria. Por ejemplo, aprenderá el concepto de **procesos iterativos,** seguido de **modelos de proceso** establecidos como el modelo de **Proceso Unificado**. La última mitad de este curso se centrará en la aplicación de prácticas -específicamente, prácticas ágiles- para apoyar los procesos de desarrollo de productos de software.

Módulo 1: Introducción a los procesos

Al finalizar este módulo, serás capaz de:

- Resumir el concepto de *proceso*.
- Definir los términos del proceso: *ciclo de vida*, *fase*, *actividad*, *tarea*, *producto del trabajo*, *recurso* y *función*.
- Diferenciar cómo las tareas se transforman en productos de trabajo.
- Determinar cómo deben utilizarse los métodos y las prácticas.
- Definir e identificar una práctica y una metodología.
- Resumir las actividades clave de la ingeniería del software.
- Identificar las actividades de ingeniería del software en cada fase.
- Definir las tareas y actividades de gestión del proyecto que son continuas a lo largo de todas las fases: gestión de riesgos, realización de estimaciones y asignación de recursos.
- Comprender las diferencias entre la *expresión*, *la reconilación* y la *obtención de requisitos*.

Procesos y prácticas

La importancia de un proceso

Establecer un proceso es un punto de partida clave para todo proyecto de desarrollo de software. Un **proceso** organiza el trabajo de un producto de software en distintas **fases**. En general, el trabajo que se realiza para un proyecto pasará de una fase a la siguiente. Cada fase tiene actividades y tareas específicas que deben realizarse. Al crear fases distintas con pasos y resultados claros, el uso de procesos estructura el desarrollo de software de forma que proporciona claridad a todos los implicados.

La creación de software se parece más a la elaboración de un libro de cocina que a seguir una receta específica. No hay una fórmula prescrita para escribir un libro de cocina; las recetas deben ser recopiladas, organizadas y probadas. Las pruebas de las recetas pueden dar lugar a nuevos refinamientos, experimentos e improvisaciones. El rechazo de algunas recetas puede obligar a volver a la fase de recopilación. Y, si tiene éxito, un libro de cocina puede tener un largo ciclo de vida que implique múltiples ediciones en las que se hagan correcciones y se añadan nuevas recetas.

Del mismo modo, el desarrollo de software requiere una evaluación continua y la toma de decisiones mientras se avanza en cada fase de un proyecto determinado. Esto continúa mucho más allá del proyecto de desarrollo inicial. Un producto de software, al igual que un libro de cocina, puede tener un largo ciclo de vida que incluye múltiples versiones con características mejoradas y defectos resueltos.

Ciclos de vida y subprocesos

Un proceso de ciclo de vida de **software se refiere a** un proceso que cubre todo el espectro del desarrollo de software, desde la concepción inicial del producto hasta su eventual retirada. Un proceso de ciclo de vida es un enfoque global del desarrollo de software, que puede incluir subprocesos para las principales versiones y la evolución del producto. De especial interés para este curso es la creación de un nuevo producto, que requiere un proceso que organice cómo hacer que el producto pase de *ser una idea en la mente del cliente* a *funcionar*

software que experimenta un usuario final. Los procesos del ciclo de vida del software presentados en este curso se centrarán en este problema de desarrollo de software.

Procesos y fases

Como ya se ha dicho, un proceso se divide en varias **fases**. Por ejemplo, las fases de un proceso para crear un producto de software pueden ser:

- Especificación
- Diseño y aplicación
- Verificación y validación

Las fases -o hitos- de un proceso sólo tienen que identificarse de forma lógica y organizarse de manera que las fases contribuyan a la finalización del proceso.

Fases, actividades y tareas

Por ejemplo, considere la fase de *Verificación y Validación* identificada anteriormente; una de las actividades dentro podría ser la *Ejecución de Pruebas.* Las tareas relacionadas con la actividad de ejecución de pruebas podrían incluir:

- Escribir el código del marco de pruebas
- Diseño de pruebas
- Pruebas de redacción
- Ejecución de pruebas

En las tareas es donde se realiza el verdadero trabajo, independientemente del proceso que se utilice. Las tareas son las unidades de trabajo pequeñas y manejables del proyecto. Son los bloques de construcción para completar una actividad, terminar una fase y, en última instancia, completar un proceso.

Última palabra sobre las fases

Adoptar y adaptar un proceso adecuado será la clave para estructurar el trabajo para un proyecto exitoso. El diseño de un proceso puede partir de cero o puede basarse en uno de los muchos **modelos de proceso** que ya se han utilizado con éxito. En este curso se presentarán diversos modelos de procesos.

Todo modelo de proceso tendrá fases; sin embargo, hay grandes diferencias entre los modelos en cuanto a las actividades que constituyen sus fases y la secuenciación de las mismas. Hay tres tipos de modelos de procesos:

- **Modelos de procesos lineales** - fases que ocurren secuencialmente, una tras otra
- **Modelos de procesos iterativos**: fases que se repiten en ciclos
- **Modelos de procesos paralelos**: actividades que ocurren simultáneamente

Tareas y dependencias de tareas

Como se ha introducido anteriormente, las tareas son la unidad de trabajo más básica que se va a gestionar. Las tareas en el desarrollo de software incluyen pequeñas piezas de trabajo, tales como:

- Escribir el código fuente de una función
- Diseñar una función
- Escribir la documentación
- Instalar una biblioteca
- Probar una función

Las tareas suelen tener **dependencias;** por ejemplo, una tarea que debe esperar a que otra se complete se dice que depende de esa otra tarea. Las dependencias impondrán un orden específico para que las tareas se completen. Es importante entender y secuenciar adecuadamente las tareas dependientes.

Por ejemplo, imagine un proceso que se ha dividido en tres fases:

1. Especificación
2. Diseño y aplicación
3. Verificación y validación

Una actividad dentro de la segunda fase podría ser "diseñar interfaces de usuario". Las tareas dentro de esta actividad se ordenarían para reflejar cualquier dependencia identificada. Una de las tareas es diseñar el aspecto de la interfaz de usuario. Otra tarea es diseñar el comportamiento de la interfaz de usuario, que depende de que se haya establecido el aspecto.

Aunque se trata de un pequeño ejemplo, ilustra la necesidad de identificar todas las dependencias de las tareas, para evitar hacerlas fuera de orden y desperdiciar recursos.

Tareas y funciones

En última instancia, son personas concretas las que realizan las tareas. Sin embargo, a la hora de asignar las tareas, es una buena práctica asignarlas a los roles en lugar de a una persona concreta (por ejemplo, Sam) o incluso a un puesto específico (por ejemplo, ingeniero de software). Hay muy buenas razones para enfocar la asignación de tareas de esta manera. En primer lugar, incluso a lo largo de una fase específica, las personas pueden cambiar en una organización, por lo que la asignación de tareas a los roles es más constante a lo largo de la vida del proyecto. En segundo lugar, asignar las tareas a los roles ayudará a evitar cualquier elección personal percibida que pueda ocurrir cuando se asignan las tareas a personas específicas en lugar de a un rol determinado.

Un **rol** es una actividad relacionada con el trabajo que se asigna a una persona. Como se mencionó anteriormente, un miembro del equipo llamado Sam puede tener un título de trabajo "Ingeniero de Software"; sin embargo, uno de sus roles es *programar*. Cualquier equipo de desarrollo tendrá varios miembros que pueden desempeñar muchas funciones, como por ejemplo

- Diseñador gráfico
- Programador

- Probador
- Jefe de proyecto
- Propietario del producto

Cada rol en un equipo describe las habilidades específicas que se necesitan para llevar a cabo cualquier responsabilidad o tarea. Por ejemplo, un programador puede escribir el código de una determinada función y esto podría asignarse a nuestro miembro del equipo de ejemplo, Sam, que es un ingeniero de software. Se asignará un probador para probar esa función, y este papel puede ser desempeñado por una desarrolladora junior llamada Theresa.

*Un rol **realiza** una tarea.*

Tareas y productos de trabajo

Un producto **del trabajo** es un resultado obtenido al completar una **tarea específica**. Los productos de trabajo, por supuesto, incluyen el producto final, pero también otros resultados intermedios como la documentación de diseño, los documentos de requisitos, el código fuente, los casos de prueba y los artefactos de gestión de proyectos, como los informes de estado y los formularios de aprobación.

*Dado que los productos del trabajo son resultados de las tareas, podemos decir que una tarea **produce** un producto del trabajo.*

Productos de trabajo como productos y entradas

Como se ha mencionado, algunas tareas dependen de otras. Así, el producto de trabajo de una tarea anterior también será la *entrada* para una tarea posterior. Por ejemplo, el resultado de una tarea para diseñar una característica se convierte en la entrada de una tarea para escribir el código fuente de esa característica.

*Una tarea no sólo produce productos de trabajo como salida, sino que también **utiliza** productos de trabajo de otra tarea como entrada.*

Tareas y recursos

La realización de cualquier tarea requiere recursos para avanzar o financiarla, como tiempo, personas, tecnología, conocimientos y dinero.

*Una tarea **consume** recursos.*

Prácticas

Las prácticas son estrategias utilizadas para ejecutar los procesos sin problemas. Las prácticas

contribuyen a la eficacia de un proceso de desarrollo. Por ejemplo, un gestor puede seguir prácticas de gestión probadas para: programar y organizar las tareas, minimizar el desperdicio de recursos y hacer un seguimiento del trabajo realizado. Además, la estimación de tareas, la celebración de reuniones diarias, la mejora de la comunicación y la realización de revisiones periódicas del

proyecto son todas prácticas que pueden ayudar a que un proceso funcione bien. Un desarrollador puede seguir las prácticas de desarrollo establecidas para: preparar pruebas eficaces, mantener la simplicidad y realizar revisiones técnicas. Esto garantiza un producto de calidad con pocos defectos.

Las prácticas suelen reunirse en una **metodología**. Dicho de otro modo, una metodología es un conjunto de prácticas. Las metodologías basadas en la filosofía descrita en el Manifiesto Ágil se conocen como metodologías **ágiles**. **Scrum** es un ejemplo de metodología ágil. Es una metodología de desarrollo de software iterativa e incremental para gestionar el desarrollo de productos. Todas las prácticas asociadas a Scrum se adhieren a los principios y la filosofía de Agile.

El Manifiesto Ágil proporciona la filosofía, y una metodología Ágil proporciona un conjunto de prácticas cotidianas que se alinean con esa filosofía.

Prácticas y procesos

Algunas prácticas se prestan a determinadas fases de un proceso. Por ejemplo, dentro de la fase de *especificación de* un proceso de ciclo de vida del software, son útiles las prácticas diseñadas para ayudar a obtener las necesidades y expresar los requisitos.

Otras prácticas son útiles a lo largo de todo el proceso, como las que pertenecen a la gestión del proyecto, como tener reuniones diarias, monitorear la productividad del trabajo del equipo y mantener a todos al tanto de los ítems pendientes que aún deben ser completados. Las prácticas específicas se examinarán más adelante en este curso cuando se presenten las metodologías ágiles.

La adopción de buenas prácticas reduce el desperdicio de recursos. La planificación y la comunicación eficaces evitan que se duplique el trabajo o que se produzca una función que no es necesaria. Las buenas prácticas ayudan a mantener el proyecto dentro de los plazos y el presupuesto.

Los procesos y las prácticas son flexibles y personalizables. Combine varios aspectos de los procesos y las prácticas para crear algo adaptado a sus necesidades de desarrollo. Por último, esté abierto a realizar ajustes que perfeccionen los procesos y prácticas para que se adapten mejor a su proyecto y producto.

Actividades de ingeniería de software

Un proceso se compone de fases. Una fase se compone de actividades. Una actividad se compone de tareas.

En esta lección se analizarán las actividades directamente relacionadas con la ingeniería del software. La comprensión de las actividades de ingeniería de software proporcionará una imagen más completa de cómo se fabrican los productos de software.

Ya hemos identificado que las tareas producen productos de trabajo que pueden convertirse en las entradas de otras tareas. Dado que las tareas relacionadas constituyen una actividad, ésta también tiene entradas y salidas de productos de trabajo (es decir, el

Nota del Instructor:

A medida que se discutan las actividades de ingeniería de software, se agruparán en fases generalizadas para dar ejemplos concretos de cómo las actividades pueden pertenecer a las fases de un proceso.

entrada(s) inicial(es) en las tareas relacionadas y la(s) salida(s) final(es) al completar esas tareas). Además, al igual que las tareas tienen dependencias, las actividades también las tienen. Por ejemplo, la actividad de "creación de pruebas" dará lugar a un producto de trabajo que permitirá iniciar la siguiente actividad de "ejecución de pruebas".

Actividades de especificación

La **fase de especificación** suele ser el punto de partida para desarrollar un producto de software. Estas actividades se centran en los requisitos del producto. En un modelo de proceso lineal, las actividades de especificación pueden llevarse a cabo una vez al principio del proceso; sin embargo, en los modelos iterativos o paralelos, las actividades de especificación pueden revisarse varias veces para captar las necesidades actualizadas del cliente y las mejoras sugeridas para el producto. Un gestor de productos de software desempeña un papel importante en las actividades de especificación.

Las actividades de especificación suelen incluir:

- Identificar ideas o necesidades
- Obtención de requisitos
- Expresión de las necesidades
- Priorizar los requisitos
- Analizar los requisitos
- Gestión de las necesidades
- Formulación de posibles enfoques

No hay que subestimar la importancia de *identificar ideas o necesidades*. Aunque parezca evidente, entender o explorar a fondo la idea o necesidad que impulsa el desarrollo ayudará a crear un gran producto de software.

Una vez entendida la idea central del producto, los siguientes pasos son definir los requisitos.

1. Obtener los requisitos (indagar sobre ellos)
2. Expresar los requisitos (escribirlos de forma estructurada)
3. Priorizar los requisitos (clasificarlos por importancia)
4. Analizar los requisitos (comprobar su claridad, coherencia e integridad)
5. Gestionar los requisitos (organizarlos, reutilizarlos, referenciarlos)

Las actividades de ingeniería de software hacen hincapié en tener un buen conjunto inicial de requisitos para guiar el trabajo de desarrollo. El curso **Necesidades del cliente y requisitos del software** dentro de esta especialización está dedicado a las actividades y prácticas para formar requisitos efectivos.

La actividad final, la *formulación de posibles enfoques*, permite al equipo de producto esbozar una estrategia, considerar alternativas y perfeccionar sus próximos pasos.

Actividades de diseño y ejecución

Se trata de actividades realizadas por el equipo de desarrollo para diseñar e implementar el producto de software. Los gestores deben hacer un seguimiento de los productos de trabajo resultantes de estas actividades.

Las actividades de diseño y ejecución pueden incluir:

- Diseño de la arquitectura

- Diseño de la base de datos
- Diseño de interfaces
- Creación de código ejecutable
- Integrar la funcionalidad
- Documentar

Todas estas actividades tienen lugar antes de que comience la programación. Tener un diseño para la arquitectura del software, las bases de datos relacionadas y las interfaces mejorará la capacidad de los desarrolladores para trabajar juntos hacia un objetivo común. Una vez que comience la programación, los programadores deben integrar o combinar su código a intervalos regulares para garantizar una funcionalidad compatible. La documentación interna ayudará a los nuevos desarrolladores cuando se incorporen al equipo o permitirá a los miembros actuales del equipo trabajar en el desarrollo de aspectos desconocidos del producto.

Actividades de verificación y validación

La verificación y validación constantes del producto garantizarán que su desarrollo vaya por buen camino. Las actividades de verificación y validación incluyen:

- Desarrollo de procedimientos de prueba
- Creación de pruebas
- Ejecución de pruebas
- Informar de los resultados de la evaluación
- Realización de revisiones y auditorías
- Demostrar a los clientes
- Realización de retrospectivas

La verificación suele consistir en pruebas internas y actividades de revisión técnica que garantizan que el producto hace lo que debe hacer, de acuerdo con el diseño y los requisitos documentados.

Las actividades de validación, como la *demostración a los clientes*, ayudan a comprobar que el producto satisface las necesidades del cliente y/o de los usuarios finales. Con los modelos de procesos iterativos o paralelos, la información obtenida en esta fase puede incorporarse a nuevas especificaciones para mejorar el producto en el siguiente ciclo de desarrollo.

Tanto el producto como el proceso pueden mejorarse. La *realización de retrospectivas* es una actividad que permite identificar las áreas que deben mejorarse la próxima vez. Los ejercicios retrospectivos con el equipo de desarrollo generarán ideas para mejorar el proceso y el producto.

Actividades de gestión de proyectos

Aunque las fases del ejemplo representan un flujo natural a través del desarrollo de software, también hay actividades de gestión de proyectos en curso que generalmente ocurren en paralelo a las actividades discutidas anteriormente.

Las actividades de gestión de proyectos suelen incluir:

- Creación de un proceso

- Establecer normas
- Gestión de riesgos
- Realización de estimaciones
- Asignación de recursos
- Realización de mediciones
- Mejorar el proceso

La creación de un proceso es una parte importante de la gestión de un proyecto porque establece las fases iniciales del desarrollo. Un proceso proporciona una estructura de actividades y trabajos previstos, que ayudan a guiar a un equipo a través de un proyecto de desarrollo de productos.

Además de un proceso, es necesario *establecer normas* para garantizar que las actividades del equipo se realicen de forma coherente. El equipo necesita un conjunto claro de expectativas sobre aspectos del desarrollo como las convenciones de codificación, los niveles de documentación, las estrategias de prueba y cuándo debe considerarse que el trabajo está *terminado*. Un equipo de desarrollo puede establecer la norma de que el trabajo para una característica se considera *hecho*, por ejemplo, cuando el código fuente implementado sigue una convención especificada, cuando la documentación se ha proporcionado a los desarrolladores y usuarios finales, o cuando las características han sido probadas.

Otra actividad de gestión de proyectos es la *gestión de riesgos*. Esto requiere un análisis constante y la mitigación de los riesgos potenciales (por ejemplo, riesgos empresariales, técnicos, de gestión, de programación y de seguridad). Para gestionar los riesgos se pueden utilizar los registros históricos del proyecto, las estimaciones del proyecto, el diseño detallado del software y el proceso del ciclo de vida. Cualquier parte del proyecto que pueda considerarse vulnerable al fracaso sería un producto de trabajo de entrada para la gestión de riesgos. El producto de trabajo de salida es un **plan de riesgos** que describe los riesgos potenciales y las estrategias de mitigación. La revisión continua de los riesgos permitirá identificar y mitigar los nuevos riesgos que puedan surgir a lo largo del proceso de desarrollo.

Además, la gestión eficaz de los proyectos se basa en la *realización de estimaciones*. La estimación es una actividad crítica en el desarrollo de software, ya que los plazos de desarrollo se basan en estimaciones del tiempo necesario para completar las tareas. Cuando las estimaciones son demasiado imprecisas, el calendario puede verse gravemente comprometido. Las tareas de la actividad de estimación incluyen la recopilación de datos para las estimaciones, la realización de cálculos de estimación y la evaluación de las estimaciones. Por ejemplo, realizar una estimación de la duración de una tarea determinada es en sí misma una tarea con un producto de trabajo de entrada de mediciones históricas de la duración de la tarea. El producto de trabajo de salida es la propia duración estimada, que puede utilizarse en las tareas de programación posteriores.

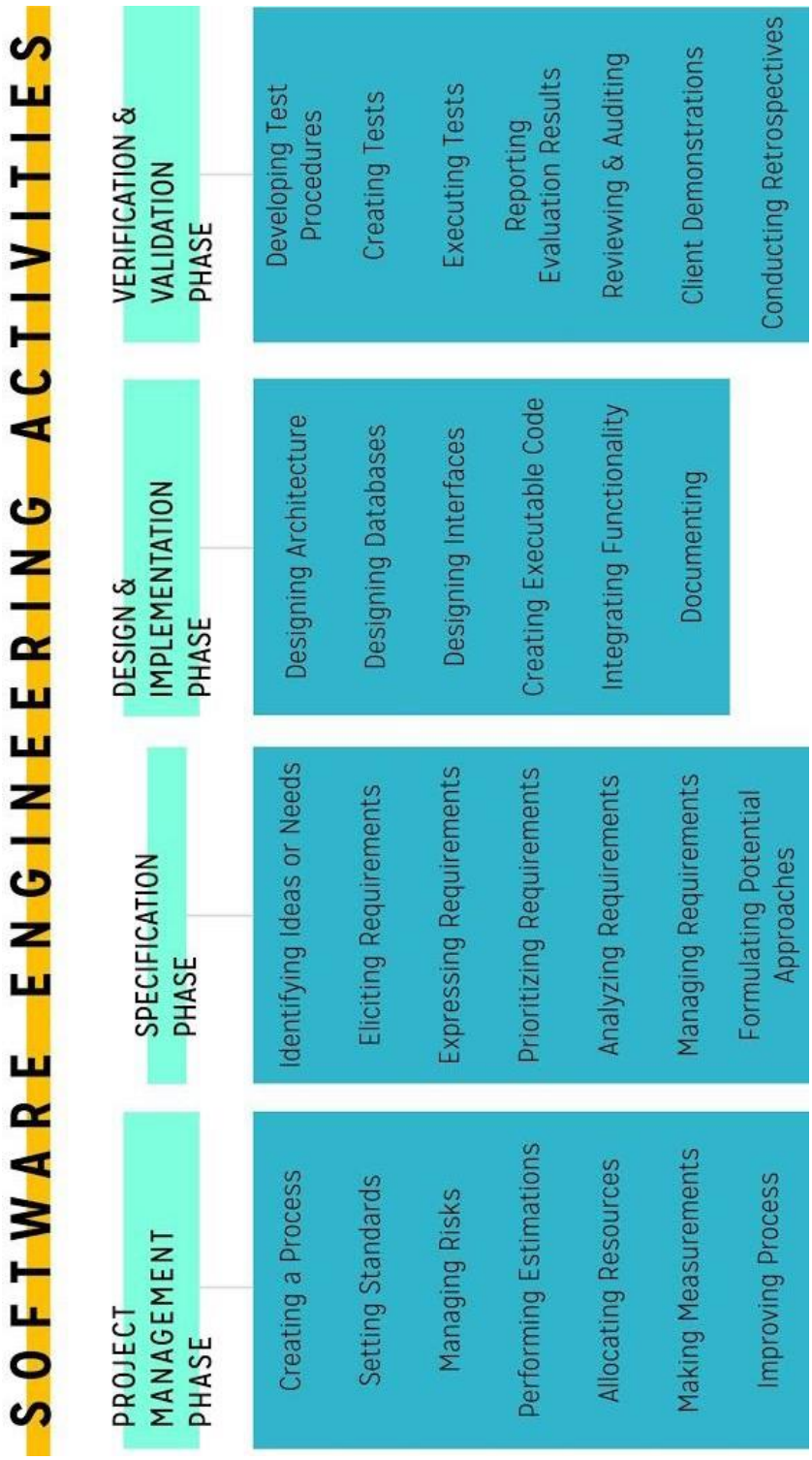
La actividad de asignación de *recursos* garantiza que las personas, el tiempo, el dinero y el equipo se desplieguen para realizar una tarea requerida. Tanto la asignación de tareas como la creación de un plan de trabajo son tareas de la actividad de asignación de recursos.

Realizar mediciones es otra actividad importante de la gestión de proyectos. Esto implica tareas como la definición de métricas utilizables, el cálculo de datos y el análisis de los datos de las métricas. **Las métricas** se utilizan para seguir y evaluar la calidad del producto y/o del proceso.

La mejora del proceso es una parte fundamental de la gestión eficaz del proyecto. Es importante reconocer que, al igual que el producto, también se puede mejorar el proceso de desarrollo. Es tan importante probar y revisar el proceso como el producto. Las mediciones son útiles para los cálculos que

determinan cómo se puede mejorar el proceso.

Resumen de las actividades de ingeniería de software



La descripción de estas actividades de ingeniería de software proporciona una idea de cómo contribuyen al desarrollo de un producto de software o a la gestión de un proyecto de software.

Módulo 2: Modelos de procesos

Al finalizar este módulo, serás capaz de:

Resumen del módulo

- Explicar la necesidad de considerar varios modelos de proceso.
- Describir completamente un modelo lineal, incluyendo sus puntos fuertes y débiles, y sus fases en el orden correcto. Esto incluye:
  - Comprender las similitudes y diferencias de los modelos lineales: Waterfall, V-Model y Sawtooth.
- Describir completamente un modelo iterativo incluyendo sus puntos fuertes y débiles. Esto incluye:
  - Comprender cómo se relacionan las fases con la iteración en el modelo iterativo en espiral.
- Describir completamente un modelo paralelo, incluyendo sus puntos fuertes y débiles. Esto incluye:
  - Describir las fases específicas asociadas a los modelos de Proceso Unificado.
  - Describir cuándo los modelos de Proceso Unificado pueden ser excesivos para pequeños proyectos y cuándo son más apropiados para proyectos más grandes.
- Enumere y describa los cinco tipos de prototipos: ilustrativo, exploratorio, desechable, incremental y evolutivo.
- Diferenciar entre los prototipos, incluyendo:
  - Evolutivo e incremental,
  - Ilustrativo y exploratorio, y
  - Ilustrativo y desechable.

Los modelos de procesos para el desarrollo de software han evolucionado a la par que los avances en la potencia informática. Aunque puede ser tentador adoptar simplemente las últimas innovaciones en modelos de procesos de desarrollo de software, no todos los modelos de procesos se ajustan a todas las necesidades del proyecto. Merece la pena adquirir una perspectiva de las fortalezas y debilidades relativas de cada modelo, con el fin de asegurarse de que se está musando la mejor herramienta para el trabajo. Un proyecto puede descubrir que algunos modelos de proceso son más complejos o exhaustivos de lo necesario. Por ejemplo, el desarrollo de un sencillo blog web para una organización local sin ánimo de lucro puede realizarse fácilmente con un modelo de procesos sencillo; de hecho, seguir un modelo de procesos más avanzado puede suponer más gastos de los que la organización puede permitirse.

Modelos lineales

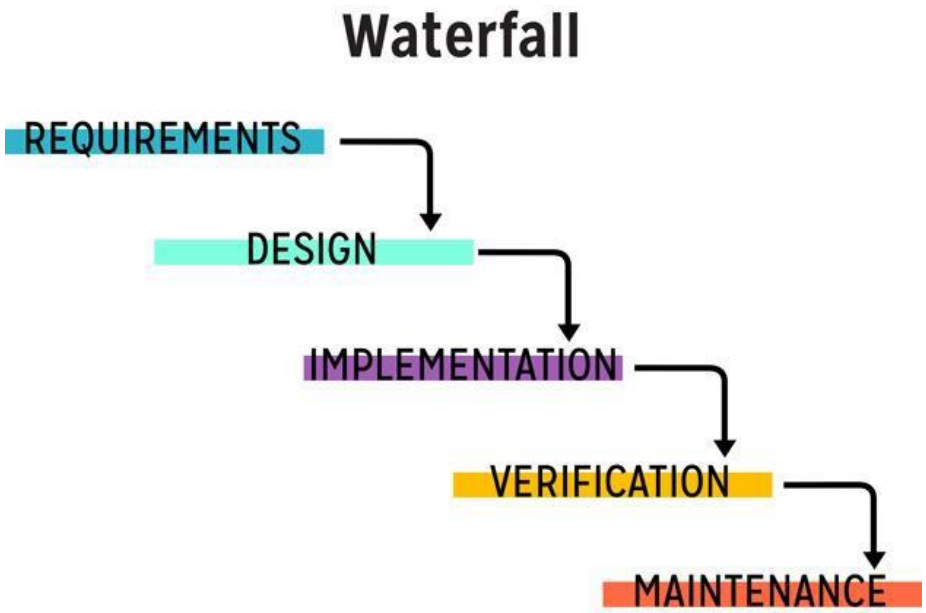
En esta lección, presentaremos tres de los primeros modelos de proceso lineal: El modelo en cascada, el modelo en V y el modelo en diente de sierra. Discutiremos los puntos fuertes y débiles de cada modo, y luego pasaremos a modelos más elaborados.

Los modelos de proceso lineal siguen una serie de fases, una por una, sin retorno a las fases anteriores. El producto se especifica, se diseña, se desarrolla y se lanza al mercado en ese orden, sin posibilidad de volver a las fases anteriores. Los procesos lineales son los más adecuados para proyectos en los que se espera muy poca retroalimentación o refinamiento. Los proyectos exitosos que siguen un modelo de proceso lineal se entienden bien desde el principio, con poco o ningún margen de cambio.

En los primeros tiempos de la informática, el proceso de desarrollo de software no favorecía la experimentación rápida. Los ordenadores no eran tan prolíficos como ahora, y desarrollar para ellos era mucho más costoso. El código informático tenía que escribirse (o perforarse en tarjetas) con una meticulosa atención al detalle, y el tiempo que se tardaba en compilar el código era a menudo importante y caro. Por ello, se hacía más hincapié en el diseño detallado y por adelantado. Los modelos de procesos lineales encajan en esta primera forma de pensar.

Modelo de cascada

El modelo de **cascada** es un modelo de proceso lineal, en el que cada fase produce un producto de trabajo aprobado que es utilizado por la fase siguiente. El producto de trabajo fluye hacia la siguiente fase, y esto continúa hasta el final del proceso, como una cascada. Hay variantes modificadas del modelo de cascada que permiten flujos hacia fases anteriores, pero el flujo dominante sigue siendo hacia adelante y lineal.



Según el proceso de cascada mostrado, al final de la fase de **requisitos**, hay un producto de trabajo de salida: un **documento de requisitos**. Este documento se aprueba formalmente y pasa a la siguiente fase: **el diseño**. Este paso de trabajo continúa de fase en fase hasta que el producto de software es el resultado final.

Ventajas e inconvenientes

Puede ser primitivo, pero el modelo de cascada tiene ventajas, incluso en los proyectos de desarrollo actuales: Este proceso es fácil de entender; define claramente los resultados y los hitos; subraya la importancia del análisis antes del diseño, y del diseño antes de la implementación; y puede adoptarse en equipos que utilizan procesos más sofisticados, para partes del producto bien especificadas que pueden subcontratarse.

El modelo Waterfall tiene un gran defecto: no es muy adaptable al cambio. Es decir, no se adapta a los

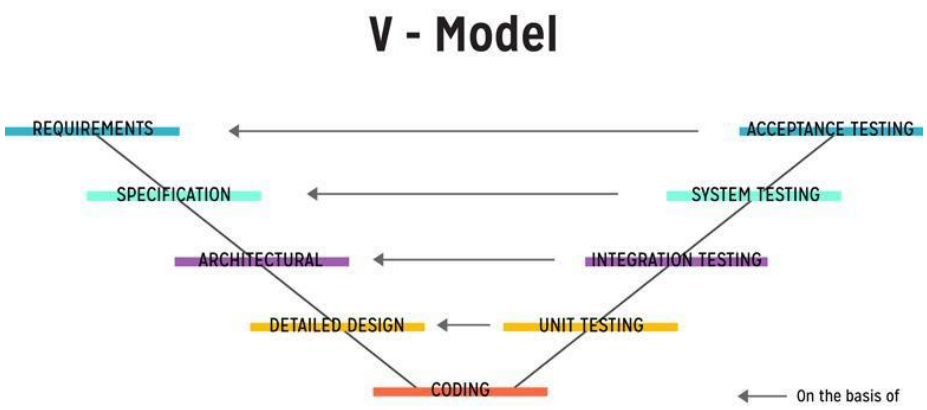
principios ágiles. El desarrollo de software moderno suele ser muy dinámico; la flexibilidad con respecto a

Se valoran las necesidades cambiantes y las tecnologías emergentes. El modelo Waterfall se centra en conocer todos los requisitos por adelantado. Simplemente, no está diseñado para abordar los cambios a mitad de camino con facilidad o sin grandes costes, porque requeriría volver a visitar las fases anteriores. Además, las pruebas se realizan en una fase tardía del proceso, y los defectos detectados tarde suelen ser más difíciles y costosos de solucionar.

En el modelo de cascada, el cliente no ve el producto hasta casi el final del desarrollo, por lo que el producto desarrollado puede no coincidir con lo que el cliente había previsto.

V-Modelo

El **modelo V** es otro modelo de proceso lineal. Se creó en respuesta a lo que se identificó como un problema del modelo en cascada; es decir, añade un enfoque más completo en la prueba del producto. El rasgo distintivo -y el nombre- del modelo V son las dos ramas que convierten una estructura lineal en forma de "V". Al igual que el modelo en cascada, el modelo en V comienza con el análisis y la identificación de los requisitos, que alimentan la información del producto en las fases de diseño e implementación. Las fases de análisis y diseño constituyen la rama izquierda de la V. La rama derecha representa las actividades de integración y prueba. A medida que el trabajo de pruebas avanza por la rama derecha, algún nivel del producto (en el lado derecho) se verifica activamente con el diseño o los requisitos correspondientes (en el lado izquierdo).



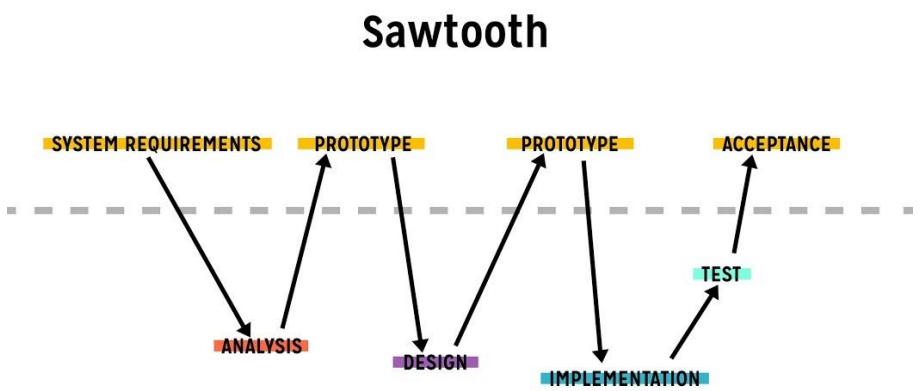
Ventajas e inconvenientes

El modelo en V tiene las mismas ventajas e inconvenientes que el modelo en cascada, ya que es fácil de entender, pero no se adapta bien al cambio. Sin embargo, el hecho de permitir al equipo de desarrollo verificar el producto en múltiples niveles en fases explícitas es una mejora respecto al modelo en cascada.

Tanto en el modelo en cascada como en el modelo en V, el cliente no ve el producto acabado hasta que todo está prácticamente terminado. La siguiente evolución de los procesos lineales es idear una forma de involucrar al cliente en el proceso de desarrollo.

Modelo de dientes de sierra

El modelo de diente de sierra también es un modelo de proceso lineal, pero a diferencia de los modelos en cascada y en V, el cliente participa para revisar los prototipos intermedios del producto durante el proceso. A medida que el proceso avanza linealmente, hay fases que implican al cliente en la sección superior del diagrama y fases que implican sólo al equipo de desarrollo en la sección inferior, lo que crea un patrón de "diente de sierra" irregular.



Ventajas e inconvenientes

La participación del cliente en las primeras fases del proceso aumenta la probabilidad de que el producto satisfaga sus necesidades; ésta es una ventaja indudable del modelo Sawtooth en comparación con el modelo Waterfall y el modelo V. Sin embargo, al igual que ellos, el modelo Sawtooth sigue siendo un proceso lineal, por lo que tiene un límite para incorporar algo más que cambios incrementales.

Últimas palabras sobre los modelos de procesos lineales

Lo principal que tienen en común los modelos de procesos lineales es su organización secuencial de fases. El desarrollo se produce de forma sencilla y directa. El hecho de no poder revisar las primeras fases, como el análisis o el diseño, da lugar a la inflexibilidad de estos modelos. Las decisiones tomadas en las primeras fases del proceso condicionan el producto final.

Los primeros modelos de procesos lineales se inscriben en una visión de *fabricación de* un producto informático. En esencia, esto significa que se mecaniza y ensambla de acuerdo con ciertos requisitos. Una vez fabricado, el producto sólo requiere un mantenimiento menor, como un electrodoméstico. El énfasis recae en conseguir que todos los requisitos se especifiquen "correctamente" al principio del proceso y no cambiarlos después. En realidad, el desarrollo de un producto de software es más bien una tarea creativa, que requiere experimentación y reelaboración constante.

Los modelos de procesos lineales son una herencia de una época en la que la potencia de cálculo era relativamente cara en comparación con el trabajo humano. Así, la programación era eficiente para los ordenadores, pero no necesariamente para las personas. El tiempo transcurrido entre la escritura de una línea de código y la visualización de su resultado podía ser de horas. Probar pequeños experimentos de programación para comprobar rápidamente las ideas no era práctico. Los modelos de procesos

lineales se ajustan a un modo de pensar: hacer las cosas bien a la primera y evitar la repetición del trabajo.

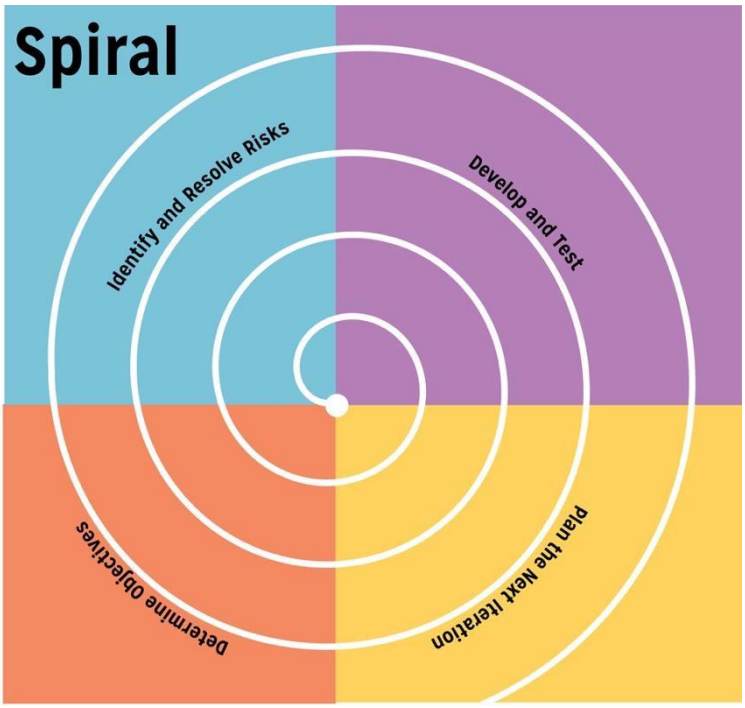
Como hemos dicho antes, estos modelos lineales pueden seguir siendo útiles para determinadas situaciones. Por ejemplo, los procesos lineales son estupendos para desarrollar productos sencillos y bien definidos. De hecho, los modelos de procesos más avanzados siguen teniendo ciertas fases que suceden linealmente.

Modelos iterativos

Los modelos de procesos iterativos se diferencian de los modelos de procesos lineales en que están diseñados para repetir etapas del proceso. Es decir, tienen una estructura iterativa o cíclica.

La ventaja de los procesos iterativos es la capacidad de hacer un bucle y volver a visitar las fases anteriores (y sus actividades) como parte del proceso. Cada "bucle de vuelta" es una iteración, de ahí el nombre de "proceso iterativo". Las iteraciones permiten incorporar los comentarios de los clientes dentro del proceso como norma, no como excepción. Los modelos de procesos iterativos se adaptan fácilmente a las prácticas ágiles, aunque también incorporan partes secuenciales que recuerdan a los modelos de procesos lineales.

Modelo en espiral



El modelo **en espiral** fue introducido por Barry Boehm en 1986. El modelo esbozaba un proceso básico en el que los equipos de desarrollo podían diseñar e implantar con éxito un sistema de software revisando las fases del proceso después de haberlas completado previamente.

Una versión simplificada del modelo en espiral tiene cuatro fases, que tienen metas asociadas: determinar los objetivos, identificar y resolver los riesgos, desarrollar y probar, y planificar la siguiente iteración. Tomadas en orden, las cuatro fases representan una iteración completa. En cada iteración posterior se revisa la secuencia de las cuatro fases, y cada iteración da lugar a un prototipo de producto. Esto permite al equipo de desarrollo revisar su producto con el cliente para recabar opiniones y mejorar el producto.

Las primeras iteraciones dan lugar a ideas y conceptos de productos, mientras que las últimas dan lugar a prototipos de software que funcionan. El modelo en espiral se suele representar con las cuatro fases en forma de cuadrantes y una espiral que crece hacia fuera para indicar la progresión a través de las fases.

Inconvenientes del modelo en espiral

La estimación del trabajo puede ser más difícil, dependiendo de la duración del ciclo de iteración en el modelo espiral. Cuanto más largo sea el ciclo de iteración, más lejos habrá que planificar y estimar; las iteraciones largas pueden introducir más incertidumbre en las estimaciones. Es más fácil estimar el esfuerzo en cosas pequeñas y planificar para dos semanas adelante que estimar el esfuerzo necesario en muchas cosas grandes para varias semanas adelante.

Además, el modelo en espiral requiere muchos conocimientos analíticos para evaluar los riesgos. Estas extensas tareas de evaluación de riesgos pueden consumir una gran cantidad de recursos para ser realizadas correctamente. No todas las organizaciones dispondrán de los años de experiencia, los datos y los conocimientos técnicos necesarios para la estimación y la evaluación de los riesgos.

Última palabra sobre el modelo en espiral

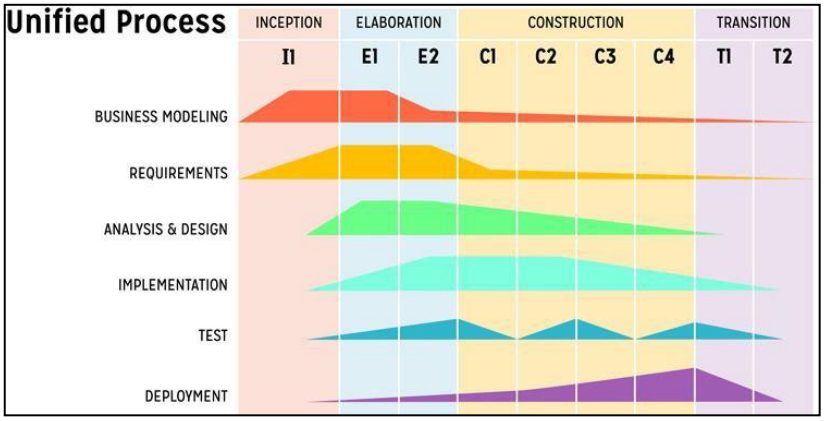
Esta descripción simplificada del modelo en espiral introduce la idea de un proceso iterativo. El trabajo se realiza en iteraciones que permiten revisar el producto en determinados intervalos específicos. La capacidad de aprovechar un proceso repetitivo para realizar nuevos perfeccionamientos de un producto es una clara ventaja sobre los anteriores modelos lineales.

Modelos paralelos

El modelo en espiral es un verdadero proceso iterativo para el desarrollo de software, lo que significa que el producto se construye en una serie repetida de fases y los refinamientos del producto se introducen en fases específicas dentro del ciclo. Los **procesos paralelos** utilizan un estilo de iteración similar -los productos se construyen en iteraciones-, pero permiten una mayor concurrencia de actividades y tareas.

Modelo de Proceso Unificado

El modelo **del Proceso Unificado** es un modelo iterativo de desarrollo de software. Su estructura básica tiene fases secuenciales dentro de un ciclo repetitivo. Dentro de la mayoría de las fases del modelo de Proceso Unificado, el trabajo se realiza en pequeñas iteraciones hasta que la fase se considera completa. Normalmente, las fases se consideran completas cuando se alcanza un **hito,** un punto específico e identificable en un



proyecto.

Aunque la estructura general del Proceso Unificado es iterativa, el modelo permite que las tareas realizadas en una fase también se realicen en otra. Así, una tarea o actividad relacionada con los requisitos puede llevarse a cabo en todas las fases en lugar de en una sola. Esto también significa que, por ejemplo, una tarea de requisitos, una tarea de diseño de arquitectura y una tarea de desarrollo de pruebas pueden ocurrir en **paralelo** en la misma fase. Por el contrario, en el modelo de cascada, estas tareas se organizarían en fases específicas y separadas, sin paralelismo para esas tareas.

Fases dentro del Modelo de Proceso Unificado

El modelo del Proceso Unificado consta de cuatro fases, y cada una de ellas tiene un objetivo distinto.

1. Fase inicial
2. Fase de elaboración
3. Fase de construcción
4. Fase de transición

Fase inicial

Esta primera fase debe ser lo suficientemente corta como para establecer un caso de negocio lo suficientemente fuerte y una razón financiera para continuar con las siguientes fases y hacer el producto.

Para ello, la *fase inicial* suele requerir la creación de **casos de uso básicos**, que describen las principales interacciones del usuario con el producto. También se definen el **alcance del proyecto** y los posibles **riesgos** del mismo. La fase de inicio es la única fase del modelo de Proceso Unificado que no se desarrolla en iteraciones. Si la fase de inicio es larga, puede sugerir que se ha perdido tiempo en el análisis de los requisitos.

La finalización de la fase inicial está marcada por un hito **objetivo del ciclo de vida**. El producto del trabajo para lograr este hito es una descripción razonable de la viabilidad del producto y la preparación para pasar a la siguiente fase del proceso.

Fase de elaboración

El Proceso Unificado se centra en la importancia de perfeccionar la arquitectura del producto a lo largo del tiempo. La arquitectura es un conjunto de diseños sobre los que se construye el producto de software. Así, el objetivo de la *fase de elaboración* es crear modelos de diseño y prototipos del producto, así como abordar los riesgos. Esta fase es también la primera de las que incorporan pequeñas iteraciones dentro de la fase.

Además de definir la arquitectura del sistema, los desarrolladores perfeccionan los requisitos concebidos anteriormente en la fase inicial. También desarrollan los requisitos clave y la documentación de la arquitectura, como **los diagramas de casos de uso** y los **diagramas de clases** de alto nivel. Esta fase sienta las bases sobre las que se construirá el desarrollo real.

La construcción de un prototipo requerirá probablemente varias iteraciones antes de que los requisitos y los modelos de arquitectura se consideren lo suficientemente completos como para seguir adelante.

Al final de la fase de elaboración, los desarrolladores entregan un plan para el desarrollo en la siguiente fase. Este plan se basa básicamente en lo desarrollado durante la fase de inicio; integra todo lo aprendido durante la fase de elaboración para que la construcción pueda llevarse a cabo de forma eficaz.

Fase de construcción

Al igual que la fase anterior, la *fase de construcción* tiene iteraciones y se centra en construir sobre el trabajo anterior. Aquí es donde el producto de software comienza a tomar forma.

Dado que el modelo del Proceso Unificado es un proceso paralelo, cuando se inicia la fase de construcción, el trabajo de la fase de elaboración seguirá adelante. La única diferencia es que el énfasis del trabajo puede cambiar.

Las actividades de prueba y programación pueden haber sido importantes en la fase de elaboración (para los estudios de viabilidad técnica o para establecer el entorno de desarrollo), pero adquieren aún más importancia en la fase de construcción. Del mismo modo, la evaluación de los riesgos es importante en la fase de inicio, pero es menos importante en la fase de construcción.

En la fase de construcción, se desarrollan **casos de uso completos** para impulsar el desarrollo del producto. Estas mejoras sobre las versiones básicas desarrolladas en la fase de inicio consisten en un conjunto de posibles interacciones secuenciales entre usuarios y sistemas. Ayudan a identificar, aclarar y organizar las funcionalidades de un producto. Estos casos de uso son más sólidos que los iniciados en la fase inicial, y ofrecen una visión más específica de cómo el producto debe apoyar las tareas del usuario final.

El producto se construye de forma iterativa a lo largo de la fase de construcción hasta que está listo para ser lanzado. En ese momento, el equipo de desarrollo comienza la transición del producto al cliente y/o a los usuarios finales.

Fase de transición

Durante esta fase, el equipo de desarrollo recibe los comentarios de los usuarios. La *fase de transición* revela hasta qué punto el diseño del producto se ajusta a las necesidades de los usuarios. Al recoger las opiniones de los usuarios, el equipo de desarrollo puede introducir mejoras en el producto, como la corrección de errores y el desarrollo de futuras actualizaciones.

Una vez completada la *fase de transición*, es posible volver a recorrer las fases del Proceso Unificado. Por ejemplo, puede ser necesario crear nuevas versiones del producto o incorporar los comentarios de los usuarios para influir en los planes de desarrollo posteriores.

Última palabra sobre el Modelo de Proceso Unificado

El modelo de Proceso Unificado es un ejemplo de proceso paralelo que presenta iteraciones. Actividades como los requisitos, el diseño y la implementación pueden tener lugar al mismo tiempo, y todas ellas conducen al producto a una etapa de "lanzamiento". Sin embargo, una vez que el producto es lanzado, el proceso puede comenzar otro ciclo para perfeccionar y mejorar el producto.

Las iteraciones también se producen dentro de las fases; por ejemplo, la fase de elaboración puede dar

lugar a varias iteraciones de diseño antes de que el producto del trabajo esté listo para la fase de construcción.
El Proceso Unificado se adapta bien a los grandes proyectos en los que las iteraciones pueden permitir que el producto crezca de forma natural más allá del inicio original del producto.

Es posible que observe un patrón de evolución de los procesos cada vez más avanzados, pero esto no significa que los procesos más sencillos estén obsoletos. Aunque los procesos lineales e iterativos anteriores pueden ser menos sofisticados que los procesos paralelos modernos, los procesos más antiguos se siguen utilizando con frecuencia en situaciones más sencillas.

Prototipos

El desarrollo de productos de software a través de una serie de prototipos intermedios era un tema en los modelos de Proceso Espiral y Proceso Unificado. Exploremos la creación de prototipos con más detalle.

Hay cinco tipos de prototipos:

1. Ilustrativo
2. Exploración
3. Desechable
4. Incremento de
5. Evolución

Prototipo ilustrativo

Este es el más básico de todos los prototipos. Pueden ser dibujos en una servilleta, un conjunto de diapositivas o incluso un par de fichas con componentes dibujados. La esencia de un prototipo *ilustrativo* es plasmar una idea básica.

Los prototipos ilustrativos pueden dar al equipo de desarrollo una idea común en la que basar su trabajo, o ayudarles a conseguir el "aspecto" correcto de un sistema sin invertir mucho tiempo o dinero en el desarrollo de un producto. Utilice prototipos ilustrativos para descartar ideas problemáticas o como guía para el desarrollo posterior.

¿LO SABÍAS?

Algunos prototipos llegan a "fingir" su funcionalidad haciendo que un humano controle algunas de las funciones del sistema entre bastidores.

La creación de prototipos ilustrativos puede implicar la elaboración de storyboards o wireframes. Todos ellos pretenden mostrar cómo funcionará un sistema o ilustrar un concepto utilizando únicamente diagramas e imágenes.

Prototipo exploratorio

La creación de prototipos exploratorios se centra en algo más que el aspecto del producto. Al construir un código de trabajo, el equipo de desarrollo puede explorar lo que es factible, con la esperanza de desechar el trabajo después de aprender del prototipo. Con más tiempo, puede surgir una visión más completa de cómo será el producto, así como una comprensión del esfuerzo necesario para construirlo.

La creación de prototipos exploratorios es costosa en términos de consumo de recursos, pero se utiliza

porque es mejor y menos costosa que descubrir más tarde en el proceso que una solución de producto no puede funcionar. La motivación habitual de los prototipos exploratorios es que los desarrolladores de productos quieren estudiar la viabilidad de una idea de producto. A diferencia del prototipo ilustrativo, que sólo se preocupa por el aspecto del producto,

La creación de prototipos exploratorios tiene que ver con la posibilidad de desarrollar el producto o la utilidad del mismo, antes de dedicar más esfuerzos a la idea.

Prototipo desechable

La primera versión de casi cualquier producto está destinada a tener varios problemas. Entonces, ¿por qué no construir una segunda versión mejor desde cero y desechar la primera? Estas primeras versiones de los productos se conocen como *prototipos desechables*: se construye un producto que funciona y que se desecha.

Los prototipos desechables permiten aprender de los errores del pasado. Puede haber muchas lecciones útiles y problemas revelados en la primera versión que pueden evitarse en una segunda. Esto permite construir el producto de software sobre una arquitectura de segunda generación más robusta, en lugar de un sistema de primera generación con parches y correcciones.

¿LO SABÍAS?

Una característica común de los *prototipos ilustrados, exploratorios* y *desechables* es que nada del esfuerzo para construir los prototipos acabará en la versión final del producto. Se aprenden lecciones del esfuerzo, pero los recursos de desarrollo invertidos en la construcción del prototipo se pierden esencialmente.

Los dos últimos tipos de prototipos demostrarán cómo el esfuerzo de desarrollo en la construcción de un prototipo puede contribuir al producto final. La clave es tener un software que funcione para cada prototipo sucesivo, cualquiera de los cuales podría ser lanzado como una versión de su producto de software.

Prototipo incremental

Cuando un producto se construye y se lanza en incrementos, es un *prototipo incremental*. El prototipo incremental funciona por etapas, basándose en un sistema de triaje. "Triaje" significa evaluar cada uno de los componentes del sistema y asignarles una prioridad. Basándose en esa prioridad, los componentes de un producto se construyen de forma iterativa en incrementos desde el "más importante" al "menos importante". De este modo, los prototipos incrementales hacen uso de las filosofías de proceso que hay detrás de los modelos de procesos iterativos.

Las prioridades de las características de un producto de software se basan en tres categorías: "debe hacer", "debería hacer" y "podría hacer". A las características principales se les asigna la máxima prioridad: deben hacer. Todas las características de apoyo no críticas son elementos "que deberían hacer". El resto de características extrañas tienen la prioridad más baja: "podría hacer".

Sobre la base de estas prioridades, las iteraciones iniciales del desarrollo incremental se centran en la entrega de las prioridades "imprescindibles". El producto de software resultante, con las características principales, podría lanzarse como un prototipo incremental completo de primera generación.

Si los recursos lo permiten, las características de la prioridad "debería hacer" pueden desarrollarse para una segunda generación de versiones incrementales similares, seguidas de una tercera versión con características de la categoría "podría hacer". El resultado final de estas iteraciones es una serie de prototipos incrementales que van desde las características básicas esenciales hasta las completas.

Prototipo evolutivo

El último tipo de prototipo es el *prototipo evolutivo*. Este enfoque del prototipo es muy similar al de los prototipos incrementales, pero la diferencia se encuentra en el prototipo de primera generación. En el

prototipo evolutivo, el prototipo de primera generación tiene todas las características del producto, aunque algunas de ellas deban "evolucionar" o perfeccionarse. Un prototipo incremental de primera generación comparable sólo tiene un conjunto básico de características principales.

Tanto el prototipo incremental como el evolutivo son formas de crear un software que funcione y que pueda mostrarse a intervalos regulares para obtener más información. En la práctica, ambos enfoques pueden combinarse. Una de las principales ventajas es el aumento de la moral del equipo de desarrollo, ya que ven el producto en funcionamiento desde el principio y pueden aportar mejoras con el tiempo.

Última palabra sobre los prototipos

Teniendo en cuenta los diferentes tipos de procesos, piense en cómo podría encajar la creación de prototipos en el modelo en espiral y en el modelo de proceso unificado.

En el modelo en espiral, una primera iteración puede ser la creación de un prototipo ilustrativo. Esto podría lograrse con unos cuantos dibujos en papel para esbozar una idea de cómo funcionará el sistema. El mismo prototipo ilustrativo podría servir para la fase inicial del Proceso Unificado. Los prototipos ayudan a visualizar mejor lo que hace el producto y, por tanto, a tomar decisiones sobre las características basándose en el aspecto que podría tener el producto.

Las iteraciones posteriores podrían basarse en el prototipo ilustrativo con un enfoque de prototipo incremental o evolutivo. A partir de la primera iteración -una idea esbozada en unas cuantas hojas de papel-, se probará la idea, se esbozarán algunas características clave y se dará forma al siguiente prototipo. En poco tiempo, estará listo un prototipo que podrá llevarse a un cliente o a posibles inversores para demostrar que el producto es conceptualmente viable.

La idea central de todos los prototipos es obtener comentarios sobre las versiones del producto, para tomar decisiones informadas sobre el futuro del mismo.

Entrega continua

En el prototipo incremental o evolutivo, el producto se va añadiendo o perfeccionando de forma iterativa a lo largo del tiempo. La idea es empezar con un producto básico. Con el tiempo, se construyen sucesivos prototipos. Normalmente, estos prototipos se entregan al cliente para que dé su opinión. Sin embargo, esta noción de entrega puede ser ad hoc y requerir mucho trabajo. Por ejemplo, a los desarrolladores les puede llevar mucho trabajo manual construir e integrar el código en un prototipo ejecutable. Este prototipo sólo puede funcionar en el entorno del desarrollador, en su dispositivo específico, en lugar de como un producto adecuado para entregar al cliente.

La entrega continua aplica la automatización para que los lanzamientos intermedios de software en funcionamiento sean más disciplinados y frecuentes. Esto permite a los desarrolladores entregar más fácilmente versiones de un producto de forma continua a medida que se construye. El objetivo es que el software funcione, esté probado, esté listo para ejecutarse y se pueda liberar para otros. Por ejemplo, cada vez que un desarrollador envíe un cambio de código, éste se construirá, integrará, probará, lanzará y desplegará automáticamente. Lo ideal es que el tiempo que transcurre desde que se realiza un cambio en el producto hasta que lo prueba un usuario final sea corto y frecuente. Los problemas que surjan en cualquiera de estos pasos automatizados pueden ser advertidos antes, en lugar de más tarde.

Las versiones que no están listas no se ven obligadas a ser liberadas. Normalmente, las versiones de los

prototipos se colocan en canales específicos. Por ejemplo, puede haber un canal de "desarrolladores" para las versiones diarias que

generen los desarrolladores, un canal de "prueba" para un grupo más amplio interno de la empresa, o un canal "estable" para las versiones que están más "probadas". En cada canal pueden darse diferentes expectativas de retroalimentación y tolerancias a los problemas.

La práctica de la entrega continua encaja bien con los modelos de procesos iterativos como el Proceso Unificado. La fase más relevante en Unified es la de construcción, en la que múltiples iteraciones cortas construyen la funcionalidad del producto a través de una serie de prototipos que se entregan continuamente.

Dentro de cualquiera de estas iteraciones, el equipo de desarrollo trabajaría en una serie de tareas para construir el siguiente prototipo para su lanzamiento, incluyendo el diseño detallado, la escritura de código y la realización de casos de prueba. La determinación de las características en las que se centra cada iteración podría guiarse por un enfoque de creación de prototipos que sea incremental, evolutivo o ambos.

En el Proceso Unificado, la arquitectura o el diseño de alto nivel del producto se modelaría principalmente en la fase de elaboración. El diseño detallado define modelos que describen unidades, relaciones y comportamientos específicos de nivel inferior en el producto. En última instancia, los desarrolladores toman estos modelos, toman decisiones sobre algoritmos y estructuras de datos para escribir o generar el código, y preparan los casos de prueba correspondientes.

Para apoyar la entrega continua, se utilizan herramientas automatizadas para construir e integrar el código, ejecutar pruebas de bajo nivel, empaquetar el producto en una forma liberable, e instalarlo o desplegarlo en un entorno de prueba. Por ejemplo, la construcción del código puede requerir la reconstrucción de otro código dependiente. Para ser rápida, especialmente en el caso de productos de software de gran tamaño, la automatización de la compilación debe ser lo suficientemente inteligente como para reconstruir sólo lo que es realmente necesario.

A medida que se escribe el código, las características comienzan a tomar forma dentro del producto. La entrega continua garantiza la existencia de un prototipo listo para que los usuarios finales y los clientes lo prueben al final de la iteración.

En caso de que se abandone el proyecto, en teoría seguiría existiendo un producto publicable, aunque incompleto.

La calidad del producto también mejoraría. A través de las iteraciones, el producto se comprueba en pequeñas y frecuentes dosis, en lugar de todo de una vez.

Microsoft Daily Build

En el caso de la Microsoft Daily Build, cada "iteración" de la fase de construcción se realiza en un día, de ahí lo de "daily build".

El objetivo de la compilación diaria de Microsoft es garantizar que todos los programadores estén sincronizados al comienzo de cada actividad de compilación. Al seguir un proceso que hace que los desarrolladores integren su código en el sistema más amplio al final de cada día, las incompatibilidades se detectan más pronto que tarde.

Para controlar esto, Microsoft utiliza un sistema de **integración continua**. Cuando un desarrollador

escribe un trozo de código y quiere compartir ese código a distancia con cualquier otra persona del equipo, el código debe pasar primero por un proceso de prueba automático, que garantiza que funcionará con el proyecto en su conjunto.

Todos los desarrolladores pueden ver fácilmente cómo su trabajo encaja en el producto en su conjunto, y cómo su trabajo afecta a otros miembros del equipo.

La integración continua no sólo mantiene alta la moral de los desarrolladores, sino que también aumenta la calidad del producto. La compilación diaria permite a los desarrolladores detectar rápidamente los errores antes de que se conviertan en un problema real. Una compilación satisfactoria permite realizar pruebas durante la noche, y los resultados de las pruebas se comunican por la mañana. En Microsoft, como incentivo para las compilaciones diarias exitosas, un desarrollador cuyo código rompe la compilación debe supervisar el proceso de compilación hasta que llegue el siguiente desarrollador que rompa la compilación.

Última palabra sobre la entrega continua

La entrega continua puede incorporarse a un proceso iterativo como el Proceso Unificado para liberar prototipos incrementales o evolutivos.

Las combinaciones de enfoques, como el Proceso Unificado con la creación de prototipos y la entrega continua, pueden crear potentes herramientas para proyectos en los que la retroalimentación periódica es valiosa, la calidad del producto es importante y los problemas deben detectarse pronto. Este enfoque combinado es un proceso significativamente más robusto que los procesos lineales, por ejemplo, pero puede no ajustarse a todas las situaciones. Habrá circunstancias, especialmente en proyectos pequeños y a corto plazo, en las que la creación de la infraestructura necesaria para un proceso de este tipo llevaría más tiempo del que merece.

Los proyectos más pequeños pueden beneficiarse de procesos lineales más sencillos, como los modelos Waterfall, V o Sawtooth. Sin embargo, los procesos lineales son demasiado simplistas para proyectos muy grandes.

El proceso como herramienta: Elija la herramienta adecuada para el trabajo

Mantén la mente abierta cuando selecciones un proceso que se adapte a tu proyecto. Es un error pensar que los modelos de procesos de software iterativos o paralelos son siempre las mejores herramientas para el trabajo. Dependiendo de la situación, un proceso diferente podría ser más eficaz. Además, se pueden combinar aspectos de diferentes procesos para que se adapten mejor a las necesidades del proyecto. Haz lo que mejor funcione para tus proyectos.

Módulo 3: Prácticas ágiles

Al finalizar este módulo, serás capaz de:

- entender cómo funcionan los procesos lineales, iterativos y paralelos con los principios de Agile.
- describir los 12 principios de la Programación Extrema (XP), y cómo encajan en los métodos ágiles.
- reconocer qué tipos de proyectos funcionarán mejor con Extreme Programming (XP) y cuáles no.
- entender el papel de una metáfora del sistema.
- describir las características esenciales de las prácticas de XP y cómo encajan.
- describir las características esenciales de las prácticas de gestión.
- describir las características esenciales de las prácticas de Scrum: Scrum master, Scrum development team, Product owner, Sprint, Sprint planning, Sprint backlog, Daily scrum, Sprint review, Sprint retrospective.
- describir cómo Scrum se centra en la gestión.
- Comprender el proceso para establecer y mantener el backlog del producto, y quién es responsable de definir, priorizar y cambiar los requisitos.

Uso de Agile con modelos de procesos

Antes de continuar, por favor, aclárese con los términos que ya se han tratado.

Los procesos de software estructuran el trabajo de desarrollo de software en fases. Por ejemplo, el modelo de Proceso Unificado es un proceso iterativo que organiza el calendario de un producto en ciclos de fases, donde cada fase suele tener iteraciones. Un proceso describe los resultados esperados de las actividades realizadas en las fases, por lo que establece "qué" hay que hacer. Por ejemplo, los casos de uso se describen inicialmente en la fase inicial del Proceso Unificado. Las **prácticas de** software son técnicas sobre "cómo" desarrollar productos de software o gestionar proyectos de software de forma eficaz. Por ejemplo, al principio de un proceso, hay prácticas sobre cómo priorizar los requisitos del producto. Las **metodologías de** software son grupos definidos de prácticas.

Teniendo en cuenta los modelos de proceso para organizar el trabajo en el desarrollo de software, ¿cómo se pueden aplicar los principios Agile?

Uno de los principios de Agile se refiere a la entrega temprana y continua, de modo que haya suficiente oportunidad para una estrecha colaboración y retroalimentación para mejorar el producto. Los modelos Waterfall y V sólo entregan realmente al cliente al final del proceso de desarrollo, lo que no supone una integración temprana. Asimismo, la documentación exhaustiva no es un punto importante en el desarrollo ágil, pero los modelos Waterfall y V se basan en gran medida en la documentación y en la aprobación de la misma cuando se pasa de una fase a otra. La cultura de los contratos y las aprobaciones tampoco es ágil. Aunque el modelo Sawtooth proporciona un par de prototipos al cliente durante el proceso, no es suficiente. Así pues, los modelos de proceso lineal descritos no funcionan bien con los principios y prácticas ágiles.

Los modelos de procesos iterativos funcionan mucho mejor con las prácticas ágiles porque permiten la reflexión y la mejora. Los modelos iterativos también permiten lanzamientos frecuentes y continuos para recopilar información. Estas versiones se refinan y mejoran en la siguiente iteración. Las iteraciones cortas y los lanzamientos pequeños son

aspectos de Agile. Por lo tanto, el modelo en espiral con iteraciones más cortas y el modelo de Proceso Unificado con sus iteraciones dentro de las fases funcionarían bien con los principios y prácticas ágiles.

Prácticas ágiles

Las prácticas son técnicas, directrices o reglas que ayudan a desarrollar el producto de software o a gestionar el proyecto de software.

Las prácticas basadas en el Manifiesto para el Desarrollo Ágil de Software se denominan **prácticas ágiles**. Las prácticas ágiles se alinean con el Manifiesto Ágil. Por ejemplo, la práctica de desarrollo ágil de tener pequeñas versiones encaja con el principio ágil de entregar software funcional con frecuencia. La práctica de gestión ágil de organizar una reunión de revisión al final de cada iteración con el cliente encaja con el principio ágil de satisfacer al cliente mediante la entrega temprana y continua de software valioso.

Las prácticas se organizan en metodologías. Las metodologías que consisten en prácticas ágiles se llaman **metodologías ágiles**. El resto de este curso examinará tres metodologías populares, Extreme Programming, Scrum y Lean, así como una práctica de gestión llamada Kanban que funciona bien con estas metodologías.

Scrum

Scrum es una metodología ágil que consiste en prácticas de gestión ligeras que tienen relativamente poca sobrecarga. Sus prácticas son sencillas de entender pero muy difíciles de dominar en su totalidad. Scrum utiliza un enfoque que es a la vez iterativo e incremental. Hay evaluaciones frecuentes del proyecto, lo que aumenta la previsibilidad y mitiga el riesgo.

Los tres pilares de Scrum

Scrum se basa en tres pilares:

- Transparencia
- Inspección
- Adaptación

Con la transparencia, todo el mundo puede ver cada parte del proyecto, desde dentro del equipo y desde fuera del equipo. Scrum fomenta la inspección frecuente de los productos del trabajo y el progreso para detectar desviaciones indeseables de las expectativas. Las inspecciones, sin embargo, no se producen con tanta frecuencia como para impedir el desarrollo. Ken Schwaber, formador y autor de Scrum, dice que "Scrum es como una suegra, te señala todos los defectos". Los desarrolladores a veces dudan en adoptar scrum porque les señala todo lo que no están haciendo bien. Cuando se detecta una desviación, el equipo debe adaptarse y ajustarse para corregir el problema. Es importante acordar unas normas para el proyecto, como una terminología y unas definiciones comunes, como el significado de "hecho".

Sprints y Eventos Scrum

El calendario del proyecto consiste en una secuencia de sprints consecutivos. Un sprint es una iteración en la que se entrega al cliente un incremento de software funcional al final. Cada sprint tiene una duración fija y coherente. Esa duración fija se elige al inicio del proyecto y suele ser de una o dos semanas.

Para hacer posible los tres pilares, Scrum describe cuatro técnicas o eventos necesarios que ocurren dentro de un sprint:

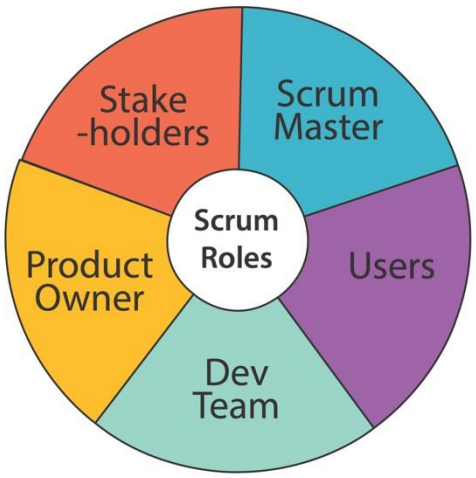
- **Planificación del** sprint (para establecer las expectativas del sprint)
- **Scrums diarios** (para garantizar que el trabajo se ajusta a estas expectativas)
- **Revisión del sprint** (para obtener información sobre el trabajo realizado)
- **Retrospectiva del sprint** (para identificar qué hacer mejor en el futuro)

La planificación del sprint tiene lugar al principio de un sprint para determinar lo que se completará en ese sprint. Esta planificación establece un objetivo del sprint, una visión de alto nivel de lo que se hará. El plan también identifica los requisitos concretos que hay que completar y las tareas de desarrollo asociadas.

Para mantener al equipo centrado en el plan, los scrums *diarios* son reuniones breves que se celebran a primera hora del día, en las que cada miembro del equipo expone lo que ha hecho anteriormente, en qué trabajará y qué puede bloquear su progreso. Para fomentar la brevedad y evitar reuniones largas, los scrums diarios se realizan con el equipo de pie. Por lo tanto, el scrum diario también se conoce como reunión diaria de pie.

La *revisión del sprint* se lleva a cabo al final del mismo para mostrar el software en funcionamiento al cliente y obtener su opinión. Una retrospectiva del sprint da la oportunidad de revelar las lecciones aprendidas que pueden abordarse para mejorar el trabajo futuro.

Una vez realizada la planificación del sprint, el trabajo avanza de acuerdo con el objetivo del sprint, los requisitos determinados y las tareas identificadas. Para no perturbar el trabajo en curso durante el sprint, se anotan los principales cambios de requisitos para más adelante.



Scrum define algunas funciones clave para un equipo de scrum. Además de los desarrolladores de software, hay un **propietario del producto** y un **scrum master**.

El *propietario del* producto es la única persona responsable del producto. Todos los requisitos del producto para los desarrolladores deben pasar por esta función. La lista de requisitos del producto se recoge en un **backlog del producto,** del que es responsable el propietario del producto. El propietario del producto también determina las prioridades de estos requisitos y se asegura de que estén

claramente
definidos.

El *scrum master se* asegura de que el equipo de scrum se adhiera a las prácticas de scrum, ayudando al propietario del producto y al equipo de desarrollo. Para el propietario del producto, esta ayuda incluye la sugerencia de técnicas para gestionar el backlog del producto, para obtener requisitos claros y para priorizar para obtener el máximo valor. Para los miembros del equipo de desarrollo, esta ayuda incluye el entrenamiento del equipo para autoorganizarse y eliminar los obstáculos. El scrum master también facilita los cuatro eventos mencionados anteriormente dentro de un sprint.

Dependiendo de la situación, hay varios mapeos para un cliente y un gerente de producto de software a estos roles. En una gran organización o un gran producto, el cliente, el gestor del producto de software, el propietario del producto y el scrum master pueden ser personas diferentes. En una pequeña empresa de nueva creación, puede haber un solapamiento significativo.

Naturalmente, el cliente podría ser el propietario del producto, que determina los requisitos y acepta el trabajo realizado. Sin embargo, si los clientes no tienen experiencia en la definición de requisitos claros para los desarrolladores, un gestor de productos de software puede representar al cliente y asumir el papel de propietario del producto. Si no hay un cliente real, como en un producto de mercado masivo, el gestor del producto de software puede representar a los usuarios finales y desempeñar el papel de propietario del producto.

Un líder del equipo podría ser el scrum master para facilitar las prácticas de scrum del equipo. En una empresa pequeña, un gestor de productos de software con cualidades de liderazgo puede tener que asumir el papel de scrum master. En cualquier caso de participación, es importante que los gestores de productos de software entiendan los roles de scrum y sus responsabilidades.

Equipo de desarrollo de Scrum

Los desarrolladores de un equipo de scrum, también conocido como equipo de desarrollo de scrum, deben autoorganizarse. Nadie fuera del equipo de desarrollo, ni siquiera el scrum master o el propietario del producto, les dice cómo convertir el backlog de requisitos para un sprint en tareas de desarrollo para producir un incremento de software de trabajo. Los equipos de desarrollo de Scrum son pequeños, idealmente entre tres y nueve personas. Son auto-contenidos, que consiste en todos y todo lo necesario para completar el producto. Además, cada miembro generalmente se encarga de tareas mixtas, como hacer tanto la codificación como las pruebas, en lugar de tener codificadores y probadores dedicados. No hay subequipos especiales. Todos los miembros del equipo son responsables de los productos del equipo. La responsabilidad recae en todo el equipo.

Definición de "Hecho"

Una definición importante en scrum es el significado de estar **"hecho".** Todo el equipo de scrum debe estar de acuerdo con esa definición. Normalmente, un requisito para una característica se considera hecho cuando se codifica, se prueba, se documenta y se acepta. El objetivo de un sprint es conseguir que los requisitos esperados estén realmente hechos, o "hechos", en lugar de tener las características requeridas en estados a medio hacer o entregar un software que no funciona. Conseguir que un requisito esté hecho requiere comunicación, integración y sincronización por parte de los miembros del equipo. El trabajo a medias puede surgir cuando un miembro trabaja de forma independiente en algo sólo para estar ocupado, en lugar de ayudar a cerrar cada requisito. Esto puede significar que a veces hay que ir despacio para asegurarse de que cada requisito está realmente hecho.

Programación extrema (XP)

Prácticas

La programación extrema (XP) es una metodología ágil que consiste en prácticas de desarrollo eficaces para lograr la satisfacción del cliente. Las prácticas se centran en la entrega constante de software, la respuesta al cambio, el trabajo en equipo eficaz y la autoorganización. Estas prácticas siguen los principios descritos en el Manifiesto Ágil. Además de fomentar la simplicidad, la comunicación y la retroalimentación, XP promueve el respeto y la valentía.

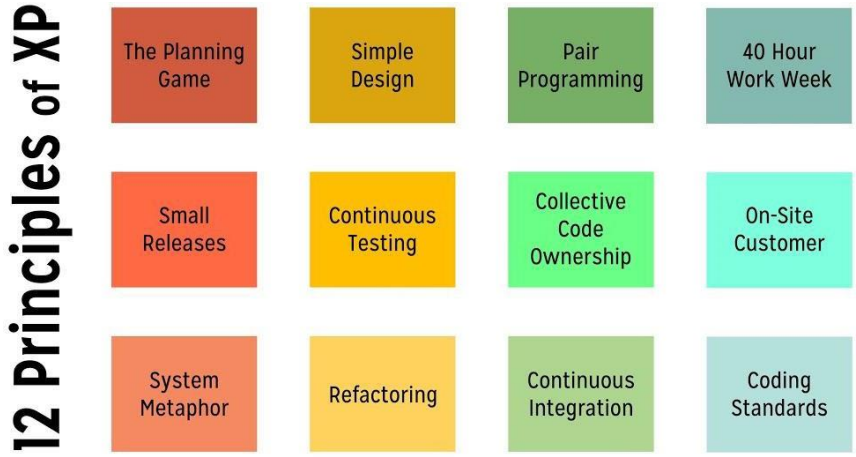
Todo el mundo es valorado y respetado por lo que aporta individualmente al proyecto, ya sea por su experiencia empresarial, sus conocimientos del sector o sus habilidades de desarrollo. Todos se consideran iguales, de modo que el cliente, el gestor de productos de software y los miembros del equipo de desarrollo están todos al mismo nivel, trabajando juntos.

En cuanto a la valentía, la siguiente cita es muy adecuada.

Diremos la verdad sobre los progresos y las estimaciones. No documentamos excusas de fracaso porque planeamos tener éxito. No tememos nada porque nadie trabaja solo. Nos adaptaremos a los cambios siempre que se produzcan.

- Programación Extrema, 1999

Para aplicar XP, hay que seguir 12 prácticas. Las prácticas funcionan juntas, por lo que seguirlas todas proporciona el mayor beneficio.



Práctica 1: El juego de la planificación

El cliente y el equipo de desarrollo trabajan juntos en la planificación del producto. Esto implica una sesión de planificación más amplia al inicio del proyecto y sesiones más pequeñas en cada iteración. En la sesión más amplia se determinan las características necesarias del producto, sus prioridades y el momento en que deben lanzarse a lo largo del tiempo. Una sesión más pequeña se centra en las características que deben completarse en una iteración y determina las tareas de desarrollo necesarias.

Práctica 2: Pequeños lanzamientos

Los lanzamientos deben ser lo más frecuentes posible para obtener una gran cantidad de comentarios, lo que se consigue mejor si los lanzamientos son pequeños en términos de nuevas funcionalidades requeridas. Hay que priorizar las funciones necesarias para aportar valor al cliente con prontitud. El cliente y el equipo de desarrollo deben encontrar el equilibrio adecuado entre lo que el cliente quiere que se complete primero y lo que puede desarrollarse antes. Las versiones más pequeñas también permiten que las estimaciones sean más seguras; es más fácil mirar una o dos semanas por delante, en lugar de meses. Así que, en el extremo, hay que mantener las iteraciones muy cortas.

Práctica 3: Metáfora del sistema

Una metáfora del sistema facilita la explicación del producto a otra persona; puede ser una analogía para describir el producto a alguien que no sea técnico. Por ejemplo, la metáfora del escritorio en las interfaces gráficas de usuario ayuda a explicar las interacciones del ordenador en términos de algo más familiar en el mundo real. Otro ejemplo es la metáfora del carrito de la compra en línea, que se basa en un concepto de la compra en el mundo real. Una metáfora también puede describir la implementación del producto a alguien más técnico. Por ejemplo, la metáfora de las tuberías y los filtros en la arquitectura del software ayuda a explicar cómo se estructura la implementación de un producto en torno a la conexión y el manejo de elementos informáticos con flujos de información.

Práctica 4: Diseño sencillo

Céntrese en el diseño más sencillo que satisfaga las necesidades del cliente. Los requisitos cambian, por lo que sería un desperdicio hacer diseños elaborados para algo que cambiará. Diseñe lo necesario para que las características requeridas funcionen. No sobredimensione el diseño para un futuro que puede no llegar. Así que, en el extremo, haz lo más sencillo que funcione.

Práctica 5: Pruebas continuas

En XP, las pruebas se preparan para una característica o funcionalidad requerida antes de escribir su correspondiente código fuente. De este modo, los esfuerzos se centran en primer lugar en comprender lo que se necesita, en hacer que la interfaz de usuario o de programación sea sencilla y en preparar las pruebas adecuadas para verificar que se ha conseguido el comportamiento requerido. Si la prueba es difícil de escribir, es una señal para rehacer la interfaz o el requisito original. La implementación de la característica o funcionalidad requerida viene después. Esta práctica se conoce como **Desarrollo Dirigido por Pruebas** o **TDD**. La automatización de las pruebas permitirá que se ejecuten continuamente. Así que, en el extremo, escriba primero las pruebas y ejecútelas todo el tiempo.

Hay dos tipos principales de pruebas: **las de aceptación** y **las unitarias**. Una prueba de *aceptación* sirve para que el cliente verifique si una característica o requisito del producto funciona según lo especificado y, por tanto, es aceptable. Una prueba de aceptación suele consistir en algo que hace un usuario final, que toca una gran parte del producto. Esta prueba puede estar automatizada o ser un conjunto de instrucciones y resultados esperados que sigue un humano. Los desarrolladores escriben y ejecutan una prueba unitaria para verificar si la funcionalidad de bajo nivel funciona correctamente. Una prueba unitaria suele ser específica, como probar un algoritmo o una estructura de datos en algún módulo de la

implementación del software.

Considere una aplicación de medios sociales con una característica requerida para que un usuario final pueda publicar un mensaje. Una prueba de aceptación para esta función implicaría acciones que los usuarios finales realizan, como iniciar una publicación, introducir un mensaje y enviar la publicación. La prueba comprobaría que las acciones funcionan básicamente para algunos datos de prueba. Detrás de esta función, y del producto en general, hay una funcionalidad de bajo nivel para almacenar publicaciones. Las pruebas unitarias comprobarían de forma más exhaustiva que esta funcionalidad funciona, como por ejemplo el tratamiento de caracteres internacionales o textos muy largos.

Práctica 6: Refactorización

La refactorización es una práctica para reestructurar el diseño interno del código sin cambiar su comportamiento. El objetivo es mejorar el diseño en pequeños pasos, según sea necesario, para permitir que se añadan nuevos requisitos al producto con mayor facilidad, adaptándose así al cambio. Por ejemplo, la refactorización podría reorganizar gradualmente el código para permitir una ampliación más fácil. Un módulo grande y poco manejable puede descomponerse en piezas más pequeñas, más cohesionadas y más comprensibles. Se puede eliminar el código innecesario. A medida que avanza la refactorización, las pruebas unitarias se ejecutan repetidamente para garantizar que el comportamiento del código sigue siendo el mismo.

Si la refactorización se aplaza o no se hace, los cambios son cada vez más difíciles. El trabajo no realizado es como incurrir en una obligación, conocida como **deuda técnica**, que hay que "pagar" en el futuro. Una pequeña cantidad puede estar bien para seguir avanzando en el producto, pero podría producirse una crisis si se acumula una cantidad suficientemente grande.

Práctica 7: Programación por parejas

Para garantizar una alta calidad, XP emplea la programación por parejas, en la que dos desarrolladores trabajan codo con codo en un ordenador para trabajar en una única tarea, como escribir el código de una función necesaria. En las revisiones de código habituales, un desarrollador escribe un código y, una vez terminado, otro desarrollador lo revisa. En cambio, en la programación en parejas, el código se revisa todo el tiempo al tener siempre otro par de ojos para considerar cada edición. De este modo, se aúnan habilidades y perspectivas complementarias a la hora de resolver un problema o generar alternativas. Se pueden llevar a cabo ideas más arriesgadas pero más innovadoras, ayudadas por la valentía que aporta tener un compañero. Emparejar a un desarrollador senior y otro junior puede fomentar el aprendizaje, ya que el senior imparte consejos estratégicos y el junior ofrece una nueva perspectiva. En general, las parejas no son estáticas, sino que cambian dinámicamente. Así que, en el extremo, haz revisiones de código todo el tiempo.

Práctica 8: Propiedad colectiva del código

Aunque un par de desarrolladores específicos puedan implementar inicialmente una parte concreta del producto, no son "dueños" de ella. Para fomentar las contribuciones, otros miembros del equipo pueden añadir algo a ella o a cualquier otra parte del producto. Así, el trabajo producido es el resultado del equipo, no de los individuos. Así, el éxito y el fracaso del proyecto recae en el equipo, no en desarrolladores concretos.

Práctica 9: Integración continua

En XP, los desarrolladores combinan su código con frecuencia para detectar los problemas de integración con antelación. Esto debería ser al menos una vez al día, pero puede ser mucho más frecuente. Si se escriben primero las pruebas, éstas también pueden ejecutarse con frecuencia para destacar el trabajo pendiente o los problemas imprevistos.

Práctica 10: Semana laboral de 40 horas

XP pretende ser amigable con el programador. Respeta el equilibrio entre el trabajo y la vida del desarrollador. En el momento de la crisis, se permite hasta una semana de horas extras, pero varias semanas de horas extras serían un signo de mala gestión o estimación.

Práctica 11: Cliente in situ

El cliente está situado cerca del equipo de desarrollo durante todo el proyecto para aclarar y responder a sus preguntas.

Práctica 12: Normas de codificación

Todos los desarrolladores acuerdan y siguen un estándar de codificación que especifica las convenciones sobre el estilo, el formato y el uso del código. Esto facilita la lectura del código y fomenta la práctica de la *propiedad colectiva*.

Ideas adicionales de gestión

Además de las 12 prácticas básicas, XP también sugiere que el equipo disponga de un espacio de trabajo abierto y dedicado, en el que las personas trabajen juntas en persona. Se colocan varios ordenadores en el centro de la sala, a disposición de todos. Una mesa de reuniones y mucho espacio en las paredes para pizarras blancas o notas adhesivas permiten al equipo colaborar y hacer una lluvia de ideas.

El espacio de trabajo abierto permite que la gente se mueva. Fomenta la comunicación, propaga el conocimiento y permite unas relaciones de trabajo flexibles, de modo que todos acaban aprendiendo a trabajar en todas las partes del producto, sin centrarse demasiado en la documentación técnica.

Una última idea de gestión es cambiar el propio XP. Es decir, arreglar XP *cuando* se rompe. Aunque la adopción de todas las prácticas básicas se promociona para maximizar los beneficios, no todas las prácticas funcionarán para cada proyecto o equipo. Cuando una práctica no funciona, hay que cambiarla. Las reuniones retrospectivas son una forma de que el equipo discuta lo que funciona y lo que no. Reúna datos que respalden las prácticas que debe adoptar.

Inconvenientes

Una de las limitaciones es que XP está pensado y adaptado para equipos de desarrollo pequeños, por ejemplo, de no más de diez personas. Además, puede que no sea posible disponer de un cliente in situ con el equipo de desarrollo.

Además, XP no ofrece prácticas para definir el diseño arquitectónico del software. En su lugar, este diseño surge en respuesta a los cambios de requisitos y a la refactorización, utilizando la metáfora del sistema como guía. Este enfoque produciría un software que funcionara pronto, pero podría implicar más retrabajo que si se dedicara algún esfuerzo a planificar la arquitectura.

La programación en parejas es un cambio importante para muchos desarrolladores. Algunos equipos prosperan en este entorno de colaboración, mientras que otros no. Esta práctica requiere personalidades respetuosas que trabajen bien juntas. En realidad, no todo el mundo es compatible con los demás.

Módulo 4: Otras prácticas

Al finalizar este módulo, serás capaz de:

- Comprender que las prácticas ágiles son aptas para cambiar porque la tecnología cambia y evoluciona.
- Explicar las prácticas ágiles distintas de Scrum y Extreme Programming, como: Método de Desarrollo de Sistemas Dinámicos, Desarrollo Orientado a las Características, Desarrollo Orientado al Comportamiento y Scrumban.
- Describa las diferencias entre el Proceso Unificado y el Proceso Unificado Ágil.
- Describa las prácticas de Lean, entre ellas: eliminar el desperdicio, amplificar el aprendizaje, decidir lo más tarde posible, entregar lo más rápido posible, capacitar al equipo, incorporar la calidad.
- Explicar que amplificar el aprendizaje es basar sabiamente nuevas características en [illegible]
- [illegible]
- [illegible]
- Explique qué significa "hecho" desde la perspectiva de Kanban.

Otras metodologías ágiles

Las metodologías de ingeniería de software han evolucionado a lo largo de los años. Aunque Scrum y Extreme Programming son muy populares y se utilizan ampliamente en la industria, también se han propuesto otras metodologías ágiles con prácticas para abordar limitaciones particulares.

Por ejemplo, una variante se llama Proceso Unificado Ágil, o AgileUP. Como su nombre indica, esta metodología ágil combina las prácticas ágiles con el modelo del Proceso Unificado. AgileUP aplica los principios de Agile para aumentar la moral de los desarrolladores y mejorar la colaboración. Utiliza prácticas ágiles, como el desarrollo dirigido por pruebas y las versiones pequeñas para complementar el Proceso Unificado básico. Al igual que el Proceso Unificado, AgileUP utiliza la misma estructura de fases. La metodología hace hincapié en la simplicidad, la potenciación de las personas y la adaptación de la metodología a las necesidades de cada proyecto.

En general, utilice un proceso, una metodología o una práctica que se adapte a las necesidades del producto, el equipo y el proyecto. Esto puede suponer seguir de cerca algo ya establecido o empezar por ahí y personalizarlo según sea necesario.

Otras metodologías ágiles son

- Método de desarrollo de sistemas dinámicos
- Desarrollo orientado a las características
- Desarrollo orientado al comportamiento
- Scrumban

Diseño de software ajustado

Una metodología ágil que ha ganado cada vez más atención es el **desarrollo de software ajustado (Lean Software Development)**. Su origen se inspira en la industria manufacturera, donde la producción **Lean surgió del** "Sistema de Producción Toyota". En Toyota, su enfoque se utilizó eficazmente para reducir los residuos en el proceso de producción y aumentar la calidad de sus vehículos.

Siete principios de Lean

Lean Software Development, o Lean para abreviar, es una metodología ágil que permite crear mejor software. Lean se basa en siete principios:

1. Eliminar los residuos
2. Ampliar el aprendizaje
3. Decidir lo más tarde posible
4. Entregar lo más rápido posible
5. Dar poder al equipo
6. Calidad de construcción en
7. Ver todo el

Estos principios pueden ser eficaces tanto para proyectos pequeños como grandes.

Eliminar los residuos

¿Cómo puede ser un despilfarro el desarrollo de software? Piense en la posible pérdida de tiempo y esfuerzo que puede suponer: requisitos poco claros, cuellos de botella en el proceso, defectos en el producto y reuniones innecesarias. Se trata de despilfarros por deficiencias.

El despilfarro también puede disfrazarse de eficiencia porque estar "ocupado" no siempre equivale a ser "productivo". Un equipo ocupado que no se centra en el desarrollo de las características principales y necesarias puede producir fácilmente características "extra" que interfieren con la funcionalidad principal del producto final. La codificación puede ser un despilfarro si no se planifica para ser un lanzamiento. Todo lo que no añade valor al producto se considera un despilfarro y debe identificarse y eliminarse.

Amplificar el aprendizaje

Explore suficientemente todas las ideas antes de proceder a las acciones.

No hay que conformarse y centrarse en una sola idea antes de explorar a fondo otras opciones. Este principio trata de encontrar la mejor solución a partir de una serie de alternativas. El pensamiento lateral genera enfoques alternativos, y cuantas más alternativas se consideren, mayor será la probabilidad de que surja una solución de calidad. El mecanismo biológico de la selección natural es una buena metáfora del aprendizaje ampliado. Los más

Las soluciones exitosas surgen de la más amplia variedad de alternativas. Esta exploración nos llevará a construir el producto adecuado.

Este principio también fomenta la realización de pruebas después de cada construcción del producto. Considera los fallos como oportunidades para ampliar el aprendizaje sobre cómo mejorar.

Decidir lo más tarde posible

A menos que tengas que tomar una decisión *ahora mismo*, no la tomes todavía.

Decidir lo más tarde posible permite explorar muchas alternativas (ampliar el aprendizaje) y seleccionar la mejor solución en función de los datos disponibles.

Las decisiones tempranas y prematuras sacrifican el potencial de una solución mejor y el "producto adecuado".

Entregar lo más rápido posible

Existe una relación entre los principios de Lean. La eliminación del despilfarro requiere pensar en lo que se hace; por lo tanto, hay que amplificar el aprendizaje. El aprendizaje ampliado requiere un tiempo adecuado para considerar alternativas; por lo tanto, debe decidir lo más tarde posible. Decidir lo más tarde posible requiere un trabajo productivo enfocado; por lo tanto, debe entregar lo más rápido posible.

Entregar lo más rápido posible consiste principalmente en hacer evolucionar un producto en funcionamiento a través de una serie de iteraciones rápidas. Cada versión puede centrarse en las características principales del producto, de modo que no se pierda tiempo y esfuerzo en características no esenciales. Las versiones frecuentes ofrecen la oportunidad de que el cliente dé su opinión para mejorar el producto.

Capacitar al equipo

El mejor ejecutivo es aquel que tiene el suficiente sentido común para elegir a gente buena para hacer lo que quiere, y la suficiente autocontención para no entrometerse con ellos mientras lo hacen.

- Theodore Roosevelt

Los cuatro primeros principios Lean tratan del proceso de desarrollo del producto adecuado. Los tres principios restantes describen *cómo se* puede hacer eficiente el proceso.

El objetivo de Lean es capacitar a los equipos que siguen sus prácticas. Anima a los directivos a escuchar a sus desarrolladores, en lugar de decirles cómo tienen que hacer su trabajo. Un gestor eficaz deja que los desarrolladores decidan cómo hacer el software.

Construir la calidad en

El objetivo es construir un producto de calidad.

Las formas de crear software de calidad son: realizar revisiones de los productos del trabajo, preparar y ejecutar pruebas automatizadas, refactorizar el código para simplificar su diseño, proporcionar documentación útil y hacer comentarios significativos sobre el código.

El desarrollo de una solución de calidad reduce la cantidad de tiempo que los desarrolladores deben dedicar a corregir los defectos. Al aplicar proactivamente este principio para incorporar la calidad al producto, el equipo de desarrollo elimina la pérdida de tiempo y esfuerzo.

Ver el conjunto

Mantener la atención en la experiencia del usuario final.

Un producto de calidad está cohesionado y bien diseñado en su conjunto. Los componentes individuales deben complementar la experiencia completa del usuario.

Principios adicionales de los Desarrolladores de Software Lean

Mary y Tom Poppendieck escribieron sobre estos siete principios en su libro de 2003, *Lean Software Development: An Agile Toolkit*. Con el tiempo, la comunidad Lean ha añadido otros principios.

Utilizar el método científico

Inicie experimentos para recoger datos y probar ideas.

Este principio anima a los gestores de productos de software a basar las características del producto en datos reales de los usuarios, en lugar de basarse en corazonadas, conjeturas o intuiciones. Recoger y analizar estos datos permite a los clientes y a los desarrolladores de software tomar decisiones informadas sobre el producto. Ganar credibilidad respaldando las posibles decisiones con datos.

Fomentar el liderazgo

Permitir que los desarrolladores individuales sean valientes, innovadores, inspiradores y colaboradores. Este principio amplía el poder del equipo para sacar lo mejor de cada persona del equipo.

Garantía

Mary y Tom Poppendieck escribieron esta garantía en su libro de 2003, *Lean Software Development: An Agile Toolkit*:

> Se garantiza que los principios Lean han sido probados y comprobados en muchas disciplinas, y cuando se aplican correctamente, se garantiza que funcionan para el desarrollo de software.
> Aplicación adecuada

> significa que se emplean todos los principios lean y que se utilizan herramientas de pensamiento para traducirlos en prácticas ágiles adecuadas al entorno. Esta garantía no es válida si las prácticas se transfieren directamente de otras disciplinas o dominios sin pensar, o si se ignoran los principios de *capacitar al equipo* y *construir la integridad en* [énfasis añadido] (p. 186).

En esencia, no funcionará utilizar prácticas de gestión de control para hacer posible el Lean. Aunque el núcleo de Lean consiste en eliminar los residuos y mejorar el producto, la forma más eficaz de conseguirlo es facultar al equipo de desarrollo para que aplique el producto de la forma que desee y garantice la calidad en la construcción del mismo.

Kanban

Kanban es una técnica para organizar y seguir el progreso de los proyectos de forma visual, que se utiliza ampliamente incluso fuera del desarrollo de software ágil o lean. *Kanban* es una palabra japonesa que, traducida de forma libre, significa tablero o tabla de señalización. Así, la técnica Kanban utiliza un tablero, con un conjunto de columnas etiquetadas por las etapas de realización.

Seguimiento de las tareas

Un tablero Kanban puede seguir el estado de las tareas completadas por un equipo. Así, para las tareas simples, las etiquetas de columna adecuadas para las etapas de finalización serían "Por hacer", "Haciendo" y "Hecho". Inicialmente, todas las tareas necesarias se escriben en notas adhesivas y se colocan en la columna "Por hacer". A medida que el trabajo avanza y una tarea por hacer entra en el "estado de realización", su nota adhesiva se retira de la columna "Por hacer" a la columna "Haciendo". Cuando la tarea está terminada, la nota adhesiva pasa de la columna "Haciendo" a la columna "Hecho". Para las tareas necesarias, el tablero muestra fácilmente sus estados de un vistazo a todo el equipo. El equipo tiene plena visibilidad de lo que hay que hacer, lo que se está haciendo y lo que se ha hecho. Es motivador ver cómo las tareas marchan de forma constante por el tablero y es útil ver cuándo una tarea aparece bloqueada.

Seguimiento de los requisitos del producto

Un tablero Kanban puede ser utilizado eficazmente con Scrum. Por ejemplo, el tablero de arriba puede rastrear el estado de las tareas de los desarrolladores que están previstas para ser realizadas dentro de un sprint. Además, un tablero Kanban diferente puede seguir el estado de los requisitos individuales en el backlog del producto a través de las etapas de ser considerado "hecho" en Scrum. Aquí, las etiquetas de columna de ejemplo para las etapas de realización podrían ser: "Backlog", "Análisis", "Diseño", "Pruebas", "Codificación", "Revisión", "Liberación" y "Hecho", siguiendo las fases de algún proceso de desarrollo de software. Al principio, los requisitos del backlog se escriben en notas adhesivas y se colocan en la columna "Backlog". A medida que el trabajo avanza, los requisitos se van desplazando por el tablero, hasta llegar a la columna "Hecho". Los requisitos se moverán a diferentes ritmos a través del tablero hacia su finalización.

En general, un requisito permanece en una determinada columna intermedia hasta que el equipo tiene

realmente la capacidad de iniciar la siguiente etapa para él. Es decir, un requisito suele ser "arrastrado" a la siguiente columna. Sin embargo, una táctica es tener una columna especial "Siguiente" justo después de "Backlog", para que un cliente o propietario del producto pueda

sacar un requisito de alta prioridad del backlog y "empujarlo" explícitamente al proceso de desarrollo para que se trabaje en él en un sprint. De este modo, Kanban puede utilizarse para organizar el trabajo, así como para su seguimiento.