DESARROLLO ÁGIL DE SOFTWARE

UOC Corporate

ÍNDICE

INTRODUCCIÓN

Supongamos que tu departamento necesita desarrollar una herramienta de software o que estás creando un producto que requiere el desarrollo de un software. Muy probablemente, una de las primeras cosas que querremos saber es cuánto va a costar y cuándo estará finalizado. El problema es que, probablemente, solo podamos tener una idea aproximada: deberíamos saber si son semanas, meses o años, pero es difícil que, en un proyecto de meses, sepamos exactamente el número de días que vamos a necesitar.

Por otro lado, es posible que, a medida que avance el desarrollo, vayamos descubriendo nuevos requisitos o que los requisitos iniciales cambien.

> El desarrollo ágil de software intenta solucionar estos problemas asumiendo que trabajamos en un entorno de alta incertidumbre y que, por lo tanto, debemos orientar los esfuerzos a trabajar con diferentes niveles de precisión.

A continuación, veremos cómo gestionar un proyecto de manera ágil y, concretamente, veremos cómo **Scrum** y **Kanban** nos pueden ayudar a mejorar nuestro proceso de desarrollo de software para adaptarlo a las condiciones concretas de nuestro proyecto y nuestro equipo.

OBJETIVOS

En esta unidad podrás:

1. Conocer el **manifiesto ágil**.
2. Conocer los **principios del desarrollo ágil**.
3. Saber qué es una **historia de usuario** y cómo se organizan en un ***backlog***.
4. Saber cómo se aplica el **ciclo de vida iterativo e incremental** en un proyecto ágil.
5. Conocer el método **Scrum**.
6. Conocer la herramienta **Kanban**.

EL MANIFIESTO ÁGIL

¿Cómo nace el desarrollo ágil?

El desarrollo ágil nace en un momento en el que la industria del desarrollo llevaba varias décadas intentando controlar los proyectos de desarrollo de software aplicando metodologías estrictas con la idea de que métodos más estrictos harían que los proyectos estuviesen más "controlados".

A pesar de ello, los resultados no eran muy alentadores. Los proyectos de desarrollo de software seguían presentando una gran variabilidad en cuanto a resultados y, de hecho, la gran mayoría acababan sin ser capaces de cumplir las expectativas originales (o bien acababan cancelados o bien incumplían plazos, costes o alcance).

Es en ese contexto que, en febrero de 2001, 17 personas relacionadas profesionalmente con el desarrollo de software (y, en muchos casos con el de alguna metodología) se reunieron para compartir experiencias y poner en común los factores que, en su experiencia, contribuían de manera más decisiva al éxito de los proyectos de desarrollo de software. La conclusión de esta reunión fue el llamado **"Manifiesto por el desarrollo ágil de software"**.

El manifesto ágil

«Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- ***Individuos e interacciones** sobre procesos y herramientas.*
- ***Software funcionando** sobre documentación extensiva.*
- ***Colaboración con el cliente** sobre negociación contractual.*
- ***Respuesta ante el cambio** sobre seguir un plan.*

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda».

Dicho manifiesto era el mínimo común denominador en el que todos los participantes estaban de acuerdo y es la base de lo que se ha venido a denominar **desarrollo ágil de software**. El manifiesto incluye también doce principios que rigen la aplicación en la práctica del manifiesto.

El desarrollo ágil

> El desarrollo ágil no es una metodología sino una serie de principios que se pueden cumplir aplicando varias metodologías.

No obstante, la manera más fácil de poner en práctica estos principios es siguiendo algún método concreto. De hecho, entre los firmantes del manifiesto había representantes de las metodologías eXtreme Programming, Scrum, DSDM, Adaptive Software Development, Crystal o Feature-Driven Development, entre otras.

1. Individuos e interacciones frente a procesos y herramientas

Supongamos que en nuestra organización tenemos dos equipos de desarrollo: en uno las diferentes personas que lo componen se complementan unas a otras y, además, está bien avenido y comprometido con el éxito del proyecto. Por otro lado, tenemos un equipo donde hay áreas de conocimiento que no están bien cubiertas (por ejemplo, no hay un experto en bases de datos) o donde la relación entre compañeros no es buena, y los diferentes miembros del equipo no se ayudan unos a otros y no intercambian conocimiento.

Es muy probable que el primer equipo lleve a cabo sus proyectos de desarrollo con éxito independientemente del método seguido, mientras que, en el caso del segundo, es posible que tengan problemas sea cual sea el método de desarrollo utilizado.

Por este motivo un elemento habitual en los métodos de desarrollo ágiles es la **inspección y adaptación**.

> Medir el desempeño del equipo de desarrollo y adaptar el proceso para mejorar el resultado.

De este modo, cada equipo acabará con un proceso diferente que se adecue a sus circunstancias y a las personas en concreto que forman ese equipo.

2. Software funcionando sobre documentación extensiva

Un malentendido habitual al hablar de desarrollo ágil es que se cree que este está reñido con la documentación, lo cual no es cierto en absoluto. Lo que sí es cierto es que en las metodologías anteriores era muy habitual que se generase un gran volumen de documentación que quedaba rápidamente obsoleta y no se actualizaba a medida que el software iba cambiando o que, simplemente, no aportaba ningún valor y se hacía simplemente porque la metodología así lo indicaba.

En este sentido, los métodos ágiles intentan generar el mínimo de documentación y que, en la medida de lo posible, esta esté escrita en **formatos que sean automatizables**, de modo que se pueda mantener actualizada con poco esfuerzo.

Lo que se intenta evitar es entregar un sistema con documentación obsoleta o errónea (en este caso la documentación en lugar de cumplir su función lo que hará será generar más problemas) o, peor aún, un conjunto extenso de documentación que no se corresponde a ningún sistema software (¿cuál es el valor de una documentación si el sistema que documenta no existe?).

3. Colaboración con el cliente sobre negociación contractual

Dada la naturaleza de los proyectos de desarrollo de software, es bastante habitual que haya cambios durante estos debido a que tanto los desarrolladores como los clientes van entendiendo mejor el problema que han de resolver, o cambia la situación del negocio (por ejemplo, un competidor lanza un producto parecido al nuestro o hay un cambio en la normativa) o, simplemente, alguna de las premisas sobre la tecnología que hay que utilizar resulta no ser correcta (por ejemplo, se pensaba utilizar una tecnología pero nos encontramos con algún problema que nos impide hacerlo y necesitamos buscar una tecnología alternativa).

En este sentido, es necesario establecer una relación diferente a la relación tradicional en la que se cerraba alcance, presupuesto y fechas, y crear así un marco de relación más flexible (los llamados **contratos ágiles**) que permita flexibilizar, al menos, una de las variables.

> Un equipo ágil, a la hora de organizar su trabajo, debe tener en cuenta este marco de relación y estar preparado para aceptar los cambios que puedan venir por parte del cliente sin que estos supongan un gran problema para el cliente ni para el equipo.

Esto acaba afectando, incluso, a la manera de programar el sistema, ya que se requieren arquitecturas más flexibles que permitan dar cabida a nuevos requisitos cuando estos lleguen.

4. Respuesta ante el cambio sobre seguir un plan

En la línea de lo que hemos ido comentando, uno de los factores fundamentales de un método ágil es que el plan trazado inicialmente se sigue en la medida en que sea válido, pero, ante un cambio, en lugar de intentar mantenerse sobre el plan inicial (lo cual a menudo es poco realista) se revisa el plan y se decide cómo adaptar el proyecto en función del resultado de la revisión.

Una manera de conseguirlo es mediante el **desarrollo incremental**.

> El desarrollo incremental consiste en desarrollar el sistema mediante pequeños **incrementos** de funcionalidad de manera que se va creando el sistema final de manera progresiva.

En lugar de programar toda la funcionalidad a la vez, se crea un esqueleto del sistema que implementa la arquitectura básica y se van añadiendo funcionalidades de manera progresiva.

El desarrollo incremental tiene la ventaja de que permite adelantar el retorno de la inversión, ya que tenemos un sistema parcial que es utilizable antes de finalizar el desarrollo completo. Esto también nos ayudará a obtener mejor feedback sobre si estamos desarrollando el producto adecuado o no, ya que podremos ponerlo antes a disposición de los usuarios finales.

También es una ventaja si se agota el presupuesto o llega la fecha límite del proyecto antes de tiempo, puesto que, aunque no tengamos el producto completo, al menos tenemos un producto parcial que podría ser suficiente para obtener un retorno de la inversión efectuada.

Otra ventaja de los procesos incrementales es que pasamos por todas las etapas del desarrollo para cada funcionalidad y así nos encontramos antes con los problemas de las últimas etapas del desarrollo (por ejemplo, que a la hora de implementar veamos que una tecnología que íbamos a utilizar no da el rendimiento adecuado y que, por lo tanto, debe replantearse la arquitectura del sistema).

El desarrollo incremental

¿Cómo podríamos aplicar el desarrollo incremental?

Imagina que quisiéramos desarrollar un aula virtual y detectamos las siguientes necesidades:

- Enviar una actividad
- Plantear una duda
- Resolver una duda
- Publicar una nota
- Descargar la bibliografía de referencia

Para desarrollar este sistema de manera incremental, podríamos empezar con una primera versión que solo permitiese **plantear dudas** (de momento, las resolveríamos por correo electrónico). Esta primera versión ya podría empezar a utilizarse antes de acabar el sistema completo si así se considerase oportuno.

Más adelante podríamos añadir la posibilidad de **resolver una duda** dentro de la misma aplicación, lo que haría más fácil encontrar la respuesta a una duda o compartir las respuestas con otros usuarios del aula virtual.

Algo parecido pasaría con **enviar una actividad, publicar una nota y descargar la bibliografía**. Todas ellas pueden incorporarse de manera más o menos independiente y pueden utilizarse aunque el sistema no esté completo.

GESTIÓN ÁGIL DE PROYECTOS

Tal y como hemos visto anteriormente, hay muchos métodos de desarrollo de software que pueden calificarse como ágiles. En este apartado veremos algunos de los elementos habituales en dichos métodos de desarrollo[1].

- **Planificación**. El **ciclo de vida iterativo** consiste en dividir el proyecto en pequeños subproyectos (iteraciones) del mismo tamaño (normalmente semanas) y repetir el mismo proceso en cada iteración.
- **Historias de usuario**. Describen la funcionalidad del sistema de una manera que es válida tanto para los clientes del desarrollo (sin jerga técnica y sin entrar en detalles de implementación más allá de lo imprescindible) como para los desarrolladores del software (con un nivel de detalle suficiente y adecuado para trabajar con ella).
- ***Backlog* del producto**. Recoge los requisitos del sistema que desarrollar. Contiene la lista de todo el trabajo que se ha de llevar a cabo para completar el desarrollo del producto de software. En concreto, incluye todas las historias de usuario del producto.
- **Estimación**. Es difícil hacer una estimación precisa si no hemos analizado por completo el problema que tenemos que resolver, por lo que necesitamos encontrar una manera de trabajar con estimaciones que sean fáciles de refinar y fáciles de obtener.

Planificación

La ventaja del ciclo de vida iterativo es que nos permite medir cómo ha ido el proceso en las iteraciones pasadas y extrapolar esta medición a las iteraciones futuras. Es decir, facilita la **gestión empírica** (basada en observaciones) del proyecto.

> Dado que el ciclo de vida iterativo es útil si todas las iteraciones son comparables entre sí, lo más habitual es combinarlo con el desarrollo incremental.

[1] En este apartado no nos centraremos en ninguno en concreto. Más adelante dedicaremos un apartado al método Scrum.

En un proyecto que siga el ciclo de vida iterativo e incremental tendremos una o más entregas parciales (idealmente más de una), en las que se pondrá a disposición de los usuarios finales una versión con un subconjunto de la funcionalidad total.

El primer nivel de planificación que tenemos en un proyecto ágil es, pues, la **planificación del producto**, que nos permite responder a preguntas como «en febrero se podrán hacer pagos por la web» o «en abril tendremos la integración con SAP».

La planificación del producto contendrá una lista de las entregas que vamos a hacer, una estimación del número de iteraciones necesarias para cada entrega, así como una idea aproximada de la funcionalidad que tendremos en cada una de las entregas.

El siguiente nivel sería **planificar una entrega** en concreto.

> La planificación de una entrega suele tener un alcance de meses y permite responder a preguntas como «¿cuánta funcionalidad tendremos disponible el 8 de mayo?» o «¿cuándo tenemos previsto tener disponible la funcionalidad X?».

Hay dos maneras de plantear una entrega:

1. **Fijando una fecha** (en cuyo caso la planificación nos dirá cuánta funcionalidad creemos que habrá disponible en esa fecha).
2. **Fijando un alcance** (en cuyo caso la planificación nos dirá en qué fecha estará disponible la funcionalidad definida).

> Para responder a estas preguntas, en un entorno ágil, se utiliza la **velocidad**, que es una medida del volumen de trabajo completado a cada iteración.

La velocidad de un proyecto se **mide** en el pasado y, a partir de ahí, **estimamos** su valor futuro. Esto significa que no es un valor exacto sino, normalmente, un rango de valores que podemos calcular de diferentes maneras, como por ejemplo:

- Haciendo una media de las últimas *n* iteraciones.
- Haciendo una media ponderada de las últimas *n* iteraciones.
- Calculando un rango entre la media de los tres valores más bajos y la media de los tres más altos en las últimas *n* iteraciones.

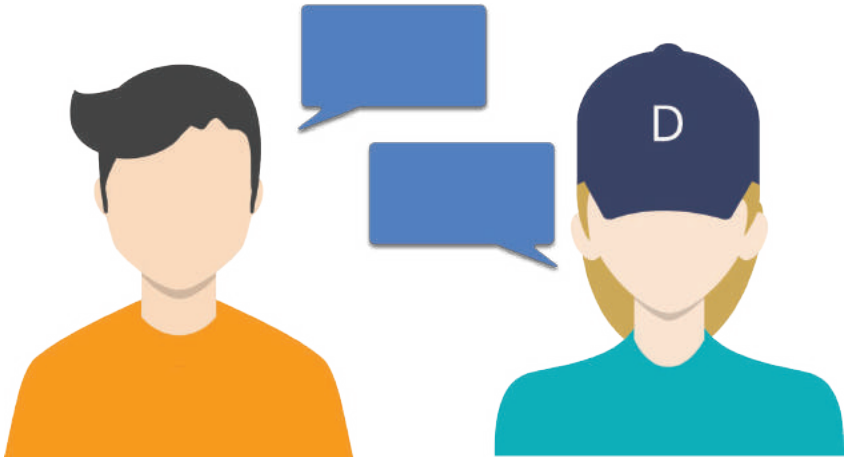
Lo importante es que vamos a poder ir observando cómo de preciso es nuestro cálculo de estimación de velocidad y que, a medida que avanza el proyecto, vamos a poder calcularla con mayor precisión.

Historias de usuario

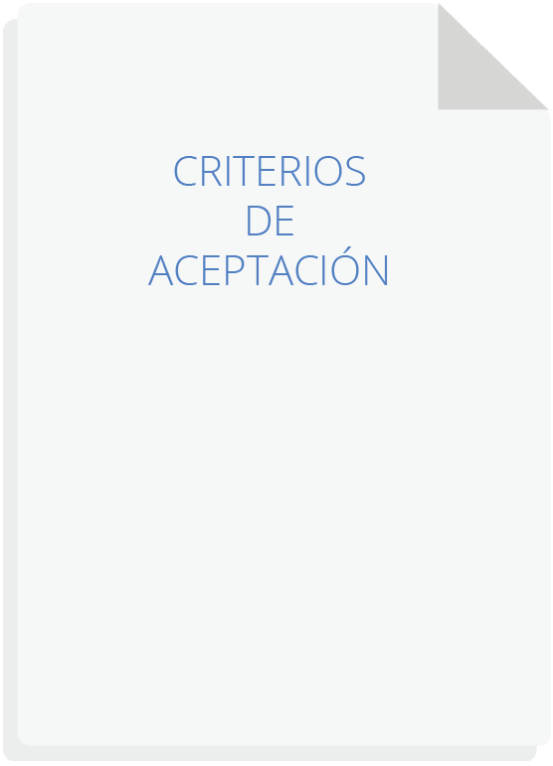
Una de las herramientas habituales para describir las funcionalidades que hemos de desarrollar son las llamadas **historias de usuario**.

Una historia de usuario describe un requisito del sistema mediante tres elementos:

1. Una **frase corta** que nos dice **quién** quiere esta funcionalidad, cuál es su **objetivo** (lo que quiere hacer con el sistema) y cuál es su **motivo** (por qué motivo quiere hacer esto con el sistema).



2. Una **conversación** entre los responsables del producto y los desarrolladores en la que se transmiten, de viva voz, los detalles de cómo debe ser la funcionalidad en cuestión. Esta conversación no queda registrada, pero sí quedan registradas sus conclusiones en forma de **criterios de aceptación**.



3. Una lista de **criterios de aceptación** que nos recuerdan qué condiciones debe satisfacer el sistema para que la funcionalidad se considere correctamente implementada.

«¿Qué pasa si un producto no tiene foto?» o «¿hay que poder ampliar la foto?». Una manera habitual de documentar estos criterios es en forma de prueba: «El sistema muestra una foto genérica si el producto no tiene foto», «El usuario puede hacer clic en la foto y se le muestra una versión ampliada».

Una historia de usuario que sea muy general puede descomponerse en subhistorias de usuario a medida que se acerca su implementación.

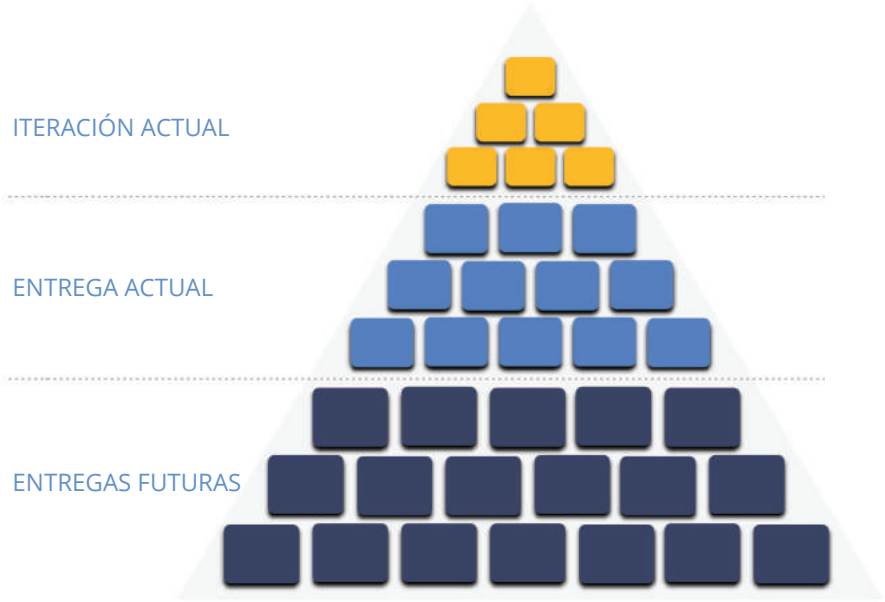
Historia de usuario general	Descomposición
Como usuario quiero poder cancelar una reserva.	Como usuario prémium quiero poder cancelar hasta el último minuto.
	Como usuario regular quiero poder cancelar hasta 24 horas antes del viaje.
	Como usuario quiero recibir un email que confirme que se ha cancelado la reserva.

A veces, cuando las historias de usuario son muy generales, se les llama **épicas** para destacar el hecho de que son historias que deberán descomponerse antes de ser abordadas en una iteración.

El *backlog* del producto

El ***backlog*** es una lista ordenada por prioridad (los responsables del producto son quienes establecen esta prioridad) y combina historias generales con historias más concretas.

Una manera de entender esto es mediante la figura del ***product backlog iceberg***:



Product Backlog Iceberg - Fuente: elaboración propia

Como ocurre con los icebergs, el equipo de desarrollo sólo ve la punta del *backlog* (las historias que están listas para ser implementadas) pero habrá una parte importante del *backlog* que queda más allá de la **iteración actual** y que consta de historias de usuario que estarán menos detalladas cuanto más lejana en el tiempo esté su implementación.

El *backlog* del producto no es fijo a lo largo del desarrollo, sino que se irá gestionando (añadiendo o quitando historias, cambiando las prioridades o detallando historias de usuario mediante nuevos criterios de aceptación o sub-historias). Este es uno de los motivos por los que las historias de usuario están menos detalladas cuanto más lejana en el tiempo está su implementación.

A lo largo del desarrollo (pero no necesariamente a cada iteración) se llevarán a cabo uno o más *story-writing workshops* en los que se generan nuevas historias de usuario para incorporar los requisitos que se hayan ido descubriendo sobre el sistema.

Estimación

Uno de los problemas que debemos resolver a la hora de gestionar un proyecto de forma ágil es el de la **estimación**. ¿Cuánto tiempo/esfuerzo vamos a necesitar para desarrollar una funcionalidad concreta?

Es difícil hacer una estimación precisa si no hemos analizado por completo el problema que tenemos que resolver, por lo que necesitamos encontrar una manera de trabajar con estimaciones que sean fáciles de refinar y fáciles de obtener.

> La solución que se suele aplicar en un proyecto ágil consiste en cambiar el problema: en lugar de estimar la **duración** (horas de esfuerzo) de una historia de usuario, estimamos su **tamaño** relativo al resto de las historias de usuario. Una vez estimado el tamaño, usamos un factor de conversión (la **velocidad**) para convertir el tamaño en duración.

Por lo tanto, para realizar la estimación de duración de un proyecto, descompondremos el problema en dos:

1. Calcular el tamaño de las historias de usuario

Para calcular el tamaño, una medida que suelen utilizar los equipos que se están iniciando en el desarrollo ágil son las **horas ideales**, que son las horas que cree el desarrollador que tardaría en implementar una funcionalidad si no estuviese trabajando en nada más, no tuviese bloqueos de ningún tipo y nadie le interrumpiese; condiciones que difícilmente se cumplirán pero que sirven para encontrar el tamaño relativo de las diferentes historias de usuario (si la historia A tardaremos lo mismo, el doble, la mitad o seis veces más que para la historia B).

El problema de las horas ideales es que es difícil justificar por qué un día no tiene 8 horas de trabajo «ideales» y al final crean cierta confusión, por lo que lo habitual es acabar usando los denominados **puntos de historia**.

2. Calcular la velocidad del equipo de desarrollo

La buena noticia es que el segundo de los problemas (la velocidad) lo podemos resolver de manea empírica si mantenemos estables las condiciones del desarrollo (mantenemos el mismo equipo, hacemos tipos de trabajo parecidos en todas las iteraciones, todas las iteraciones duran lo mismo, etc.), lo que encaja perfectamente con la idea de desarrollo iterativo.

Triangulación

Una técnica habitual para obtener la estimación en puntos de historia de una tarea es la **triangulación**, que consiste en determinar el valor de una historia comparándola con valores ya establecidos para otras historias.

Una consecuencia de estimar en puntos de historia es que, así como la diferencia entre 1 punto y 2 puntos es bastante importante (una es el doble de trabajo que la otra), la diferencia entre, por ejemplo, 20 puntos y 21 es menor y, por lo tanto, no aporta mucho sentido dedicar esfuerzo a discutir si son 20 o 21 puntos, ya que la diferencia cae dentro del margen de error de la propia estimación. Por este motivo se recomienda utilizar valores cuya distancia crezca progresivamente, como, por ejemplo, los de la serie de Fibonacci (1, 2, 3, 5, 8, 13, 21, etc.) redondeados (y añadiendo el 0 y el 1/2 si se considera necesario), de modo que podríamos utilizar 0, 1/2, 1, 2, 3, 5, 8, 13, 20, 40, 100. En cualquier caso, los últimos valores son indicativos de historias de usuario muy grandes que deberán descomponerse antes de entrar en una iteración y, por eso, lo habitual es que nos movamos en el rango entre 1 y 10.

Planning póker

Uno de los problemas tradicionales con las estimaciones es que es difícil conseguir que el equipo de desarrollo asuma como propia una estimación que viene del jefe de proyectos o del cliente y que no ha tenido en cuenta su opinión. Para ello, una técnica que se puede aplicar es la del ***planning* póker**, que funciona de la siguiente manera:

1. Cada estimador recibe un conjunto de cartas, cada una con un valor diferente escrito en ella (típicamente, los valores que acabamos de ver: 0, 1/2, 1, 2, 3, 5, 8, etc.).
2. El cliente o el responsable del producto lee una historia de usuario y se discute brevemente entre los presentes para asegurarnos de que todo el mundo entiende cuál es el alcance de la tarea.
3. Cada estimador selecciona una carta con un valor, pero no la muestra al resto (así evitamos condicionar la estimación del resto de los estimadores).
4. Cuando todos los estimadores han elegido una carta se muestran para que todos vean los valores.
5. Se discuten las diferencias (especialmente si son muy grandes) para asegurarnos de que todo el mundo está teniendo en cuenta la misma información y que nadie está pasando nada por alto.
6. Repetimos el proceso hasta que los resultados converjan.
7. Pasamos a la siguiente historia de usuario.

SCRUM

Scrum es, sin lugar a dudas, el método ágil de desarrollo más popular.

Su popularidad se basa, en gran parte, en que es un método muy ligero y fácil de entender (lo que no quiere decir que sea fácil de implantar). Además, a diferencia de eXtreme Programming (XP), con el que tiene muchos elementos en común, no obliga al equipo de desarrollo a implantar ninguna práctica concreta a nivel de programación, sino que deja que cada equipo de desarrollo encuentre el conjunto de prácticas que le funciona (en este sentido, es menos prescriptivo que XP).

Scrum en sí no se define como un proceso sino como un **marco de trabajo**, ya que la personalización del proceso es una parte fundamental de la aplicación de Scrum.

Scrum emplea un enfoque iterativo e incremental para el proceso de desarrollo y se basa en la teoría de control de procesos empírica, soportada en tres pilares básicos:

- **Transparencia**. Los aspectos significativos del proceso deben ser visibles para aquellos que son responsables del resultado.
- **Inspección**. Los usuarios de Scrum deben inspeccionar frecuentemente los artefactos generados y el progreso para detectar variaciones indeseadas.
- **Adaptación**. Si un inspector detecta una desviación y determina que el producto resultante será inaceptable, el proceso debe ajustarse para corregir dicha desviación.

Scrum prescribe cuatro eventos formales para la inspección y adaptación:

- Planificación del sprint (*sprint planning*)
- Scrum diario (*daily Scrum*)
- Revisión del sprint (*sprint review*)
- Retrospectiva del sprint (*sprint retrospective*)

El equipo Scrum

A nivel organizativo, un equipo Scrum consta de:

1. Dueño de producto (*product owner*)

Según la guía de Scrum:

> *«El dueño de producto es el responsable de maximizar el valor del producto y el trabajo del equipo de desarrollo».*

Es la persona de referencia para el equipo de desarrollo a la hora de tomar cualquier decisión sobre el producto y los aísla del resto de la organización y/o clientes.

Aunque pueda parecer sencillo, es un trabajo muy difícil, ya que debe dar una visión consistente que refleje todos los intereses y las prioridades de los diferentes *stakeholders* del proyecto. Esto lo hará mediante la gestión del *backlog* de producto, añadiendo o quitando historias, así como cambiando la priorización de estas.

No solo es responsable del contenido del *backlog*, sino que también es responsable de asegurar que los miembros del equipo de desarrollo entienden los elementos del backlog correctamente.

> Es importante que este rol recaiga sobre una sola persona, ya que, en caso de ser un rol compartido, dejaría de ser efectivo. Del mismo modo, es importante que este rol sea respetado por la organización y que, por ejemplo, nadie pueda modificar el backlog del producto sin su consentimiento.

2. Equipo de desarrollo

Un equipo de desarrollo formado por varios desarrolladores (idealmente entre 5 y 7). Según la guía de Scrum:

> *«El **equipo de desarrollo** consiste en los profesionales que realizan el trabajo de entregar un incremento de producto "Terminado" que potencialmente se pueda poner en producción al final de cada sprint».*

Un equipo de desarrollo tiene las siguientes características:

- **Autoorganizado**: nadie (ni siquiera el *Scrum master*) indica al equipo cómo convertir un elemento del *backlog* en un incremento de funcionalidad.
- **Multifuncional**: aunque no se distinguen roles individuales, como equipo cuenta con todas las habilidades necesarias para crear un incremento de producto.
- **No reconoce títulos**: todos los miembros del equipo de desarrollo son desarrolladores, independientemente del trabajo que realicen. No existen analistas, programadores, arquitectos, diseñadores, etc. Solo existen desarrolladores.
- **No reconoce subequipos**: solo hay un equipo con independencia de los dominios particulares que requieran tenerse en cuenta. No existe equipo de pruebas, de negocio, etc.
- Los miembros individuales del equipo pueden tener habilidades especializadas y áreas en las que estén más enfocados, pero la responsabilidad recae en el equipo de desarrollo como un todo.

3. *Scrum* master

Según la guía de Scrum:

> *«El Scrum master es el responsable de asegurar que Scrum se entienda y se adopte».*

Es la persona que vela por que Scrum se aplique correctamente y se den las condiciones que se tienen que dar a nivel de organización para que sea efectivo.

Por ejemplo, ayuda al dueño del producto en la elaboración del *backlog* y en la planificación del proyecto, al equipo eliminando los impedimentos que se vayan encontrando o a la organización ayudando a sus miembros a entender y llevar a cabo Scrum y el proceso empírico.

Eventos de Scrum

Scrum define una serie de **eventos** que se repiten a intervalos regulares. Todos los eventos tienen una longitud máxima definida *a priori* y, a excepción del *sprint* (la iteración), todos pueden acabar antes de dicha longitud máxima si se consigue el objetivo que se buscaba con dicho evento.

1. *Sprint*

El ***sprint*** es un bloque de tiempo (normalmente entre dos y cuatro semanas) durante el cual se desarrolla un incremento de producto terminado utilizable y potencialmente desplegable (es decir, que podría ponerse a disposición de los usuarios finales de manera inmediata). Esto no significa que se tenga que desplegar necesariamente, pero sí que, si se decide que se quiere desplegar, no es necesario dedicar más tiempo a pruebas o a «refinar» el producto.

Un *sprint* consta de:

1. Una planificación del sprint (*sprint planning*)
2. Los Scrum diarios (*daily Scrum*)
3. El trabajo de desarrollo
4. La revisión del sprint (*sprint review*)
5. La retrospectiva del sprint (*sprint retrospective*)

> Cada *sprint* es como un miniproyecto autocontenido que pasa por todas las fases del desarrollo (análisis, diseño, implementación, pruebas, etc.) y que tiene un objetivo concreto (*sprint goal*).

Si queremos que la velocidad medida sea útil, es muy importante que todos los *sprints* tengan la misma longitud (es decir, es como un miniproyecto de duración fija y alcance variable).

Una característica muy importante del sprint es que, una vez decidido y planificado el objetivo del *sprint*, no se pueden añadir tareas que no sean para conseguir dicho objetivo. De esta manera, el dueño de producto tiene flexibilidad para repriorizar el *backlog* (priorizando historias de usuario para el *sprint* siguiente), pero el equipo mantiene una ventana de estabilidad que le permite trabajar para conseguir un objetivo.

2. Planificación del sprint (*sprint planning*)

La **planificación del *sprint*** tiene en cuenta un máximo de duración de 8 horas para un *sprint* de un mes (menos si el *sprint* es más corto) y responde a las siguientes preguntas:

- ¿Qué puede entregarse en el incremento de producto resultante del *sprint* que comienza?
- ¿Cómo se conseguirá hacer el trabajo necesario para entregar el incremento?

Para ello, habrá que establecer el objetivo del *sprint* y el conjunto de historias de usuario necesarias para cumplir dicho objetivo. Las historias de usuario se analizarán en detalle y se determinarán las tareas que se han de llevar a cabo para implementar cada una de ellas. Dichas tareas deberán poderse completar en un día o menos (si no, deben descomponerse en tareas más sencillas).

El equipo de desarrollo es quien decide cuántas historias de usuario pueden completar en este *sprint* en concreto y, bajo este criterio, puede **negociar** con el dueño del producto para ampliar o reducir la lista de historias de usuario que implementar para este *sprint* en concreto.

3. Objetivo del *sprint*

Tal y como hemos visto, el **objetivo del *sprint*** es una meta establecida que sirve como guía para el equipo de desarrollo y para que todo el equipo trabaje alineado hacia un mismo objetivo y, al mismo tiempo, para poder renegociar el alcance de un *sprint* si el equipo detecta que no es posible completar todas las historias de usuario comprometidas o que es posible implementar más historias de las comprometidas inicialmente.

4. Scrum diario (*daily Scrum*)

El **Scrum diario** es una reunión de 15 minutos para que el equipo de desarrollo sincronice sus actividades y planifique las siguientes 24 horas.

Para ello, se inspecciona el trabajo realizado durante el día anterior y se hace una previsión del trabajo que se desarrollará durante el día siguiente. En concreto, cada miembro del equipo de desarrollo explica:

- ¿Qué hice ayer que ayudó al equipo de desarrollo a lograr el objetivo del *sprint*?
- ¿Qué haré hoy para ayudar al equipo de desarrollo a lograr el objetivo del *sprint*?
- ¿Veo algún impedimento que evite que el equipo de desarrollo o yo logremos el objetivo del *sprint*?

Los Scrum diarios mejoran la comunicación y reducen la necesidad de realizar otras reuniones. También ayudan a mejorar el conocimiento que los miembros del equipo tienen del trabajo que están desarrollando los otros miembros, así como a detectar oportunidades de colaboración entre miembros del equipo (por ejemplo, si un miembro explica que tiene un problema a la hora de utilizar una herramienta concreta, otro miembro del equipo que la conozca bien podría ofrecerse a ayudarle en esa tarea).

5. Revisión de sprint (*sprint review*)

La **revisión de *sprint*** es un evento de máximo cuatro horas de duración en el que se explica qué elementos del *backlog* se han terminado (y cuáles no), se muestra el funcionamiento del incremento (es decir, se muestra el software funcionando) y se revisa el *backlog* y la planificación para preparar el siguiente *sprint*.

6. Retrospectiva de sprint (*sprint retrospective*)

La **retrospectiva** es una reunión limitada a tres horas máximo en la que se inspecciona cómo fue el último *sprint* en cuanto a personas, relaciones, procesos y herramientas, se identifican elementos que han ido bien, así como posibles mejoras, y se crea un plan para implementar dichas mejoras.

Al final de la retrospectiva el equipo tendrá una lista de mejoras que implementar durante el siguiente *sprint*. De este modo, el equipo va adaptando el proceso y mejorándolo continuamente.

Artefactos de Scrum

1. La lista de producto (product backlog)

La **lista de producto** es una lista ordenada de todo lo que podría ser necesario en el producto. El dueño del producto es la persona responsable de gestionarla y quien tiene la última palabra en cuanto a su contenido (qué elementos forman parte de la lista y cuáles no) y su ordenación (qué elementos son más prioritarios).

La lista de producto evoluciona a medida que evoluciona el desarrollo y que cambian los requisitos de negocio, el mercado o, simplemente, a medida que vamos utilizando el producto y vamos identificando mejor las necesidades reales.

2. La lista de pendientes del *sprint* (*sprint backlog*)

Esta otra lista contiene los elementos del *backlog* de producto que han sido elegidos para un *sprint* en concreto.

A diferencia del *product backlog*, el *sprint backlog* lo elabora el equipo de desarrollo y solo el equipo de desarrollo puede añadir o quitar elementos. Eso sí, siempre a partir del *product backlog* y la negociación con el *product owner* para determinar qué elementos son necesarios para conseguir el objetivo del *sprint*.

En este caso, los elementos de la lista están analizados y descompuestos a un nivel de detalle tal que el avance pueda ser visible en el *daily Scrum* (normalmente, tareas que puedan completarse en menos de 8 horas laborables).

3. Incremento

El **incremento** es la suma de todos los elementos del *product backlog* que han sido completados durante un *sprint*. Para poder determinar objetivamente cuándo un elemento está terminado, se establecen unos criterios estándares (***definition of done***) que debe cumplir cualquier elemento del *product backlog* para poderse incluir en el incremento.

De cara a la definición de la definition of done, es importante tener en cuenta que una vez terminado un elemento del backlog ya no es necesario realizar ningún trabajo adicional para desplegarlo y ponerlo en manos de los usuarios finales (por ejemplo, no hay que hacer QA, ni documentación adicional).

KANBAN

Kanban no es en realidad un proceso ni un método de desarrollo sino una herramienta para gestionar el flujo de trabajo en un proceso dado.

Aplicación

La principal ventaja de Kanban es que no requiere que cambiemos nuestro proceso de trabajo, sino que nos ayuda a visualizar mejor el estado del proyecto y a responder a preguntas como:

- ¿Cuál es la situación actual del producto?
- ¿Cuándo estará disponible una cierta funcionalidad?
- ¿Quién está trabajando en una tarea concreta?
- ¿Cuál es la siguiente tarea con la que debería ponerme?

1. Documentar el flujo de trabajo

Para ponerlo en práctica, el primer paso será **documentar el flujo de trabajo** de manera que este sea lo más lineal posible.

Por ejemplo, en el caso de desarrollo de software, podríamos identificar los siguientes pasos: requisitos, análisis, implementación, pruebas, entrega; mientras que si estamos gestionando un proceso de ventas podríamos identificar: generar *lead,* presentar el producto, elaborar propuesta, negociar contrato, firmar contrato, seguimiento.

> Dado que lo que se persigue con Kanban es establecer un mecanismo de mejora continua, es importante que se documente el proceso real y no una versión idealizada de este.

De este modo, las ineficiencias y los cuellos de botella que iremos detectando serán reales y no ficticios.

El siguiente paso es establecer una **limitación de trabajo en progreso** para cada uno de los estados que hemos detectado. Por ejemplo, sabemos que, si un programador está trabajando en tres tareas al mismo tiempo, los cambios de contexto le restarán muchísima productividad, por lo que deberíamos establecer el límite de tareas en estado «Implementación» a una o dos como máximo por desarrollador.

A medida que estos límites vayan bajando, iremos descubriendo ineficiencias que deberemos corregir, por lo que es importante no pasarse por arriba, ya que en ese caso será difícil encontrar dónde están exactamente los problemas, ni por abajo, ya que entonces podríamos destapar demasiados problemas al mismo tiempo y esto podría generar resistencias a la hora de implantar la herramienta.

2. Visualizar el flujo de trabajo

Una vez hemos documentado el flujo de trabajo, podemos proceder a la elaboración del **panel Kanban**:

Panel Kanban - Fuente: elaboración propia a partir de https://commons.wikimedia.org/wiki/File:Kanban_board_example.jpg

En el panel Kanban utilizamos columnas para los diferentes estados del proceso y, en cada columna, apuntamos el límite de trabajo en progreso. A continuación, ponemos las tareas en el panel ordenadas por prioridad (las más prioritarias más arriba), de manera que podamos ver rápidamente en qué estado está cada tarea y cuáles serán las siguientes tareas que deberán acometerse.

Podemos aprovechar el panel para guardar más información sobre cada tarea, como por ejemplo qué persona es la responsable de esta tarea, la fecha de creación, la fecha límite, los criterios de aceptación (sería como la *definition of done de Scrum*), etc.

3. Poner en práctica el panel

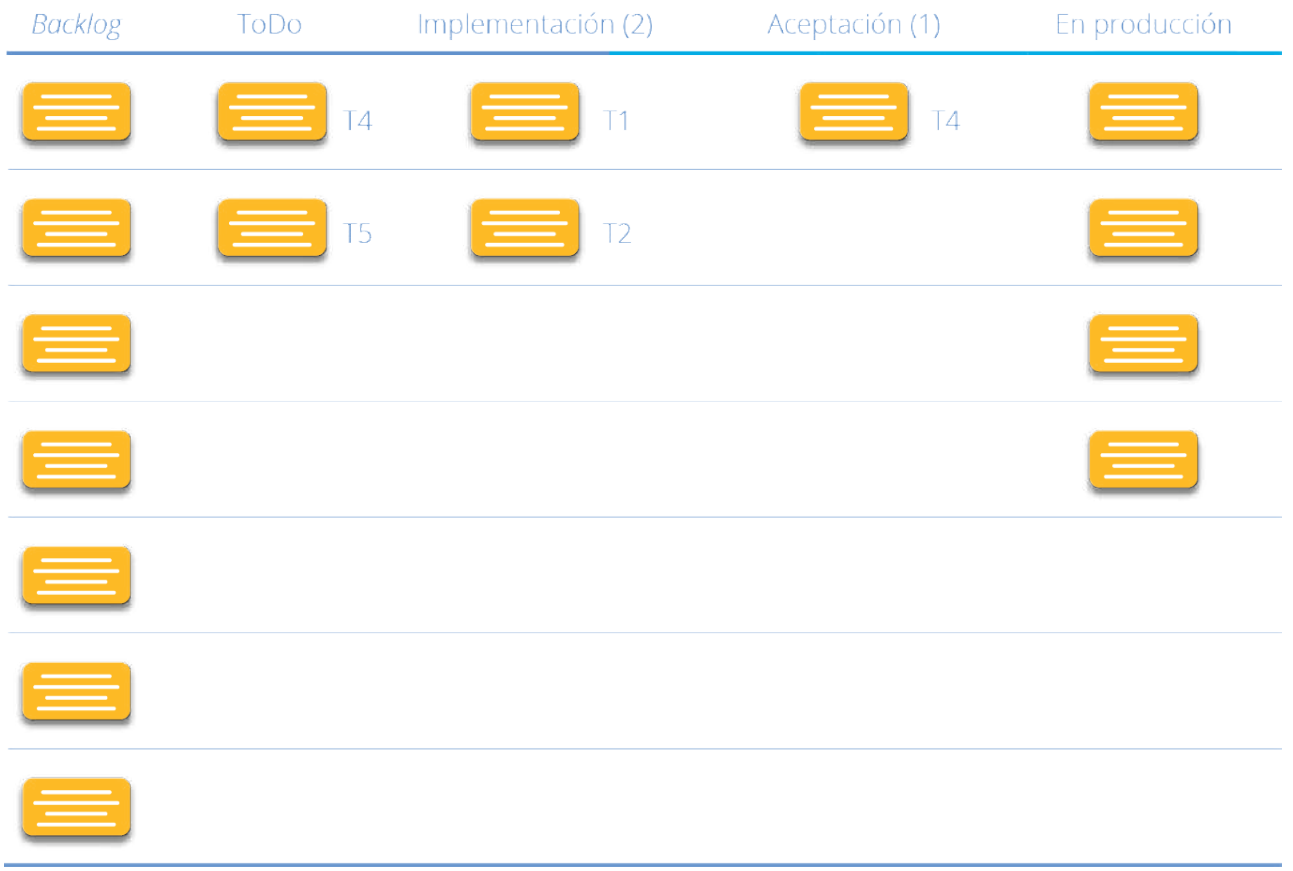
Kanban es un sistema pull.

El concepto ***pull*** viene de la fabricación de productos en plantas industriales (por ejemplo, de coches), donde solo se producen piezas si el siguiente paso de la cadena las demanda. El sistema alternativo es el sistema ***push***, en el que cada paso de la cadena crea piezas a un ritmo concreto y almacena el *stock* sobrante.

En el caso del panel del ejemplo anterior, no pondremos tareas en la columna «En prueba» (límite 1 y ya tiene 1 tarea) hasta que la tarea actual pase al estado siguiente («Hecho»), en cuyo caso moveremos una de las dos tareas que están en estado «Revisión por pares», lo que podría provocar que se moviese una de las tareas en estado «En progreso» a «Revisión por pares» y así hasta llegar al inicio del proceso.

En general, cuando alguien está disponible para trabajar en una tarea, irá al panel y buscará una tarea que esté lista para pasar al siguiente estado, la moverá a la columna correspondiente y trabajará en ella hasta que finalice este estado, momento en el que esperará a que otra persona haga lo mismo con el siguiente estado.

Supongamos ahora que tenemos el siguiente panel:



Fuente: elaboración propia

Supongamos, también, que somos un programador y hemos finalizado las tareas T1 y T2. *A priori*, parecería que lo lógico es ponernos a trabajar en T4, ya que es la siguiente tarea en la columna anterior. Sin embargo, tenemos un problema: hemos alcanzado el límite de trabajo en progreso de nuestra columna actual (2). Esto significa que acabamos de encontrar un cuello de botella: no somos capaces de aceptar tareas al ritmo que las implementamos.

En este punto, acabamos de conseguir lo que se buscaba con la aplicación de Kanban: detectar un cuello de botella en el proceso actual.

4. Inspección y adaptación

Una vez detectado un cuello de botella, se debe trabajar en su eliminación:

- Por un lado, las personas que han quedado bloqueadas ayudan en la medida de lo posible a hacer avanzar la tarea que las ha bloqueado para eliminar este caso concreto del cuello de botella.

- Por otro lado, se deberá buscar la causa raíz del problema y elaborar una propuesta de mejora que evite que se vuelva a producir.

Para saber si las propuestas de mejora son efectivas, necesitamos alguna métrica que nos dé una idea de lo bien que esté funcionando nuestro proceso. En Kanban se suelen usar dos métricas:

1. El **tiempo de espera** (*lead time*) es el tiempo que pasa desde que alguien pide que se haga una tarea hasta que esta se completa (ha pasado por todos los estados).

2. El **tiempo de ciclo** (*cycle time*) es el tiempo que tarda una persona en completar una tarea una vez la ha comenzado.

Con estas dos métricas podremos responder a preguntas como ¿cuánto tardará esta funcionalidad en estar disponible? También nos sirven para ver si las tareas fluyen adecuadamente o si pasan demasiado tiempo esperando para pasar al siguiente estado.

RESUMEN

Hemos visto que el desarrollo ágil de software no es una metodología sino un conjunto de principios que se pueden seguir a través de diferentes metodologías de desarrollo de software. Para ser ágiles, debemos ser pragmáticos y evitar dedicar esfuerzo a tareas que dan falsa sensación de control del proceso o de calidad del trabajo.

Debemos priorizar la relación entre personas, el obtener un producto tangible y utilizable cuanto antes, la colaboración con el cliente y el saber adaptarnos a los cambios. Para ello, desarrollaremos el software de manera incremental y, en según qué circunstancias, de manera iterativa.

Una herramienta muy útil para gestionar los requisitos del software son las **historias de usuario**, que recogeremos en un *backlog*, que incluirá historias definidas a diferentes niveles de detalle. La manera habitual de gestionar un proyecto con historias de usuario es estimando el tamaño relativo de estas y calculando la velocidad del equipo de desarrollo para determinar o bien la fecha de entrega del backlog completo, o bien el alcance del desarrollo hasta una fecha concreta.

Scrum es, probablemente, el método ágil de desarrollo más popular. Consiste en un marco de trabajo mediante el cual cada equipo tiene que encontrar su propio proceso de desarrollo. El marco de trabajo es muy simple y se basa en un ciclo de desarrollo iterativo e incremental. El **dueño del producto** es el responsable de tomar decisiones a nivel de producto, mientras que el equipo de desarrollo es el responsable de transformar las historias de usuario en funcionalidad implementada. El ***Scrum master*** es el responsable de que se aplique Scrum adecuadamente.

También hemos visto que **Kanban** es una herramienta que nos puede ayudar a agilizar una organización sin hacer un cambio radical en sus procesos. Consiste en documentar el flujo de trabajo, visualizarlo y buscar los cuellos de botella en el proceso para ir mejorándolo de manera continua.

Una manera de resumir el desarrollo ágil es usando lo que Alistair Cockburn (uno de los firmantes del manifiesto) denomina el corazón de Agile:

- ***Deliver***: entregar productos tangibles que proporcionen valor.
- ***Reflect***: reflexionar sobre el proceso, identificando puntos positivos y negativos.
- ***Improve***: mejorar el proceso de manera gradual sobre la base del resultado de la reflexión.
- ***Collaborate***: colaborar entre todos los implicados para conseguir el objetivo final: un software que ayude a sus usuarios a cumplir sus objetivos.

CONCEPTOS CLAVE

- ***Backlog*** es la lista de requisitos que se deben implementar para considerar el sistema completo a nivel funcional.

- **Ciclo de vida iterativo** consiste en organizar un proyecto de desarrollo de software en pequeñas iteraciones de duración regular que funcionan como miniproyectos.

- **Contrato ágil** es un contrato que recoge un modelo de colaboración entre cliente y desarrolladores más flexible que el modelo tradicional, en el que se cerraba alcance, presupuesto y plazo de entrega.

- ***Definition of done*** es la lista de criterios que se aplicarán para determinar objetivamente que un elemento del *backlog* está completado.

- **Desarrollo incremental** consiste en desarrollar el sistema mediante pequeños incrementos de funcionalidad de modo que, en todo momento, se tiene un sistema utilizable, aunque sea incompleto.

- **Dueño de producto (product owner)** es el responsable de maximizar el valor del producto y el trabajo del equipo de desarrollo.

- **El manifiesto ágil** es un conjunto de principios que definen los elementos básicos para facilitar el éxito de un proyecto de desarrollo de software.

- **Historias de usuario** describen la funcionalidad del sistema de una manera que es válida tanto para los clientes como para los desarrolladores.

- **Inspección y adaptación** son las herramientas básicas para la mejora continua: consiste en inspeccionar el proceso actual (midiendo el rendimiento en el pasado) y adaptar el proceso de desarrollo para mejorarlo.

- **Kanban** es una herramienta que facilita la detección de cuellos de botella en un proceso.

- **Panel Kanban** es una herramienta que facilita la visualización del flujo de trabajo y del estado de las tareas dentro de este.

- ***Planning* póker** es una técnica de estimación colaborativa.

- **Punto de historia** es una unidad abstracta que nos da una idea del esfuerzo necesario para completar una historia de usuario en relación con el resto de las historias del *backlog*.

- **Retrospectiva de *sprint*** es el último evento del *sprint*; en él se revisa cómo ha ido el *sprint*, se identifican elementos que han ido bien y se detectan oportunidades de mejora.

- **Revisión de *sprint*** es un evento que tiene lugar al final del *sprint* y en el que se muestra el incremento de producto que se ha desarrollado.

- **Scrum diario** es una reunión de 15 minutos para que el equipo de desarrollo sincronice sus actividades y planifique las siguientes 24 horas.

- **Scrum** es un marco de trabajo para la gestión de proyectos de manera ágil.

- ***Scrum master*** es el responsable de asegurar que Scrum se entienda y se adopte.

- ***Sprint*** es el nombre que se da en Scrum a una iteración.

- **Velocidad** es la medida del volumen de trabajo completado en una iteración por un equipo de desarrollo. Sirve para proyectar el rendimiento futuro a partir del tamaño de los elementos del *backlog*.

BIBLIOGRAFÍA

Beck, K.; Beedle, M.; van Bennekum, A.; Cockburn, A.; Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; Kern, J.; Marick, B.; Martin, R. C.; Mellor, S.; Schwaber, K.; Sutherland, J.; Thomas, D. (2001). *Manifesto for Agile Software Development*. [en línea]. http://agilemanifesto.org/

Cohn, M. (2006). *Agile Estimating and Planning*. [en línea]. https://www.mountaingoatsoftware.com/presentations/agile-estimating-and-planning

Cohn, M. (2013). *Agile Planning and Project Management*. [en línea]. https://www.mountaingoatsoftware.com/presentations/agile-planning-and-project-management

Klipp, P. *Getting Started with Kanban*. [en línea]. https://kanbanery.com/ebook/GettingStartedWithKanban.pdf

Schwaber, K.; Sutherland, J. (2014). *La Guía de Scrum*. [en línea]. http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-Spanish.pdf