METODOLOGIAS AGILES

Eric Gustavo Coronel Castillo
INSTRUCTOR

Lima – Perú
2021

INDICE

Capítulo 1
ENTENDIENDO UN PROYECTO AGIL

INTRODUCCIÓN

Desarrollo Tradicional

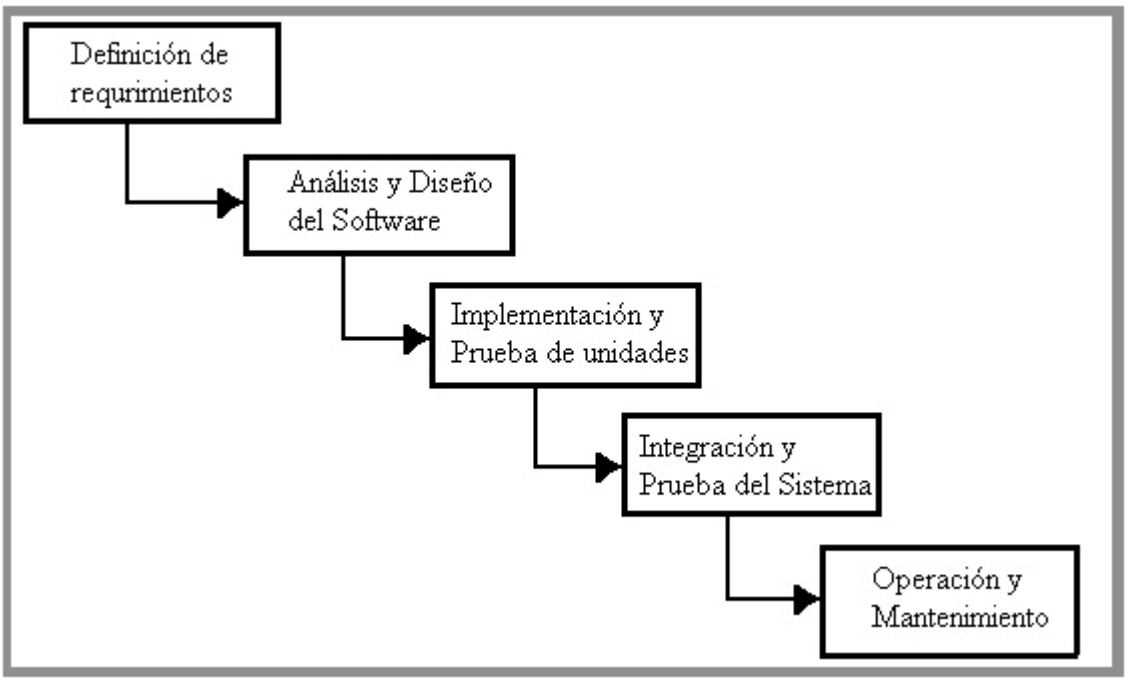


Figura 1 La predictibilidad, ciclo de vida en cascada o desarrollo tradicional

En su nacimiento, la gestión de proyectos software intentó imitar la gestión de proyectos de otras disciplinas, como la arquitectura, las industria o la ingeniería civil, hasta el punto de heredar y adaptar al mundo del software muchos de sus roles (p.e. arquitectos software) y tipos de organizaciones (p.e. fábricas de software).

Hoy en día una de las prácticas más discutidas y polémicas de las que se han querido heredar desde otras disciplinas es la llamada predictibilidad, también conocida como gestión de proyectos dirigida por la planificación, desarrollo tradicional o incluso también conocida como desarrollo pesado.

La predictibilidad se basa en dividir un proyecto en fases, por ejemplo, de manera simplificada, “requisitos”, “diseño” y “construcción”, y que cada una de estas fases no comience hasta que termine con éxito la anterior.

Se le llama predictibilidad porque cada fase intenta predecir lo que pasará en la siguiente; por ejemplo, la fase de diseño intenta predecir qué pasará en la programación, y esos diseños intentarán ser muy precisos y detallados, para ser cumplidos sin variación por los programadores.

Además, en este tipo de gestión, cada una de estas fases se realiza una única vez (no hay dos fases de requisitos). Y las fases están claramente diferenciadas (en teoría, está claro

cuándo termina el diseño, comienza la programación), hasta el punto de tener profesionales claramente diferenciados y especializados en cada una de ellas: “analistas de requisitos”, “arquitectos de diseño software”, programadores, personas para pruebas, etc.

Normalmente cada fase concluye con un entregable documental que sirve de entrada a la siguiente fase, la “especificación de requisitos software” es la entrada al diseño, el “documento de diseño” la entrada a la construcción, etc.

La gestión de proyectos predictiva es típica en disciplinas como la arquitectura. Y desde sus orígenes, la ingeniería del software intentó perseverantemente emular a las ingenierías clásicas. Tener una fase de diseño muy claramente separada de la programación (hasta el punto de intentar tener una organización cliente que detalle los diseños y otra organización, normalmente llamada fábrica de software, que los implemente). Que la programación no comenzase hasta que terminase el diseño. Que el diseño concluya con unos planos precisos que guíen totalmente la construcción. Que una vez que se hace un diseño éste no se modifique; de hecho, notaciones como UML (Unified Modeling Language es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema) se concibieron para construir los “planos detallados” del software.

Al anterior tipo de gestión de proyectos predictiva, en el mundo del software se le conoce como ciclo de vida en cascada, tal como se ilustra en Figura 1.

Ágil vs. Tradicional

Construir software no es como construir coches o casas

En software, la experiencia nos dice que es muy difícil especificar los requisitos en una única y primera fase. Por la complejidad de muchas de las reglas de negocio que automatizamos cuando construimos software, es muy difícil saber qué software se quiere hasta que se trabaja en su implementación y se ven las primeras versiones o prototipos.

También es muy difícil documentar de una única vez, a la primera, antes de la codificación, un diseño que especifique de manera realista y sin variación todas las cuestiones a implementar en la programación.

Las ingenierías clásicas o la arquitectura necesitan seguir este tipo de ciclos de vida en cascada o predictivos porque precisan mucho de un diseño previo a la construcción, exhaustivo e inamovible: disponer de los planos del arquitecto siempre antes de empezar el edificio. Nadie se imagina que una vez realizados los cimientos de un edificio se vuelva a rediseñar el plano y se cambie lo ya construido.

Además, los planos para construir son precisos y pocas veces varían, ya que la mayoría de los diseños de las ingenierías clásicas, arquitecturas, etc., pueden hacer un mayor uso de las matemáticas o la física. En software no es así. Y aunque se pretenda emular ese modo de fabricación, en software no funciona bien, y debemos tener muy claro que es casi imposible cerrar un diseño a la primera para pasarlo a programación sin tener que modificarlo posteriormente.

Por lo general, realizar un cambio en el producto final que construyen las ingenierías clásicas o la arquitectura es muy costoso. Cambiar, por ejemplo, la posición de una columna en un edificio o realizar modificaciones a la estructura de un puente ya construido tiene un alto coste. Y de ahí que la arquitectura o las ingenierías clásicas pretendan lograr a toda costa diseños o planos de un alto nivel de detalle, para que una vez que comience la fase de construcción no tengan que ser modificados. Además, normalmente, en la arquitectura o en las ingenierías clásicas los costes de construir son muy elevados en comparación con los de diseñar. El coste del equipo de diseñadores es sustancialmente inferior al de la realización de la obra, del puente, edificio, etc.

La anterior relación de costes no se comporta igual en el caso del software. Por un lado, el software, por su naturaleza (y si se construye mínimamente bien), es más fácil de modificar. Cambiar líneas de código tiene menos impacto que cambiar los pilares de un edificio ya construido. De ahí que existan numerosas propuestas que recomiendan construir rápido una versión software y modificarla evolutivamente (la técnica de la refactorización trabaja sobre esta idea). En software no existe esa división tan clara entre los costes del diseño y los de la construcción.

También en las ingenierías clásicas o la arquitectura los roles y especialización necesaria en cada fase son diferentes. Los planos o diseños los realizan arquitectos que no suelen participar en la fase de construcción. La construcción tiene poco componente intelectual y mucho manual, al contrario que el diseño. Y todo apoya a que existan dos actividades claramente diferenciadas: el diseño y la construcción.

En nuestro caso, el producto final, el software, tiene diferencias muy sustanciales con estos productos físicos. Estas diferencias hacen que el proceso de construcción sea diferente. Durante muchos años, quizás por la juventud de la ingeniería del software, se han obviado estas diferencias, e incluso se han intentado encontrar metodologías que imitasen y replicasen los procesos de construcción tradicional al software. Ejemplo de ello son las primeras versiones y usos de lenguajes de diseño como UML, o metodologías como RUP.

Sin embargo, en muchas ocasiones, estos intentos de emular la construcción de software a productos físicos han creado importantes problemas y algunos de los mayores errores a la hora de gestionar proyectos software.

Diferenciar el cómo se construye software del cómo se construyen los productos físicos es uno de los pilares de los métodos ágiles (M. Fowler, 2005)1. De hecho, es un tema del que se ha escrito mucho. Y también se ha debatido bastante, desde hace muchos años, con posturas a favor y en contra.

Y es que, en software, es frecuente que diseño y construcción muchas veces se solapen, y por ello se recomiende construir por iteraciones, por partes, y el uso de prototipos incrementales.

[1] Fowler, M. (2005). The new methodology.

Ciclo de vida iterativo e incremental

Es curioso ver como el concepto "**ciclo de vida**", una de las piezas más fundamentales, y transcendentales, de la gestión de un proyecto software produce tanta confusión.

En parte, tampoco es de extrañar, debido a que no existe una única terminología al respecto, existen muchas definiciones, conceptos confusos, etc. Por eso vamos a aclarar los distintos tipos de ciclos de vida que hay.

Cascada

Las fases del ciclo de vida (requisitos, análisis, diseño, etc.) se realizan (en teoría) de manera lineal, una única vez, y el inicio de una fase no comienza hasta que termina la fase anterior.

Su naturaleza es lineal, típica de la construcción de productos físicos y su principal problema viene de que no deja claro cómo responder cuándo el resultado de una fase no es el esperado.

El ciclo de vida más criticado en los últimos años. En muchos proyectos su implantación ha sido un fracaso, mientras que hay otros proyectos que trabajan perfectamente de esta manera.

El ciclo de vida incremental

Cada iteración (una iteración es un periodo de tiempo, no confundir con el ciclo de vida iterativo, que veremos luego, siendo este punto confuso, por las definiciones) contiene las fases del ciclo en cascada estándar, pero cada iteración trabaja sobre un sub conjunto de funcionalidad. La entrega total del proyecto se divide en subsistemas priorizados.

Desarrollar por partes el producto software, para después integrarlas a medida que se completan. Un ejemplo de un desarrollo puramente incremental puede ser la agregación de módulos en diferentes fases. El agregar cada vez más funcionalidad al sistema.

El ciclo de vida iterativo

En cada ciclo, iteración, se revisa y mejora el producto. Un ejemplo de desarrollo iterativo es aquel basado en refactorizaciones, en el que cada ciclo mejora más la calidad del producto.

Es importante señalar que este ciclo no implica añadir funcionalidades en el producto, pero si la revisión y la mejora.

Iterativo e incremental

Incremental = añadir, iterativo = retrabajo

Se va liberando partes del producto (prototipos) periódicamente, en cada iteración, y cada nueva versión, normalmente, aumenta la funcionalidad y mejora en calidad respecto a la anterior.

Además, el ciclo de vida iterativo e incremental es una de las bases de un proyecto ágil, más concretamente, con iteraciones cortas en tiempo, de pocas semanas, normalmente un mes y raramente más de dos

El ciclo de vida iterativo e incremental es una de las buenas prácticas de ingeniería del software más antiguas, su primer uso en el software se data en los 50.

CICLO DE VIDA ÁGIL

Sería un ciclo de vida iterativo e incremental, con iteraciones cortas (semanas) y sin que dentro de cada iteración tenga porque haber fases lineales (tipo cascada). A partir de la anterior, matizaciones, adaptaciones, etc., hay por cada metodología ágil que existe.

Quizá el caso más popular es el de Scrum. Hace ya sus años, en el 85, y en la primera presentación oficial de Scrum, Ken Schwaber, uno de sus creadores, hablaba sobre el ciclo de vida de Scrum, y sus diferencias con los anteriores ciclos de vida

Según comenta Ken Schwaber, el ciclo de vida en Cascada y el Espiral cierran el contexto y la entrega al inicio de un proyecto. La principal diferencia entre cascada, espiral e iterativo y los ciclos de vida ágiles, concretamente en Scrum, es que estos últimos asumen que el análisis, diseño, etc., de cada iteración o Sprint son impredecibles. Los Sprints, o iteraciones cortas, no son (o a priori no tienen porqué) lineales y son flexibles.

Pero, como comentaba, cada metodología de las llamadas ágiles, FDD, Crystal, DSDM, XP, etc., matizará su ciclo de vida.

CURIOSIDADES: El ciclo de vida iterativo e incremental es incluso ¡más antiguo que el cascada!

Con la creciente popularidad de los métodos ágiles en muchas ocasiones se cree que el ciclo de vida iterativo e incremental es una práctica moderna, nueva frente al antiguo ciclo de vida en cascada, pero su aplicación data de mitad de los años 50, y desde entonces ha sido ampliamente usado y se ha escrito mucho sobre él (C. Larman& V. Basili, 2003)[2].

En 1950 la construcción del avión cohete X-15 supuso un hito en la aplicación del ciclo de vida iterativo e incremental, hasta el punto de que dicho ciclo de vida fue una de las principales contribuciones al éxito del proyecto.

J. Garzás, Veterano ciclo de vida iterativo e incremental (J. Garzás, 2010).

EL PROYECTO ÁGIL

Un proyecto ágil se podría definir como una manera de enfocar el desarrollo software mediante un ciclo iterativo e incremental, con equipos que trabajan de manera altamente colaborativa y auto-organizados. Es decir, un proyecto ágil es un desarrollo iterativo más

[2] Larman, C. & Basili, V. (2003). Iterative and incremental Development: A brief History. Computer 36(6), 47-56

la segunda gran característica implicada directamente por la iteración extrema: equipos colaborativos y auto-organizados.

A diferencia de ciclos de vida iterativos e incrementales más “relajados”, en un proyecto ágil cada iteración no es un “mini cascada”. Esto no es así, porque el objetivo de acortar al máximo las iteraciones (normalmente entre 1 y 4 semanas) lo hace casi imposible. Cuanto menor es el tiempo de iteración más se solapan las tareas. Hasta el punto que implicará que, de manera no secuencial, muchas veces solapada, y repetitivamente, durante una iteración se esté casi a la vez diseñando, programando y probando. Lo que implicará máxima colaboración e interacción de los miembros del equipo. Implicará equipos multidisciplinares, es decir, que no hay roles que sólo diseñen o programen, todos pueden diseñar y programar. E implica auto-organización, es decir, que en la mayoría de los proyectos ágiles no hay, por ejemplo, un único jefe de proyecto responsable de asignar tareas.

Frente a un ciclo de vida en cascada, o frente a un ciclo iterativo de iteraciones largas compuestas por pequeñas cascadas, en un ciclo de vida ágil:

- **El diseño en el desarrollo y las pruebas se realizan de manera continua.** En un ciclo de vida en cascada se realizan de manera secuencial.
- **Las personas que integran un equipo de desarrollo realizarán diferentes tareas.** No existen equipos o roles especializados, que sin embargo sí existían en el ciclo de vida en cascada.
- **La duración de una iteración es fija**, incluso si no se han podido desarrollar todas las actividades planificadas para la misma. En cambio, en los ciclos de vida en cascada generalmente se supera el tiempo planificado.

Un proyecto ágil lleva la iteración al extremo:

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

Normalmente **un proceso ágil se basa en un ciclo de vida iterativo e incremental, pero no todo proceso iterativo e incremental es un proceso ágil**. Además, están la colaboración y los equipos auto-organizados, entonces, ¿qué caracteriza un proyecto ágil? en el siguiente capítulo veremos que derivado de todo lo anterior se debe cumplir además otros valores y principios.

EL MANIFIESTO ÁGIL

El 12 de febrero de 2001, 17 destacados y conocidos profesionales de la ingeniería del software escribían en Utah el **Manifiesto Ágil**. Entre ellos estaban los creadores de

algunas de las metodologías[3] ágiles más conocidas en la actualidad: XP, Scrum, DSDM, Crystal, etc. Su objetivo fue establecer los valores y principios que permitirían a los equipos desarrollar software rápidamente y respondiendo a los cambios que puedan surgir a lo largo del proyecto.

Se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

De esta forma se establecieron cuatro valores ágiles:

1. **Valorar a los individuos y las interacciones del equipo de desarrollo sobre el proceso y las herramientas.** Se tendrán en cuenta las buenas prácticas de desarrollo y gestión de los participantes del proyecto (siempre dentro del marco de la metodología elegida). Esto facilita el trabajo en equipo y disminuir los impedimentos para que realicen su trabajo. Asimismo, compromete al equipo de desarrollo y a los individuos que lo componen.
2. **Desarrollar software que funciona más que conseguir una documentación exhaustiva.** No es necesario producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante. Los documentos deben ser cortos y centrarse en lo fundamental. La variación de la cantidad y tipo de documentación puede ser amplia dependiendo el tipo de cliente o de proyecto. El hecho de decir que la documentación es el código fuente y seguir esa idea sin flexibilidad puede originar un caos. El problema no es la documentación sino su utilidad.
3. **La colaboración con el cliente más que la negociación de un contrato.** Es necesaria una interacción constante entre el cliente y el equipo de desarrollo. De esta colaboración depende el éxito del proyecto. Este es uno de los puntos más complicados de llevar a cabo, debido a que muchas veces el cliente no está disponible. En ese caso desde dentro de la empresa existirá una persona que represente al cliente, haciendo de interlocutor y participando en las reuniones del equipo.
4. **Responder a los cambios más que seguir estrictamente un plan.** Pasamos de la anticipación y la planificación estricta sin poder volver hacia atrás a la adaptación. La flexibilidad no es total, pero existen muchos puntos (todos ellos controlados) donde se pueden adaptar las actividades.

LO QUE NO DICE

Ausencia total de documentación.

Ausencia total de planificación: planificar y ser flexible es diferente a improvisar.

El cliente debe hacer todo el trabajo y será el Jefe de Proyecto.

El equipo puede modificar la metodología sin justificación.

[3] El término metodología es utilizado de manera coloquial, siendo rigurosos las metodologías son más bien marcos de trabajo o frameworks.

LOS PRINCIPIOS ÁGILES

De estos cuatro valores surgen los doce principios del manifiesto. Estos principios son características que diferencian un proceso ágil de uno tradicional. Los principios son los siguientes:

1. La prioridad es **satisfacer al cliente** mediante **entregas tempranas y continuas** de software que le **aporten valor**.
2. **Dar la bienvenida a los cambios.** Se capturan los cambios para que el cliente tenga una ventaja competitiva.
3. **Entregar frecuentemente software que funcione** desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.
4. **La gente del negocio y los desarrolladores deben trabajar juntos** a lo largo del proyecto.
5. **Construir el proyecto en torno a individuos motivados.** Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.
6. **El diálogo cara a cara es el método más eficiente** y efectivo para comunicar información dentro de un equipo de desarrollo.
7. **El software que funciona es la medida fundamental de progreso.**
8. Los procesos ágiles **promueven un desarrollo sostenible**. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.
9. La **atención continua a la calidad técnica y al buen diseño** mejora la agilidad.
10. La **simplicidad** es esencial.
11. Las mejores arquitecturas, requisitos y diseños surgen de los **equipos organizados por sí mismos**.
12. En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser **más efectivo**, y según esto ajusta su comportamiento.

COMPARATIVA RESUMEN DE AGILIDAD VS TRADICIONAL

A modo de resumen, esta son las principales diferencias entre las metodologías ágiles y tradicionales:

CARACTERISTICA	TRADICIONAL	ÁGIL
PROCESO	Altamente controlado	Ligeramente controlado
CONTRATO	Cerrado	Flexible
INTERACCIÓN CON EL CLIENTE	Reuniones	Forma parte del equipo de desarrollo
TAMAÑO DE LOS GRUPOS	Grandes	Pequeños, menos de 10 integrante
ARQUITECTURA DEL SOFTWARE	Muy importante	Menos importante
DOCUMENTACIÓN	Exhaustiva	Poca, sólo la necesaria para entender el código

UNIÓN LOS MODELOS DE PROCESOS Y METODOLOGÍAS ÁGILES

En ocasiones existe la percepción de que es incompatible unir modelos como CMMI o ISO/IEC 15504 – ISO/IEC 12207 con metodologías ágiles. Sin embargo, esta concepción no es correcta, ya que los modelos y las metodologías se encuentran en distintos niveles de abstracción.

Los modelos de procesos establecen qué es lo que espera encontrarse en los procesos, pero son las metodologías las que indicarán cómo deben realizarse. Es por esto que el uso de modelos de procesos y metodologías ágiles no debe considerarse un aspecto contradictorio si no complementario.

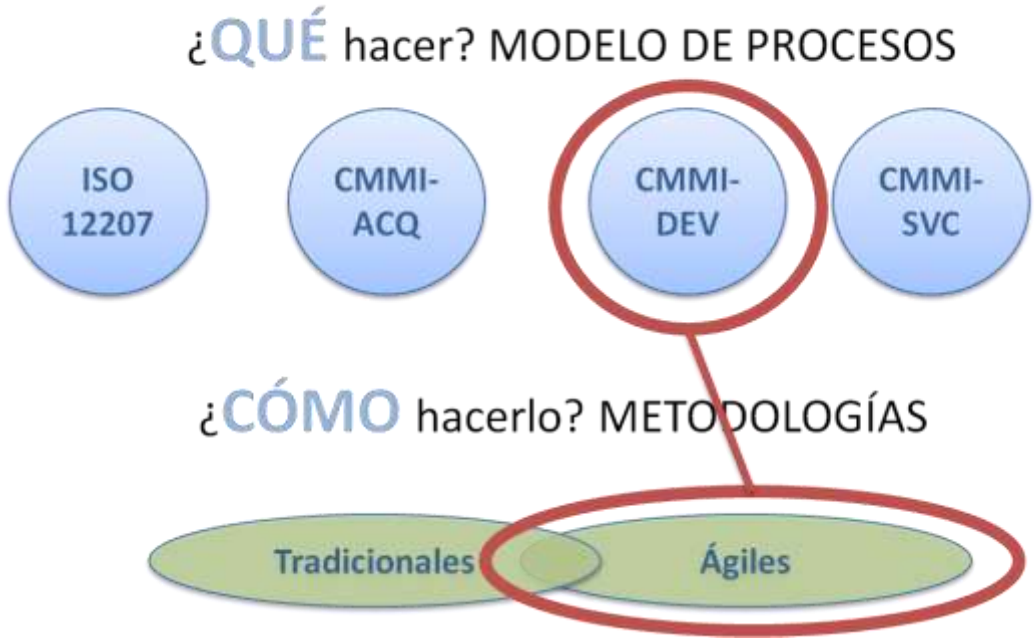


Figura 2 Unión de Modelos

Capítulo 2
EL PRODUCT OWNER Y LAS HISTORIAS DE USUARIO

EL PRODUCT OWNER

El "**product owner**" (o propietario del producto) es aquella persona con una visión muy clara del producto que se quiere desarrollar, que es capaz de transmitir esa visión al equipo de desarrollo y, además, está altamente disponible para transmitirla.

El product owner también es el **responsable** de la comunicación entre equipo y usuarios, y de **gestionar qué trabajo se tiene que desarrollar, en qué orden y qué valor se va entregando**.

Normalmente, y según lo anterior, el product owner suele ser uno de los futuros usuarios del sistema, o a veces, alguien de marketing o cualquier persona que entienda lo que quieren los usuarios, el mercado del producto, la competencia, y el futuro del sistema en desarrollo.

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

Desgraciadamente, es frecuente encontrar implantaciones erróneas alrededor de este rol. En unas ocasiones sus tareas se minimizan, en otras suelen pasarse por alto muchas de sus importantes responsabilidades, etc.

Quizás **el papel más importante** del product owner en la gestión de un proyecto ágil **es gestionar qué historias de usuario se desarrollarán, en qué orden y cuáles no**.

La mayoría de las ocasiones, el negocio, los usuarios, etc., van proporcionando una cantidad de ideas a implementar, que se van convirtiendo en historias de usuario, muy superiores en número. La función del **product owner es vital, debe ser quien decida que historias de usuario entran en el product backlog, cuáles no, y además la prioridad de las historias del product backlog**.

Habitualmente, en un proyecto ágil, para la gestión y priorización de lo que se espera que haga el product owner se usa el "product backlog" (en terminología Scrum), que es un repositorio priorizado de funcionalidades (normalmente en formato de historia de usuario) a ser implementadas por el equipo de desarrollo.

HISTORIAS DE USUARIO

Un proyecto de desarrollo software tiene como objetivo principal satisfacer las necesidades del cliente o usuario. A la hora de reflejar estas necesidades, los requisitos son uno de los aspectos más importantes.

En las metodologías ágiles la descripción de estas necesidades se realiza a partir de las historias de usuario (user story) que son, principalmente, lo que el cliente o el usuario quiere

que se implemente; es decir, son una descripción breve, de una funcionalidad software tal y como la percibe el usuario (M. Cohn, 2004)[4].

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

Las historias de usuario se representan normalmente en post-it por su fácil manejabilidad, y para evitar que se extiendan y se conviertan en especificaciones de requisitos (hablaremos sobre esto más adelante en este capítulo), como se muestra en la Figura 3.

Figura 3 Ejemplo de historia de usuario en pósit.

Ron Jeffries (R. Jeffries, 2001)[5] escribía que una historia no es sólo una descripción de una funcionalidad, normalmente en un pósit, una historia de usuario además está formada por tres partes:

Figura 4 Ejemplo de aclaraciones representadas en un pósit.

[4] Cohn, M. (2004). User stories applied: For agile software development Addison-Wesley.
[5] Jeffries, R. (2001). Essential XP: Card, conversation, confirmation. Disponible en: http://xprogramming.com/articles/expcardconversationconfirmation/

1. **Creación de la tarjeta** (o pósit). Una descripción escrita con un título asociado que sirve como identificación de la funcionalidad. También puede contener la estimación, el valor de negocio que tiene la historia para el usuario y/o cliente.
2. **Conversación.** El diálogo llevado a cabo entre el equipo y el cliente o usuario para aclarar los detalles y las dudas que puedan surgir sobre la historia de usuario. Esta es la parte más importante de la historia. En el caso de estar usando pósit, estas aclaraciones (o confirmaciones) se pueden especificar a la vuelta del mismo, como se muestra en la Figura 2.
3. **Confirmación.** Selección (entre el equipo y el cliente o usuario) de las pruebas que se realizarán para comprobar que la historia de usuario se ha completado con éxito.

Aparte de su manejabilidad y su capacidad para la interacción con el cliente o usuario, las historias de usuario han tomado un gran protagonismo en las metodologías ágiles ya que:

- Favorecen la participación del equipo en la toma de decisiones.
- Se crean y evolucionan a medida que el proyecto avanza.
- Son peticiones concretas y pequeñas.
- Contienen la información imprescindible.
- Apoyan la cooperación, colaboración y conversación entre los miembros del equipo.

Figura 5 Ejemplo real de uso del pósit con tareas e historias de usuario.

INFORMACIÓN DE UNA HISTORIA DE USUARIO

Aunque dependiendo del proyecto se podría incluir cualquier otro campo que proporcionase información útil, en este apartado se describen aquellos campos que se consideran más necesarios para describir de manera adecuada una historia de usuario. Estos campos se pueden observar en la Figura 6.

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

1. **ID**: identificador de la historia de usuario.
2. **Título**: título descriptivo de la historia de usuario.
3. **Descripción**: descripción sintetizada de la historia de usuario. Si bien el estilo puede ser libre, la historia de usuario debe responder a tres preguntas: ¿Quién se beneficia? ¿Qué se quiere? y ¿Cuál es el beneficio? Cohn (M. Cohn, 2004)[6] recomienda seguir el siguiente patrón: **Como [rol del usuario], quiero [objetivo], para poder [beneficio]**. Con este patrón se garantiza que la funcionalidad se

[6] Cohn, M. (2004). User stories applied: For agile software development Addison-Wesley.

captura a un alto nivel y que se está describiendo de una manera no demasiado extensa.

4. **Estimación**: estimación del tiempo de implementación de la historia de usuario en unidades de desarrollo, conocidas como puntos de historia (estas unidades representarán el tiempo teórico de desarrollo/persona que se estipule al comienzo del proyecto).
5. **Valor**: valor (normalmente numérico) que aporta la historia de usuario al cliente o usuario. El objetivo del equipo es maximizar el valor y la satisfacción percibida por el cliente o usuario en cada iteración. Este campo determinará junto con el tiempo, el orden con el que las historias de usuario van a ser implementadas.

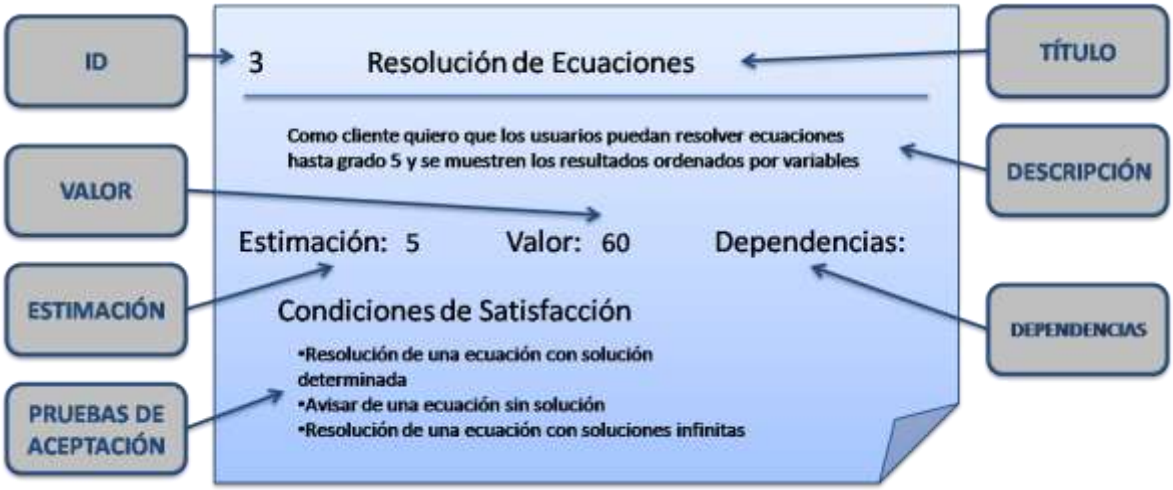


Figura 6 Historia de usuario.

6. **Dependencias**: una historia de usuario no debería ser dependiente de otra historia, pero en ocasiones es necesario mantener la relación. En este campo se indicarían los identificadores de otras historias de las que depende.
7. **Pruebas de aceptación**: pruebas consensuadas entre el cliente o usuario y el equipo que el código debe superar para dar como finalizada la implementación de la historia de usuario. Este campo también se suele denominar "**criterios o condiciones de aceptación**".

Cohn en (M. Cohn, 2004)[6] comenta que si bien las historias de usuario son lo suficientemente flexibles como para describir la funcionalidad de la mayoría de los sistemas, no son apropiadas para todo. Si por cualquier razón, se necesita expresar alguna necesidad de una manera diferente a una historia de usuario, recomienda que se haga. Por ejemplo, las interfaces de usuario se suelen describir en documentos con muchos pantallazos. Igual puede ocurrir con documentos sobre especificaciones de seguridad, normativas, etc.

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

INTERPRETACIONES INCORRECTAS DEL CONCEPTO DE HISTORIA DE USUARIO

¿Las historias de usuario equivalen a requisitos funcionales?

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

Una historia de usuario describe funcionalidad que será útil para el usuario y/o cliente de un sistema software. Y aunque normalmente las historias de usuario asociadas a las metodologías ágiles, suelen escribirse en pósit o tarjetas, son mucho más que eso. Como se ha comentado anteriormente, una historia no es sólo una descripción de una funcionalidad, sino también es de vital importancia la conversación que conllevan.

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

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

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

Para resolver el anterior problema hay que entender que el objetivo de las historias de usuario es, entre otros, lograr la interacción entre el equipo y el cliente o el usuario por encima de documentar (manifiesto ágil, ver lección 1) por lo que no se deben sobrecargar de información. Sin embargo, la realidad de los proyectos y de los negocios es otra y hace que la teoría se deba ajustar a la práctica, por lo que se pueden dar varias soluciones para reflejar toda esa información que en un primer momento parece que no cuadra en las historias de usuario:

- Relajar la agilidad usando los tradicionales requisitos en ciclos de vida ágil.
- Usar casos de uso en vez de historias de usuario.
- Utilizar técnicas de trazabilidad para relacionar las historias de usuario con otros documentos: de diseño, normativas, etc.

¿Las historias de usuario equivalen a casos de uso?

Otro concepto que suele crear confusión sobre las historias de usuario son los casos de uso. Aunque hay quien ha logrado incluir casos de uso en su proceso ágil, no quiere decir que las historias de usuario sean equivalentes a los casos de uso. Realmente, como comenta Cockburn (A. Cockburn, 2002)[7], las historias de usuario están más cerca de la captura de requisitos (aquella frase que sirve para extraer las necesidades del usuario).

Básicamente, **si decíamos que una historia de usuario es el "qué" quiere el usuario, el caso de uso es un "cómo" lo quiere**.

Generalmente, cuando un proyecto comience a seguir una metodología ágil, se deberían olvidar completamente los casos de uso y el quipo debería centrarse en la realización de historias de usuario. Según Cockburn(A. Cockburn, 2008)[8], esto puede producir los siguientes problemas:

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

Por tanto, si en un proyecto ocurre alguno de estos problemas se puede barajar la posibilidad (relajando la agilidad) de complementar las necesidades descritas en las historias de usuario con casos de uso donde quede reflejado el comportamiento necesario para cumplir dichas necesidades.

> *En el caso de que se usen las historias de usuario y los casos de uso de manera complementaria, una historia de usuario suele dar lugar a la especificación de varios casos de uso.*

[7] Cockburn, A. (2002). Agile software development. Boston, MA, USA. Addison-Wesley Longman Publishing Co., Inc.
[8] Cockburn, A. (2008). Why I still use use cases. Disponible en: http://alistair.cockburn.us/Why+I+still+use+use+cases

CREANDO BUENAS HISTORIAS DE USUARIO

Para asegurar la calidad de una historia de usuario, Bill Wake describió en 2003 un método llamado INVEST (Wake, 2003)[9]. El método se usa para comprobar la calidad de una historia de usuario revisando que cumpla las características descritas en la Tabla 1. Descripción del INVEST.

Independient (independiente)	Es importante que cada historia de usuario pueda ser planificada e implementada en cualquier orden. Para ello deberían ser totalmente independientes (lo cual facilita el trabajo posterior del equipo). Las dependencias entre historias de usuario pueden reducirse combinándolas en una o dividiéndolas de manera diferente.
Negotiable (negociable)	Una historia de usuario es una descripción corta de una necesidad que no incluye detalles. Las historias deben ser negociables ya que sus detalles serán acordados por el cliente/usuario y el equipo durante la fase de "conversación". Por tanto, se debe evitar una historia de usuario con demasiados detalles porque limitaría la conversación acerca de la misma.
Valuable (valiosa)	Una historia de usuario tiene que ser valiosa para el cliente o el usuario. Una manera de hacer una historia valiosa para el cliente o el usuario es que la escriba el mismo.
Estimable (estimable)	Una buena historia de usuario debe ser estimada con la precisión suficiente para ayudar al cliente o usuario a priorizar y planificar su implementación. La estimación generalmente será realizada por el equipo de trabajo y está directamente relacionada con el tamaño de la historia de usuario (una historia de usuario de gran tamaño es más difícil de estimar) y con el conocimiento del equipo de la necesidad expresada (en el caso de falta de conocimiento, serán necesarias más fases de conversación acerca de la misma).
Small (pequeña)	Las historias de usuario deberían englobar como mucho unas pocas semanas/persona de trabajo. Incluso hay equipos que las restringen a días/persona. Una descripción corta ayuda a disminuir el tamaño de una historia de usuario, facilitando su estimación.
Testable (comprobable)	La historia de usuario debería ser capaz de ser probada (fase "confirmación" de la historia de usuario). Si el cliente o usuario no sabe cómo probar la historia de usuario significa que no es del todo clara o que no es valiosa. Si el equipo no puede probar una historia de usuario nunca sabrá si la ha terminado o no.

Tabla 1 Descripción del método INVEST

[9] Wake, B. (2003). INVEST in good stories, and SMART tasks. Disponible en: http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

ASIGNAR VALOR A UNA HISTORIA DE USUARIO

Como se comentaba en la introducción del tema, los ítems del Product backlog, deben estar priorizados, es decir, deben tener asignados un valor. **Dicho valor es asignado por el Product Owner**, y pondera básicamente las siguientes variables:

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

Uno de los aspectos más importantes aquí es que la definición de "valor" para cada cliente puede variar. Por lo tanto, es muy recomendable incluir algún tipo de escala cualitativa.

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

Existen otro tipo de ponderaciones, por ejemplo, la técnica MoSCoW. Esta técnica fue definida por primera vez en el libro Case Method Fast-Track: A RAD Approach, en el año 2004. Su fin es obtener el entendimiento común entre cliente y el equipo del proyecto sobre la importancia de cada requisito o historia de usuario. La clasificación es la siguiente:

- **M - MUST**. Se debe tener la funcionalidad. En caso de que no exista la solución a construir fallará.
- **S – SHOULD**. Se debería tener la funcionalidad. La funcionalidad es importante pero la solución no fallará si no existe.
- **C – COULD**. Sería conveniente tener esta funcionalidad. Es en realidad un deseo.
- **W - WON'T**. No está en los planes tener esta funcionalidad en este momento. Posteriormente puede pasar a alguno de los estados anteriores.

Capítulo 3
SCRUM

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

El conjunto de buenas prácticas de Scrum se aplica esencialmente a la gestión de proyectos.

> *Por otro lado, aunque normalmente hablamos de la "metodología Scrum", lo correcto sería decir el "framework Scrum", porque realmente es un conjunto de buenas prácticas que necesita su adaptación en cada organización, o, incluso, a cada equipo. (J. Sutherland & Schwaber, 2011)[10].*

Las características principales de Scrum pueden resumirse en dos:

- El desarrollo software mediante iteraciones incrementales.
- Las reuniones a lo largo del proyecto.

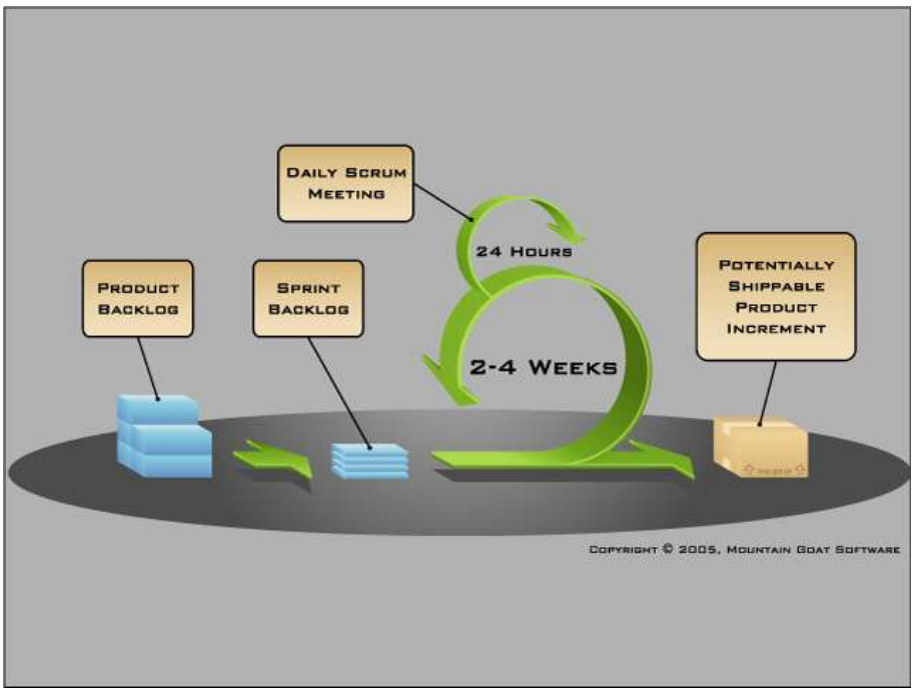


Figura 7 Framework Scrum

Como se indica en la "Scrum Guide" (K. Schwaber & Sutherland, 2011)[10] , existen tres pilares en los que se basa:

- **Transparencia**: todos los aspectos del proceso que afectan al resultado son visibles para todos aquellos que administran dicho resultado. Por ejemplo, se

[10] Schwaber, K., & Sutherland, J. (2010). Scrum guide. Disponible en: https://www.scrum.org/Scrum-Guides

utilizan pizarras y otros mecanismos o técnicas colaborativas para mejorar la comunicación.

- **Inspección**: se debe controlar con la frecuencia suficiente los diversos aspectos del proceso para que puedan detectarse variaciones inaceptables en el mismo.
- **Revisión**: el producto debe estar dentro de los límites aceptables. En caso de desviación se procederá a una adaptación del proceso y el material procesado.

EL EQUIPO EN SCRUM

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

Cada equipo Scrum tiene tres roles:

- **Scrum Master**: es el responsable de asegurar que el equipo Scrum siga las prácticas de Scrum. Sus principales funciones son:
  - ✓ Ayuda a que el equipo Scrum y la organización adopten Scrum.
  - ✓ Liderar el equipo Scrum, buscando la mejora en la productividad y calidad de los entregables.
  - ✓ Ayudar a la autogestión del equipo.
  - ✓ Gestiona e intenta resolver los impedimentos con los que el equipo se encuentra para cumplir con las tareas del proyecto.
- **Propietario del Producto (ProductOwner)**: es la persona responsable de gestionar las necesidades que serán satisfechas por el proyecto y asegurar el valor del trabajo que el equipo lleva a cabo. Su aportación al equipo se basa en:
  - ✓ Recolectar las necesidades o historias de usuario.
  - ✓ Gestionar y ordenar las necesidades (representadas por las historias de usuario, descritas en la lección 2).
  - ✓ Aceptar el producto software al finalizar cada iteración.
  - ✓ Maximizar el retorno de inversión del proyecto.
- **Equipo de desarrollo**: El equipo está formado por los desarrolladores, que convertirán las necesidades del Product Owner en un conjunto de nuevas funcionalidades, modificaciones o incrementos del producto software final. El equipo de desarrollo tiene características especiales:
  - ✓ **Auto-gestionado**: el mismo equipo supervisa su trabajo. En Scrum se potenciarán las reuniones del equipo (aquí tienes un post sobre ese tema: http://www.javiergarzas.com/2012/07/equipo-agil-auto-organizado.html), aumentando la comunicación. No existe el rol clásico de jefe de proyecto. El Scrum Master tiene otras responsabilidades vistas en el apartado anterior.
  - ✓ **Multifuncional**: no existen compartimientos estancos o especialistas, cada integrante del equipo puede encargarse de tareas de programación, pruebas, despliegue, etc. Asimismo, las personas pueden tener capacidades diferentes o conocimientos más profundos en diferentes áreas. Lo importante es que cualquier integrante del equipo sea capaz de realizar cualquier función.
  - ✓ **No distribuidos**: es conveniente que el equipo se encuentre en el mismo lugar físico. Esto facilita la comunicación y la autogestión que nace del mismo equipo.

No obstante se ha conseguido realizar proyectos Scrum con equipos distribuidos gracias a herramientas de trabajo colaborativo (Hossain et al., 2009)[11].

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

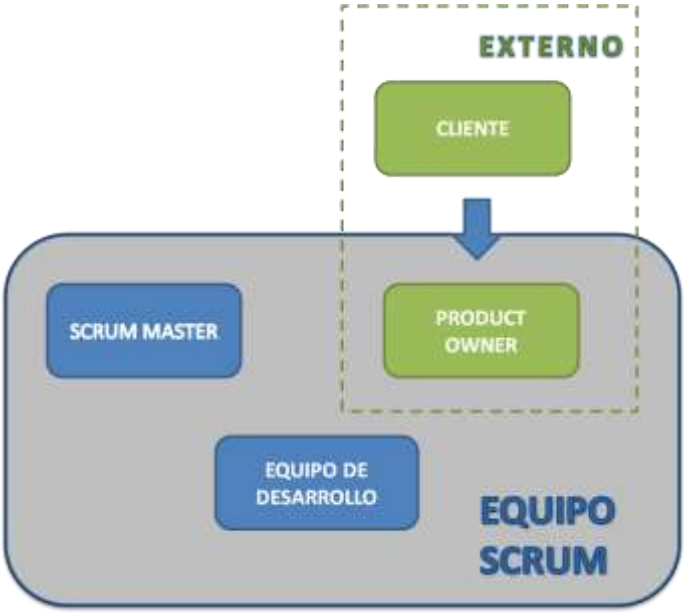


Figura 8 Equipo Scrum

EL PRODUCT BACKLOG

La pila de producto o Product Backlog (utilizaremos las palabras en inglés al ser la forma más utilizada en los proyectos) en Scrum es uno de los elementos fundamentales. A partir del Product Backlog se logra tener una única visión durante todo el proyecto. Y, por lo tanto, los fallos en el Product Backlog repercutirán profundamente en el proyecto.

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

Para ello, este listado estará ordenado. Aunque no existe un criterio preestablecido en Scrum para ordenar las historias de usuario, el más aceptado es partir del valor que aporta al negocio la implementación de la historia de usuario. **El responsable de ordenar el**

[11] Hossain, E., Babar, M. A., & Hye-young Paik. (2009). Using scrum in global software development: A systematic literature review. Artículo presentado en Fourth IEEE International Conference on Global Software Engineering, 2009. pp. 175.

Product Backlog es el Product Owner, aunque también puede ser ayudado o recibir asesoramiento de otros roles como, por ejemplo, el Scrum Master y el equipo de desarrollo.

Tal y como se describe en (Pichler, 2010)[12] un Product Backlog cuenta, esencialmente, con cuatro cualidades: debe estar detallado de manera adecuada, estimado, emergente y priorizado:

- **Detallado adecuadamente**: el grado de detalle dependerá de la prioridad. Las historias de usuario que tengan una mayor prioridad se describen con más detalle. De esta manera las siguientes funcionalidades a ser implementadas se encuentran descritas correctamente y son viables. Como consecuencia de esto, las necesidades se descubren, se descomponen, y perfeccionan a lo largo de todo el proyecto.
- **Estimado**: las estimaciones a menudo se expresan en días ideales o en términos abstractos. Saber el tamaño de los elementos del Product Backlog ayuda a darle prioridad y a planificar los siguientes pasos. Una de las técnicas que se pueden emplear para estimar las historias de usuario es el Planning Poker, que veremos en la Lección 4.
- **Emergente**: las necesidades se van desarrollando y sus contenidos cambian con frecuencia. Los nuevos elementos se descubren y se agregan a la lista teniendo en cuenta los comentarios de los clientes y usuarios. Así mismo, otros elementos podrán ser modificados o eliminados.
- **Priorizado**: los elementos del Product Backlog se priorizan. Los elementos más importantes y de mayor prioridad aparecen en la parte superior de la lista. Puede no ser necesario priorizar todos los elementos en un primer momento, sin embargo, sí es conveniente identificar los 15-20 elementos más prioritarios.

EL SPRINT

Como hemos visto anteriormente, una de las bases de los proyectos ágiles es el desarrollo mediante las iteraciones incrementales. **En Scrum a cada iteración se le denomina Sprint**. Scrum recomienda iteraciones cortas, por lo que **cada Sprint durará entre 1 y 4 semanas**. Y como resultado se creará un producto software potencialmente entregable, un prototipo operativo. Las características que van a implementarse en el Sprint provienen del Product Backlog.

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

Una de las implementaciones que más se llevan a cabo para el Sprint Backlog es crear un desglose de tareas normalmente representadas en una tabla, donde se describe cómo el equipo va a implementar las historias de usuario durante el siguiente Sprint. Al realizar las tareas se dividen en horas, donde **ninguna tarea debe durar más de 16 horas** al ser realizada por un integrante del equipo. Si una tarea es mayor de 16 horas, es recomendable

[12] Pichler, R. (2010). Agile product management with scrum: Creating products that customers love Addison-Wesley Professional.

dividirla en otras menores. Además, la lista de tareas se mantendrá inamovible durante toda la iteración.

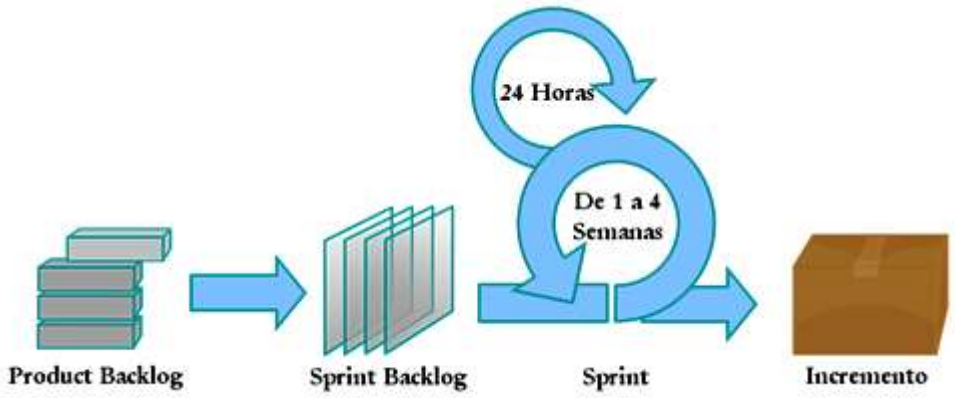


Figura 9 Ejemplo de un proceso Scrum.

Aun así, en muchas ocasiones es complicado realizar una división de tareas a partir de las historias de usuario elegidas para el Sprint. Se podría utilizar una descripción a alto nivel o un diseño tradicional. La relación entre el Product Backlog, el Sprint Backlog y el Sprint está gráficamente explicada en la Figura 9.

> *Importante: Aunque todos los Sprints dan como resultado un incremento del producto software, no todos implican un paso a producción. Es responsabilidad del ProductOwner y los clientes decidir el momento en el que los incrementos son puestos en producción.*
>
> *Una posibilidad para realizar la puesta en producción es con los denominados "**Sprints de Release**". Estos Sprints contendrán, en general, tareas solamente relacionadas con el despliegue, instalación y puesta en producción del sistema. Es decir, no existen tareas donde se agreguen nueva funcionalidad.*

Para mejorar la gestión de las historias de usuario y las tareas de cada Sprint usualmente se utilizan pizarras u otros mecanismos que brinden información inmediata al equipo. En el ejemplo de la Figura 10, cada historia de usuario del Product Backlog es dividida en tareas.

Figura 10 Product Backlog y Sprint Backlog.

Además, el equipo de desarrollo deberá estimar el esfuerzo que nos llevarán las tareas del Sprint Backlog (un método de estimación muy usado en Scrum y que veremos más adelante es el Planning Poker, descrito en la Capitulo 4).

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

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

REUNIONES

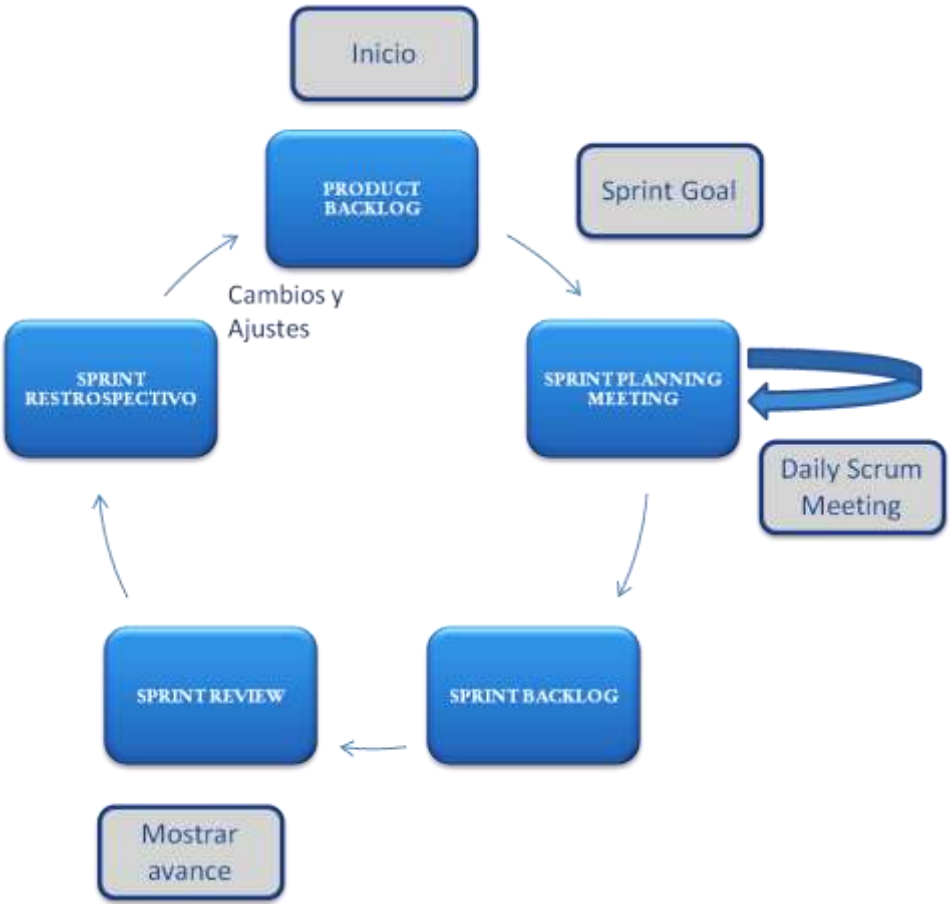


Figura 11 Reuniones

Las reuniones son un pilar importante dentro de Scrum. Se realizan a lo largo de todo el Sprint como muestra la Figura 11. Se definen diversos tipos de reuniones:

- **Reunión de planificación del Sprint (Sprint Planning Meeting)**: se lleva a cabo al principio de cada Sprint, definiendo en ella que se va a realizar en ese Sprint. Esta reunión da lugar al Sprint Backlog. En esta reunión participan todos los roles. El Product Owner presenta el conjunto de historias de usuario en el ProductBacklog y el equipo de desarrollo selecciona las historias de usuario sobre las que se trabajará. La duración de la reunión no debe ser mayor de 8 horas y como resultado de la misma el equipo de desarrollo hace una previsión del trabajo que será completada durante el Sprint.
- **Reunión diaria (Daily Scrum)**: es una reunión diaria de no más de 15 minutos en la que participan el equipo de desarrollo y el Scrum Master. En esta reunión cada miembro del equipo presenta lo qué hizo el día anterior, lo qué va a hacer hoy y los impedimentos que se ha encontrado.
- **Reunión de revisión del Sprint (Sprint Review Meeting)**: se realiza al final del Sprint. Participan el equipo de desarrollo, el Scrum Master y el Product Owner. Durante la misma se indica qué ha podido completarse y qué no, presentando el trabajo realizado al Product Owner. Por su parte el Product Owner (y demás interesados) verifican el incremento del producto y obtienen información necesaria para actualizar el Product Backlog con nuevas historias de usuario. No debe durar más de 4 horas.

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

MEDIR EL PROGRESO DEL PROYECTO

En el caso de las prácticas ágiles y en particular de Scrum uno de los mecanismos más utilizados es el gráfico BurnDown (J. Sutherland, 2001)[13].

Este gráfico representa el trabajo que queda por hacer en un Sprint en función del tiempo y compara el progreso real del Sprint con su planificación inicial, facilitando las labores de seguimiento del mismo. Normalmente el número de historias de usuario (aunque también suelen usarse puntos de historia) se muestra en el eje vertical y el tiempo en el eje horizontal. De esta manera, al representar el trabajo pendiente y el realizado hasta el momento, este gráfico es útil para predecir desviaciones y para estimar el siguiente Sprint, basándonos en la cantidad de funcionalidades realizadas habitualmente.

En el siguiente ejemplo, ver Figura 12, la línea roja muestra la cantidad de historias de usuario implementadas realmente durante el Sprint, mientras que la línea azul muestra la estimación realizada al inicio del Sprint. Como puede verse en el ejemplo, hasta el día cuatro del Sprint se habían implementado menos historias de usuario de las esperadas. El quinto fue especialmente productivo y los siguientes días, hasta el séptimo, no se implementaron más historias de usuario.

[13] Sutherland, J. (2001). Agile can scale: Inventing and reinventing scrum in five companies. 14(12).

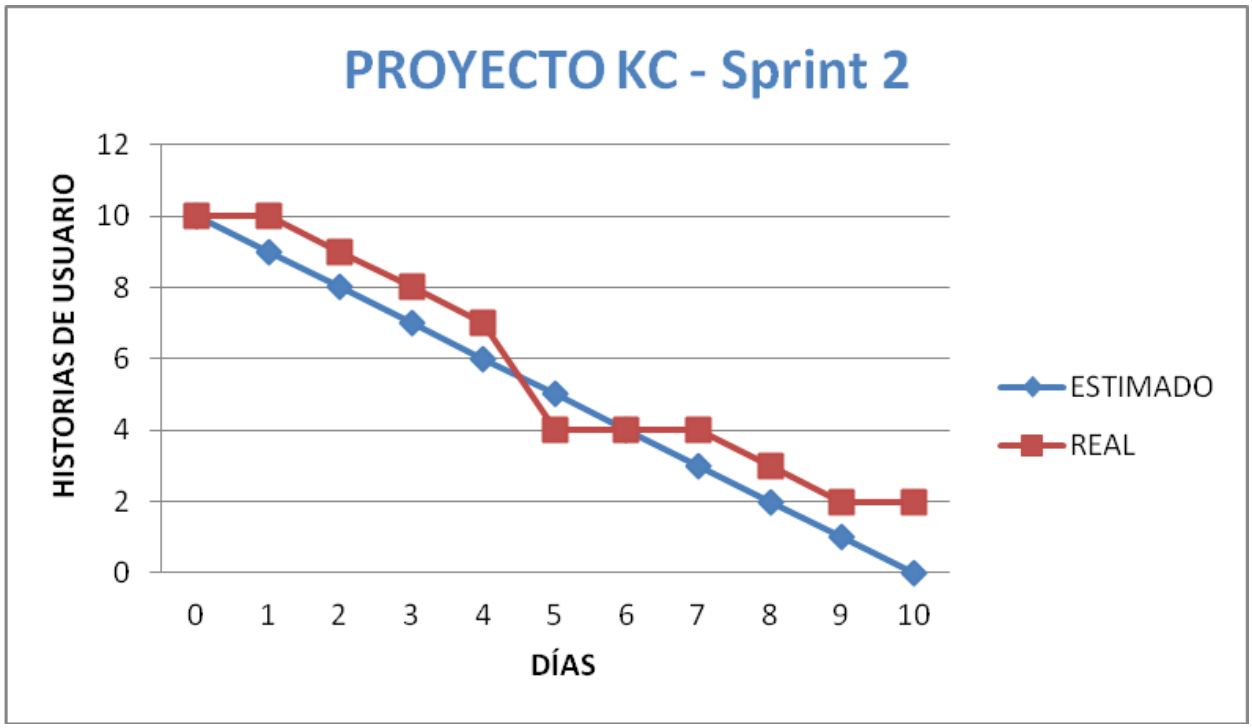


Figura 12 Gráfico BurnDown.

BENEFICIOS DE SCRUM

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

- **Entrega periódica de resultados**. El Product Owner establece sus expectativas indicando el valor que le aporta cada historia de usuario y cuando espera que esté completado. Por otra parte, comprueba de manera regular si se van cumpliendo sus expectativas.
- **Entregas parciales ("time to market")**. El cliente puede utilizar las primeras funcionalidades de la aplicación software que se está construyendo antes de que esté finalizada por completo. Por tanto, el cliente puede empezar antes a recuperar su inversión. Por ejemplo, puede utilizar un producto al que sólo le faltan características poco relevantes, puede introducir en el mercado un producto antes que su competidor, puede hacer frente a nuevas peticiones de clientes, etc.
- **Flexibilidad y adaptación respecto a las necesidades del cliente**. De manera regular el Product Owner redirige el proyecto en función de sus nuevas prioridades, de los cambios en el mercado, de los requisitos completados que le permiten entender mejor el producto, de la velocidad real de desarrollo, etc. Al final de cada iteración el Product Owner puede aprovechar la parte de producto completada hasta ese momento para hacer **pruebas de concepto** con usuarios o consumidores y tomar decisiones en función del resultado obtenido.
- **Mejores estimaciones**. La estimación del esfuerzo y la optimización de tareas es mejor si la realizan las personas que van a desarrollar la historia de usuario, dadas sus

diferentes especializaciones, experiencias y puntos de vista. De la misma manera, con iteraciones cortas la precisión de las estimaciones aumenta. En el capítulo siguiente veremos las técnicas de estimación en proyectos ágiles.

Capítulo 4
LA PLANIFICACIÓN ÁGIL

INTRODUCCIÓN

En la actualidad el desarrollo software sigue afectado por graves problemas:

- Los proyectos no se ajustan al presupuesto inicial
- Son entregados fuera de plazo
- Existe una baja calidad del software generado
- El software no cumple las especificaciones
- El código muchas veces es inmantenible

Lo cual dificulta la gestión y evolución del proyecto. Las prácticas ágiles no escapan a este problema, y de hecho por estar asociadas a requisitos cambiantes el desafío es mayor.

La mayoría de los errores que se producen con más frecuencia en los proyectos software están relacionados con aspectos de estimación. Entre estos errores se pueden destacar:

- **Calendario optimista**: la tendencia al estimar es hacerlo de manera optimista. Esta tendencia es general y se va disminuyendo con la experiencia. Aun así en un histórico de estimaciones estimar por encima es mucho menos frecuente que estimar por debajo.
- **Expectativas no realistas**: muchas veces las tareas sobre las que se ha estimado son ambiguas. Posteriormente el equipo seguramente se llega a un consenso común pero el cliente puede mantener otra expectativa. Y presionará para obtener el resultado que él cree es el correcto.
- **Confundir estimaciones con objetivos**: al estimar tenemos que tener en cuenta nuestra capacidad actual real y el histórico de productividad de la empresa. Es decir, si queremos terminar en 3 meses (un objetivo) puede que lo consigamos o no sabiendo que contamos con 2 programadores y que el 20% del tiempo estamos respondiendo incidencias de otros proyectos.
- **Omisión de estimar tareas necesarias**: las pruebas, la documentación, las tareas relacionadas con la gestión de configuración o la calidad puede que no se planifiquen. Pero muchas veces consumen tiempo.

LA UNIDAD DE ESTIMACIÓN: PUNTOS HISTORIA

Históricamente, la unidad clásica de estimación del “tamaño” de un requisito ha sido el Punto Función. Pero, en los últimos años, los puntos historia se han convertido en una unidad de tamaño muy usada, principalmente en proyectos ágiles. Por ejemplo, la técnica de estimación **Planning Poker**, que explicaremos más adelante, emplea los **Puntos Historia**.

Cualquier medida del tamaño, bien sean puntos historia, puntos función, puntos casos de uso, o cualquier otra, parte de los requisitos para calcular el tamaño del software. Estos requisitos pueden venir dados en diferentes formatos, más o menos detallados o

especificados. Formatos que pueden ir desde un documento, a un pliego de descripciones técnicas, casos de uso o, como suele ser habitual en proyectos ágiles, historias de usuario.

Los puntos historia son una unidad usada, normalmente, para medir el tamaño de una historia de usuario. El número de puntos historia asociados a una historia de usuario representa el tamaño global de la historia.

Más concretamente un punto historia es una fusión de la cantidad de esfuerzo que supone desarrollar la historia de usuario, la complejidad de su desarrollo y el riesgo inherente (Cohn, 2005)[14].

A diferencia del punto función, no existe una fórmula para calcular los puntos historia de una historia de usuario. A la hora de asignar puntos historia a cada historia de usuario lo que importa son los valores relativos de una historia frente al resto: una historia a la que se le asignan dos puntos historia debe requerir el doble de esfuerzo, complejidad o tamaño que una historia a la que se le asigna un único punto.

Y normalmente hay dos formas de hacer esta asignación:

- **Seleccionar una historia de las más pequeñas y asignarle 1 punto historia**. Esta historia de usuario con 1 punto historia hará de unidad, y al resto se le asignarán puntos historia en función de su complejidad respecto a ésta.
- **Seleccionar una historia de tamaño medio y asignarle un número en la mitad del rango de puntos historia a utilizar**. Normalmente, se usan rangos de puntos historia entre 1-10. Según esto, buscaríamos una historia de tamaño medio y le asignaríamos cinco puntos historia. Cada historia adicional se calcula comparándola con la primera historia.

En cualquier caso, recomendamos el primer método por ser más fácil de aplicar.

La escala de puntos historia

1 punto historia	Historia D, Historia B
2 punto historia	Historia C, Historia L
3 punto historia	Historia E, Historia G, Historia I, Historia J
5 punto historia	Historia F, Historia H
8 punto historia	Historia A

Tabla 2 Ejemplo de asignación de Puntos Historia a historias de usuario.

Varios estudios han demostrado que es mejor estimar usando rangos o escalas de posibles valores que se pueden asignar (Miranda, 2001)[15] a la estimación, es decir, fijando un tope máximo de puntos historia.

Según lo anterior, normalmente iremos agrupando historias en puntos historia, como muestra la Tabla 2.

[14] Cohn, M. (2005). Agile estimating and planning. Upper Saddle River, NJ, USA. Prentice Hall PTR.
[15] Miranda, E. (2001). Improving subjective estimates using paired comparisons.IEEE Software, 18(1), 87-91.

Normalmente, hay dos escalas de puntos historia que suelen tener buenos resultados:

1, 2, 3, 5, 8, etc.

1, 2, 4, 8, etc.

La primera escala es la secuencia de Fibonacci, útil porque la separación entre los números de la secuencia se hace más grande a medida que los números aumentan. En la segunda, cada número es dos veces el número que le precede. Estas secuencias no lineales funcionan bien porque reflejan mayor incertidumbre asociada a estimaciones de grandes unidades de trabajo.

Además, normalmente, deberíamos limitar los valores de la escala, por ejemplo, a 10 valores.

PLANNING POKER

Una vez elegidas las funcionalidades a realizarse en un Sprint, de acuerdo con lo priorizado por el cliente, es necesario realizar una estimación más detallada, utilizando, por ejemplo, la técnica de Planning Poker.

El Planning Poker es una técnica que se utiliza para asignar los puntos historia a las historias de usuario. Esta técnica recibe el nombre de Planning Poker ya que cada una de las personas implicadas en el proceso de estimación toma un mazo de cartas que suelen estar numeradas con alguna de las secuencias que vimos antes (por ejemplo, la de Fibronacci).

La técnica de Planning Poker está basada en el consenso y es utilizada para estimar el esfuerzo o el tamaño relativo de las tareas de desarrollo de software. Está muy arraigado en las técnicas ágiles por su sencillez, facilidad y bajo coste. No es una práctica de Scrum pero se suele utilizar con esta metodología ágil.

Figura 13 Planning Poker

Los pasos a seguir en el Planning Poker son los siguientes:

1. **Se presentan las historias de usuario a estimar** uno por uno haciendo una descripción de los mismos. Y se procede a discutir aquellos detalles más relevantes o que no hayan quedado claros. Suele darse un tiempo máximo de discusión para mejorar la productividad.
2. Tras este período de discusión **cada una de las personas** implicadas en el proceso de estimación **toma un mazo de cartas** (que están numeradas según la serie de Fibonacci, como objetivo de reflejar la incertidumbre inherente en la estimación). De ahí escoge la carta que representa su estimación del trabajo que implica implementar el requisito que se está discutiendo. Las estimaciones son privadas.
3. **Las unidades utilizadas pueden ser variadas y deben estar definidas previamente**. Pueden ser días reales, días ideales o puntos de la historia. Como consenso se denominan "**puntos de poker**". La diferencia entre días ideales y reales está en la cantidad de horas a tener en cuenta. En los proyectos en los que se suceden muchas interrupciones es conveniente estimar en días reales (o incluso en horas).
4. **Finalmente se publican todas las estimaciones**, es decir, cada integrante del equipo muestra **a la vez** la carta seleccionada (esto es así para evitar que las estimaciones de unos modifiquen las de otros). Si existe una gran dispersión entre las estimaciones (unos dicen 2, otros 21) se vuelve al discutir el requisito y se vuelve a realizar el proceso de estimación.
5. Generalmente **la dispersión en las estimaciones es síntoma de que la información** que manejan parte de los involucrados en el proceso de estimación **no es completa o exacta**. La segunda ronda de discusión permite aclarar aquellos puntos poco claros, diferencias de criterio y desvelar información que pueda no ser obvia sobre el requisito. En la siguiente ronda de estimación la dispersión de las estimaciones será mucho menor y se podrá llegar fácilmente a un consenso.

Usando Planning Poker es fácil estimar las historias de usuario de una manera ágil y rápida combinando la analogía y el juicio experto a un entorno grupal democrático. Nos lleva a estimaciones respaldadas por todos los involucrados y basadas en consensos.

UN EJEMPLO DE CONVERGENCIA EN PLANNING POKER

A continuación, se presenta un ejemplo de estimación y el problema de lograr la convergencia.

Como hemos visto anteriormente, **será necesario lograr la convergencia** en caso que las estimaciones de los integrantes del equipo no estén del todo de acuerdo y persistan algunas estimaciones muy diferentes.

Las personas cuya estimación esté más alejada del consenso **deberán explicar** en un tiempo corto (2 minutos, por ejemplo) **por qué su votación es más alta o más baja**.

A continuación, se presenta un histórico de Planning Poker. Donde los cuatro desarrolladores de un equipo que usa metodologías ágiles planean estimar 3 historias de usuario.

*Usando la notación PP = puntos de poker.

Historia de usuario 1				
Turno	D1	D2	D3	D4
1	2	5	6	10
2	5	7	9	9
3	6	8	8	9
4	8	8	8	9
Final	8 puntos de historia			
Historia de usuario 2				
Turno	D1	D2	D3	D4
5	8	3	3	8
6	7	7	7	7
Final	7 puntos de historia			
Historia de usuario 3				
Turno	D1	D2	D3	D4
7	7	2	6	14
8	7	5	6	10
9	7	7	6	7
Final	7 puntos de historia			

PLANNING POKER II

A continuación, plantearemos un ejemplo más complejo de estimación con Planning Poker.

Supongamos que en la próxima iteración debemos implementar 3 historias de usuario. Nuestro equipo está formado por 4 desarrolladores: D1, D2, D3 y D4. La unidad que usaremos serán los Puntos Poker.

Se presenta el siguiente histórico de estimaciones:

Historia de usuario 1				
Turno	D1	D2	D3	D4
1	2	5	6	10
2	5	7	9	9
3	6	8	8	9
4	8	8	8	9
Final	8 puntos de historia			
Historia de usuario 2				
Turno	D1	D2	D3	D4
5	8	3	3	8
6	7	7	7	7
Final	7 puntos de historia			
Historia de usuario 3				
Turno	D1	D2	D3	D4
7	7	2	6	14
8	7	5	6	10
9	7	7	6	7
Final	7 puntos de historia			

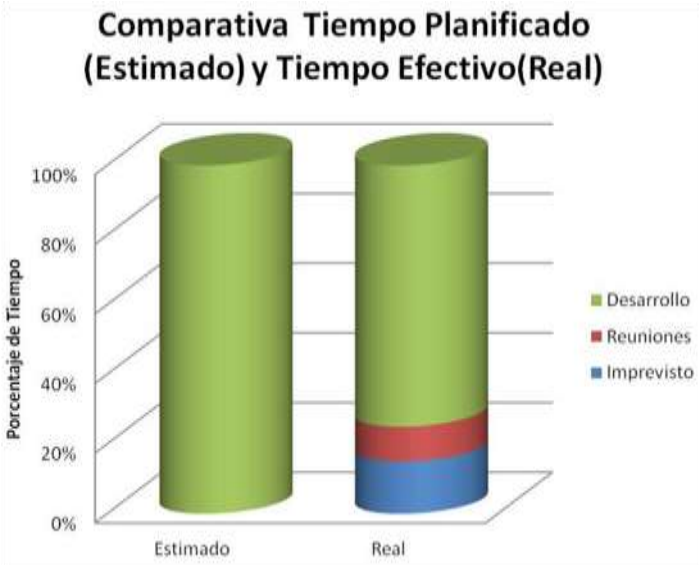
Siendo la estimación final de la iteración 22 puntos poker.

Ahora vamos a analizar cada uno de los históricos de las estimaciones de las historias de usuario.

- En la historia de usuario 1, se aprecia cómo D1 y D4 tenían una **diferencia de información**. Por ello, tras exponer sus razones vemos cómo realmente D4 era el mejor informado ya que a lo largo de los turnos, con sus respectivas exposiciones, los demás convergen a su estimación inicial.
- En la historia de usuario 2, hay una información asimétrica entre los desarrolladores D1 y D4 respecto a D2 y D3. Ello puede ser debido a **desconocimiento de la tarea**, o del campo a tratar, etc. Por ello tras una exposición de las razones, todos convergen en sus puntuaciones.
- El caso de la historia de usuario 3, es aquel en el que tanto D2 como D4 **carecen de la información completa**. Ello se ve ya que, al avanzar el histórico, convergen a la estimación inicial de los otros desarrolladores.

Importante: Este modo dinámico de estimación **hace participar a todos los asistentes**, **reduce el tiempo de estimación** de una funcionalidad, y lo más importante, **se consigue alcanzar una convergencia en la estimación de las historias de usuario**.

PELIGROS AL ESTIMAR

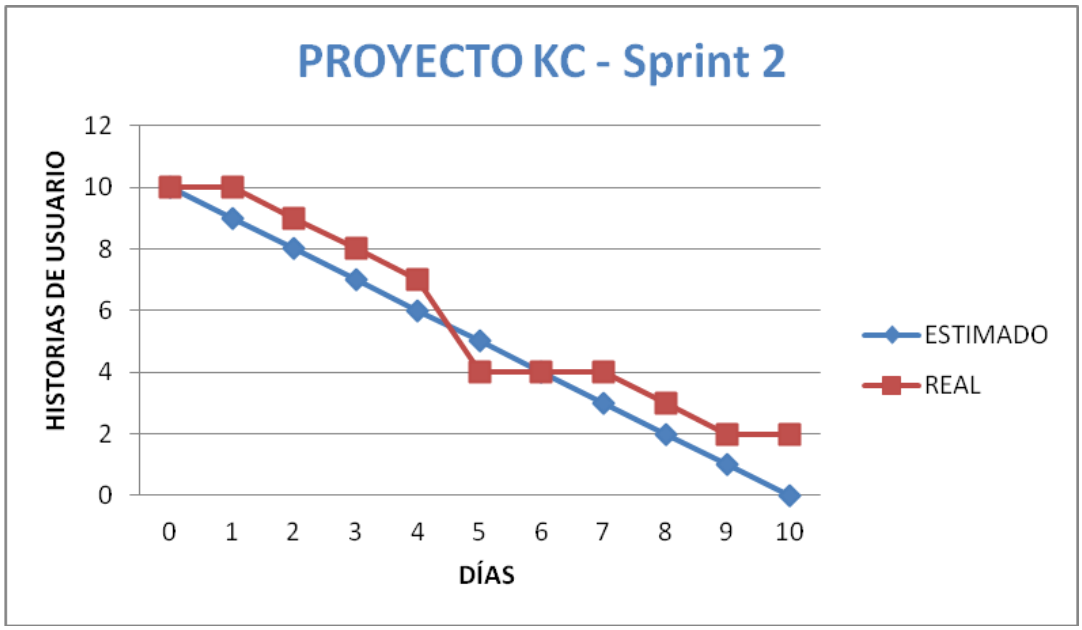


Al realizar estimaciones teniendo en cuenta jornadas de trabajo es necesario recordar lo siguiente: los días son "ideales". Éstos son aquellos en los que se cumplen las horas de trabajo definidas sin interrupciones, interferencias o distracciones de ningún tipo.

Entre un 25% y 35% del tiempo se invertirán en otras actividades (imprevistos, llamadas, correo, etc.). Asimismo, es bueno recordar que el trabajo se expande hasta ocupar todo el tiempo del que se disponía para su realización.

Estos dos aspectos significan que si realizamos una estimación por arriba (es decir que indicamos 3 días a una tarea que se realiza en 2 días) por lo general los equipos van a utilizar todo el tiempo del que disponen. Y si estimamos por debajo (decimos 3 días reales para una tarea de 3 días ideales) tendremos retrasos.

LA VELOCIDAD



La velocidad es la **cantidad de trabajo que un equipo puede hacer en una iteración**. En el inicio de la iteración el equipo estima el trabajo que será medido tomando un sistema de referencia de puntos, llamados puntos de sistema. Éstos serán arbitrarios, y pueden tomar numerosas unidades: historias de usuario, tareas técnicas, etc.

Importante: La desviación entre la velocidad planificada y real puede ser grande al principio y se necesitará de varias iteraciones para lograr su estabilización ya que es, **en principio, una medida arbitraria que se obtiene empíricamente**.

En este campo de la gestión ágil toman importancia los diagramas burndown. Son una representación gráfica del trabajo por hacer, en un proyecto, en el tiempo. Usualmente el Sprint Backlog se muestra en el eje vertical y los días hábiles de los que se compone el Sprint en el eje horizontal. El gráfico se actualiza diariamente, por lo que el diagrama representa el trabajo pendiente y el realizado hasta el momento, útil para predecir las desviaciones.

La importancia de esta medida es su consistencia, ya que se utilizará para estimar en la siguiente iteración. Es decir, es una herramienta de motivación para ser igual o más productivos en las siguientes iteraciones.

En el siguiente ejemplo la línea roja muestra la realidad, mientras que la línea azul muestra la estimación que se realizó al inicio.

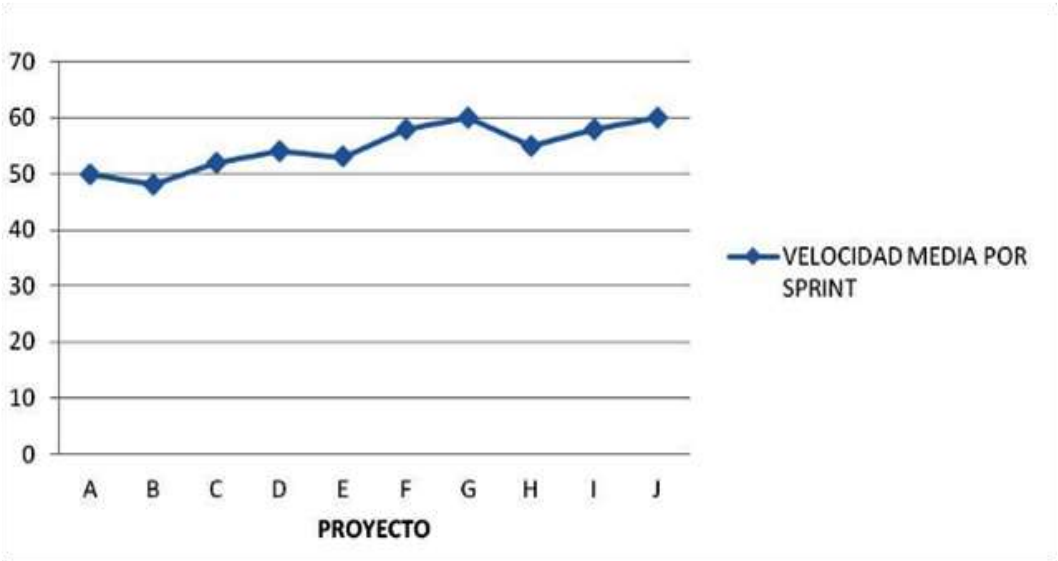
HISTÓRICO DEL EQUIPO

Ajustando las estimaciones

Ahora imaginemos que en el proceso de estimación nuestro equipo ha realizado una estimación poco realista. Esto se puede dar por diferentes causas, por ejemplo, nuestro equipo de desarrollo está compuesto por 3 personas y los 3 tenían un día "optimista".

Es decir, es necesario tener en cuenta la historia del equipo y la velocidad de desarrollo para comprobar si las estimaciones han sido demasiado optimistas o pesimistas.

En el siguiente ejemplo, vemos un histórico de la velocidad media por iteración de nuestro equipo. Para este caso la velocidad vendrá determinada por el número medio de puntos historia que ha realizado nuestro equipo en las iteraciones de un proyecto. En el eje vertical se encuentra la cantidad de puntos historia desarrollados. En el eje horizontal se indican los proyectos realizados.



Esto nos sirve desde dos puntos de vista:

- **Antes del inicio de una estimación**: para que el equipo conozca su capacidad
- **Después de realizar la estimación**: para que se controle que tan bien o mal se ha realizado la estimación.

Importante: Hemos dicho que las estimaciones suelen ser optimistas. Por lo tanto, una vez se van afianzando las técnicas de estimación **la "estimación por debajo" disminuye**.

De esta manera, como puede verse en el gráfico, al inicio del proyecto las estimaciones suelen ser dispersas e imprecisas. Sin embargo, según avanza el proyecto, a través del histórico, la introducción de nuevas prácticas de estimación y la experiencia conseguida en las mismas se tiende a una mejora en las estimaciones futuras.

Estimando el número de iteraciones necesarias para el proyecto

Si sumamos los puntos historia de todas las historias de usuario a desarrollar tendremos una estimación del tamaño total del proyecto. Y si conocemos la velocidad del equipo, por

datos históricos, podremos dividir el tamaño entre la velocidad para llegar a una estimación del número de iteraciones necesarias.

Por ejemplo, supongamos que todas las historias de usuario se han estimado y que la suma de esas estimaciones es de 100 puntos historia. En base a la experiencia previa sabemos que la velocidad media del equipo es de 11 puntos historia por iteración de dos semanas. Ya que 100/11 = 9,1 se puede estimar que el proyecto necesita 9,1 iteraciones, con un enfoque conservador redondeando a 10 iteraciones. Dado que cada iteración es de dos semanas, nuestra estimación de la duración es de veinte semanas.

En cualquier caso, no olvides que cuando hablamos de velocidad hacemos referencia a una media. Si necesitas hacer estudios más profundos sobre la velocidad de un equipo, o comparar más fielmente a dos equipos según su velocidad, etc., deberías acompañar la velocidad media de otros valores, por ejemplo, la desviación típica.

CUÁNTO DEBE DURAR UNA ITERACIÓN

Un punto clave a la hora de planificar un proyecto iterativo es decidir la duración de las iteraciones (o de los sprint en terminología Scrum). En la siguiente figura se puede ver un ejemplo con iteraciones de 3 semanas de duración.

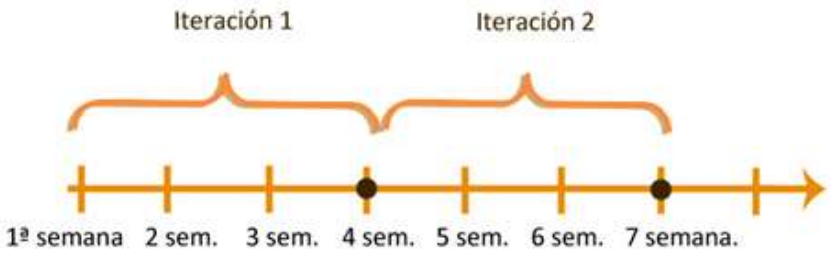


Figura 14 Ejemplo de planificación de un proyecto iterativo con iteraciones de 3 semanas.

> *Ten cuidado, que en este punto la terminología puede ser confusa.*
>
> *Una iteración, es un periodo de tiempo, no confundir con el ciclo de vida iterativo.*

Para seleccionar la duración de una iteración, podemos encontrar recomendaciones de todo tipo. Por ejemplo, metodologías como **XP** recomiendan iteraciones de **un par de semanas**, mientras que lo habitual en **Scrum** es que sean de **un mes de duración**. Y lo normal que nos solemos encontrar en proyectos son iteraciones que están entre una semana y un mes.

¿Qué debería determinar la duración más recomendable para una iteración? Sin que exista una norma exacta, los siguientes factores pueden servirnos de ayuda:

- **La frecuencia con la que hay que mostrar el software a los usuarios**. Normalmente, el software se puede mostrar al final de una iteración (demos de prototipos), por lo que, a mayor frecuencia requerida para mostrar demos y avance, menor debiera ser la duración de la iteración.

- En línea con el anterior, debiéramos pensar **con qué frecuencia hay que medir, o mostrar, el progreso del proyecto**. En teoría sólo al final de una iteración podemos medir con precisión la cantidad de trabajo que realmente ha sido completado.
- **La frecuencia con la que se pueden re-ajustar objetivos del proyecto**. No debiéramos hacer cambios de objetivo, funcionalidad, o alcance, una vez que ha comenzado una iteración. Los ajustes y cambios deben esperar hasta el momento en que una iteración termina. Por ello, si se requiere mayor ratio de cambio de alcance, menor debiera ser la duración de la iteración. Por lo tanto, el tiempo que se puede aguantar sin cambios de prioridad es un factor determinante a la hora de fijar la temporalidad de una iteración.

Con todo, por ejemplo, si tenemos cuatro meses de proyecto, y nuestras iteraciones son de un mes de duración, tendremos tres momentos (al final de la iteración 1, 2 y 3) para tomar “feedback”, medir progreso y re-ajustar prioridades.

Otra consideración clave es el tiempo que tarda una idea (un requisito funcional, una historia de usuario) en transformarse en software. Por ejemplo, consideremos de nuevo iteraciones de cuatro semanas. Suponiendo que una nueva idea aparece en el medio de una iteración, esa idea solo puede introducirse en la próxima versión, que comenzará en dos semanas. Dos semanas que quedan de la iteración en que aparece el nuevo requisito, más cuatro de la siguiente, hacen que la nueva idea esté implementada en 6 semanas, y si 6 semanas es mucho para nuestro negocio... habrá que considerar iteraciones menores.

Una última consideración. No olvides que a menor tiempo de iteración mayor nivel y sofisticación debe tener el equipo de desarrollo, por lo que la madurez, compenetración, experiencia, cualidades técnicas, etc., juegan un papel importante.

> *A iteraciones más cortas, harás más entregas, con mayor frecuencia, y menor será la desviación respecto a lo que el usuario espera.*

¿Deberían las iteraciones durar el mismo tiempo?

En nuestra experiencia, hemos visto equipos trabajar muy bien con iteraciones, o sprint, de duración variable, es decir, que justo antes de comenzar el mismo, en su planificación, se decidía la duración que tendría la iteración.

Es recomendable que todas las iteraciones duren lo mismo, esto sincroniza a la organización, permite hacer cálculos de velocidad más exactos y detectar problemas en el proyecto.

Pero conviene decir que han sido más los equipos que trabajan bien con iteraciones de la misma duración. Es más, en equipos poco experimentados, variar la temporalidad de la iteración suele conducir a problemas.

Capítulo 5
LEAN Y KANBAN

INTRODUCCIÓN

El término "Lean" o "Lean Manufacturing" (cuya traducción sería fabricación esbelta) es otro término que, al igual que el Kanban (como ya veremos más adelante), tiene su origen en Toyota. De hecho, "Lean" es sinónimo de Toyota Production System, una estrategia de fabricación aplicada con mucho éxito en Japón y ahora muy famosa en el mundo del software, muchas veces bajo el término de Lean Software Development.

En los 50 la industria japonesa estaba recuperándose de la segunda guerra mundial y logró con gran éxito aplicar a sus fábricas de coches los conceptos de calidad en la producción creados por los principales gurús estadounidenses, de entre los que destaca Deming. La paradoja fue que siendo métodos idealmente originados por estadounidenses fueron aplicados por los japoneses, convirtiendo a Japón en líder en la industria automovilística, pasando por encima de los EEUU.

El artífice de Lean, quien introdujo esta nueva manera de fabricar en Toyota, fue Taiichi Ohno (1912 – 1990), cuya estrategia se fundamentó en tres bases:

- Construir sólo lo necesario.
- Eliminar todo aquello que no añade valor.
- Parar si algo no va bien (lo que está relacionado con el principio de cero defectos).

Además, conviene destacar que el Lean incluye siete importantes principios, que son los siguientes:

- Eliminar desperdicios (eliminating waste)
- Amplificar el aprendizaje (amplifying learning)
- Decidir lo más tarde posible (decide as late as possible)
- Entregar lo más rápido posible (delivering as fast as possible)
- Capacitar y potenciar al equipo (empowering the team)
- Construir con calidad (build quality in)
- Ver el todo (seeing the hole)

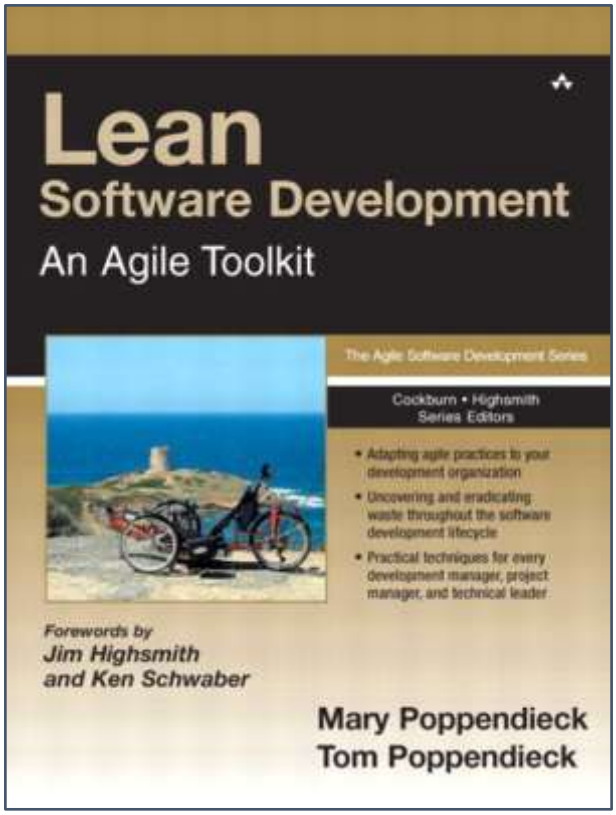
LEAN SOFTWARE DEVELOPMENT

En el mundo del software, fue el libro "Microsoft Secrets" el que primero habló de la similitud de la estrategia de "daily builds" que usaba Microsoft (similar a la integración continua pero planificada, en la que diariamente se compila el software y se pasan algunas pruebas unitarias automatizadas, o el "smoke test"), donde los ingenieros software tenían que parar y corregir errores diariamente y la filosofía de producción de Toyota – JIT, donde los trabajadores paraban las líneas de montaje cuando detectaban problemas, solucionándolos inmediatamente.

El libro "Microsoft Secrets" no utilizó el término "lean", ni lean software development, aunque hablaba de la aplicación de JIT, del control de calidad, y la reducción de la burocracia en Microsoft.

Hay claramente muchos elementos comunes entre estilo el de producción de Toyota, el estilo iterativo de Microsoft y el desarrollo ágil (teniendo en cuenta que en Microsoft no siempre se desarrolló así, a finales de 1990 y principios de 2000 trabajaban con equipos de desarrollo exageradamente grandes).

La popularización del término **"lean" aplicado al software, el lean software development, y su asociación a lo "ágil"** aparece principalmente con el libro "Lean Software Development" de **Mary y Tom Poppendieck**. Estos también hacen hincapié en la eliminación de desperdicios, eliminar la burocracia en el desarrollo de productos, fomentarse el aprendizaje por ciclos cortos y frecuentes, iteraciones rápidas, etc., obteniendo una rápida retroalimentación que "tire" (pull) del producto, en lugar de documentos de requisitos y planes rígidos que "empujen" (push) el trabajo de desarrollo. Y la diferencia entre métodos antiguos y más basados en mano de obra, burocráticos, "pushstyle", inicialmente asociados con el negocio de las computadoras mainframe.



El lean, y el lean software development, no es una metodología de ingeniería de software en el sentido convencional. Es más, una síntesis de principios y una filosofía para construir sistemas de software. Si lean se considera un conjunto de principios más que prácticas, la aplicación de conceptos lean al desarrollo software y la ingeniería software tiene más sentido y puede ayudar a mejorar la calidad.

AGILIDAD Y LEAN NO SON EXACTAMENTE LO MISMO

El desarrollo ágil es un paraguas que incluye varias metodologías (Scrum, XP, FDD, etc.). Todas ellas tienen en común que siguen en mayor o menor medida los valores y principios del manifiesto ágil. Y hay incluso quien afirma que las ideas de Lean y las ideas agiles son tan similares que se dice que aplicar la filosofía ágil es aplicar la filosofía Lean. Y un proceso Lean, es un proceso ágil.

La conexión de lo ágil con el Lean viene de que muchos creadores de métodos ágiles estuvieron influenciados por los métodos Lean, como por ejemplo Mary y Tom Poppendieck.

Mary, esposa de Tom, trabajó en una fábrica que usaba el método Lean, y Tom era un desarrollador software. De ahí que Mary y Tom Poppendieck sean los pioneros en la aplicación del Lean al software. Y que escribieran el libro que ha inspirado las ideas del Lean aplicado al desarrollo software (Lean Software Development).

Una paradoja más: aunque la filosofía ágil rechaza que el proceso de desarrollo software sea un proceso de fabricación industrial tradicional (como el que sucede al construir un coche o una casa) los ágilistas han tenido una gran influencia y adopción de los métodos de fabricación de Toyota.

Hoy agilidad y lean han llegado casi a ser sinónimos. Los principios ágiles son compatibles con los principios lean. Sin embargo, los principios lean son de mayor alcance, aplican a la hora de seleccionar prácticas de desarrollo apropiadas a otras situaciones, más allá del desarrollo software, y más allá de los entornos en los que la agilidad funciona bien.

A continuación, te enumero las principales diferencias entre agilidad y lean:

1. El lean software development se ve al desarrollo software como un paso dentro de un flujo global, un paso dentro de un todo. Lean no sólo trata el desarrollo, trata hasta la puesta, y éxito, en el mercado del producto.
2. En lean no hay "product owner" o roles de cliente, típicos del desarrollo ágil.
3. En lean existen figuras cercanas a "jefes de proyecto" o "product managers". En el lean software development los equipos están dirigidos por alguien que ocupa el rol de ingeniero jefe (como se llamaba en Toyota), el gerente de programas o "program manager" (Microsoft), o "product champion" (3M).
4. Otra diferencia entre agilidad y lean es que en lean no hay un rol que priorice el trabajo del equipo de desarrollo, como suele suceder en las metodologías ágiles. Los métodos ágiles suelen proponer al cliente o representante del cliente (product owner) que priorice el trabajo del equipo software.
5. Muchas veces respecto a las metodologías ágiles existen interrogantes sobre cómo hacer frente al diseño. Las iteraciones hacen complejo este punto. Dado que el desarrollo lean establece una serie de principios que exigen tratar al producto como un todo, un ciclo de vida completo, enfoque multifuncional, ponen más énfasis en cómo organizar una combinación de diseño, desarrollo, implementación y la validación.

DESPERDICIOS DE LEAN

El Lean, y el Lean Software Development (como ya vimos), se centra en la **eliminación continua de desperdicios**, los desperdicios del lean software son todo aquello que no aporta valor y que ocurre tanto en todo proyecto.

Aunque el Lean se ha aplicado históricamente a la industria, ha sido en los últimos años cuando se ha adaptado al software, donde las cosas no son exactamente iguales (recuerda aquello de que fabricar software no es lo mismo que fabricar coches o casas). Y en esta sección vamos a ver los desperdicios del lean software, **desperdicios reales de día a día de un proyecto software**.

Ya veréis que los desperdicios del lean software pueden aplicar a tu día a día, a cualquier cosa que hagas en tu vida, desde estudiar para un examen, a mantener un blog, a mejorar tu vida social.

Hay 7 típicos desperdicios del lean software (según los clasificaron Mary and Tom Poppendieck en el mejor libro que hay sobre el tema):

1. **Desperdicios del lean software 1: Trabajo realizado parcialmente**

   Aquella documentación que tardamos meses en elaborar pero que quedó sin codificar (esto me recuerda a mis tiempos haciendo diseños UML al detalle y que luego al chocar con la realidad no los seguíamos). Los requisitos, y las historias de usuario obsoletas.

   Ese código en el PC de un desarrollador durante mucho tiempo, tanto que integrarlo ya es complicadísimo es un desperdicio. O esas ramas de código en la herramienta de control de versiones que viven durante meses antes de "mergearse" con la línea principal, desperdicio.

   O el código no probado: sin pruebas, no hay manera de saber que el código funciona (ni con pruebas tenemos seguridad, pues imagínate sin ellas).

2. **Desperdicios del lean software 2: Características extra**

   Aquello que creemos que el cliente va a necesitar pero que cliente no ha pedido. Cuando un comercial, cliente, etc., insiste en incluir cosas en el producto, que al final no se usan tenemos un enorme desperdicio. La funcionalidad que ha quedado obsoleta es un desperdicio.

   Y no olvidéis que las características introducidas sólo para probar la última moda y esa tecnología moderna acaban siendo un desperdicio.

3. **Desperdicios del lean software 3: Reaprendizaje**

   ¿Cuántas veces has resuelto un problema y no has implantado inmediatamente la solución? ¿Cuántas veces has tenido que redescubrir semanas después la misma solución? Desperdicio de tiempo, otro de los desperdicios del lean software.

   Por otro lado, muchas veces no preguntamos, ni hacemos caso a expertos, y trabajamos en solitario. Cada minuto que pasamos redescubriendo cosas que rápidamente nos podrían haber contado, es tiempo perdido.

Y también está el código mal escrito o indocumentado que conduce a una incalculable cantidad de reaprendizaje, desperdiciando tiempo.

Por último, otro clásico y repetido desperdicio: desarrolladores que están constantemente siendo reasignados a funciones diferentes, dejando sin terminar su trabajo actual.

4. **Desperdicios del lean software 4: De mano en mano**

   Documentos de análisis de requisitos, que luego pasan a las manos de un diseñador. El diseñador elabora un diseño y entonces pasa a manos de los programadores. O lo que viene a ser el ciclo de vida en "cascada". Por desgracia, poco conocimiento puede transmitirse SOLO con documentos y diagramas, por lo que mucho papel acaba siendo un desperdicio.

5. **Desperdicios del lean software 5: Las pausas**

   Empezar a trabajar en el desarrollo de un proyecto mucho tiempo después del contacto inicial con el cliente. Esperar a que el equipo esté disponible para empezar a trabajar. Esas largas fases de documentación de requisitos (esto también me recuerda a mis tiempos haciendo diseños UML al detalle) que tratan de especificar de forma exhaustiva todos los aspectos de un proyecto.

   Y no sólo eso, la revisión o aprobación de procesos que actúan como puertas entre fases y que requieren de un individuo apenas disponible.

6. **Desperdicios del lean software 6: Cambio de Tarea**

   El coste de cambiar de tarea durante el desarrollo de software ha sido un problema de siempre. Tom DeMarco y Timothy Lister ya lo trataban en su libro Peopleware (muy recomendable, por cierto).

   Desgraciadamente, concentrarse es extremadamente difícil. De esto hay mucho escrito, pero se dice que necesitamos por lo menos quince minutos para concentrarnos en algo. Y durante esos quince minutos, estar concentrándose. Una vez concentrado, cualquier interrupción nos obliga a empezar de nuevo, lo que cuesta otros quince minutos de tiempo improductivo. Cuatro interrupciones cuestan una hora de productividad. Treinta y dos interrupciones un día.

7. **Desperdicios del lean software 7: Defectos**

   Por último, quizás uno de los más obvios: los defectos. Todo aquello que no se hace bien, es un desperdicio, no aporta valor, y consume tiempo a la hora de repararlo.

KANBAN

Kanban es una palabra japonesa que significa "tarjetas visuales" (kan significa visual, y ban tarjeta). Esta técnica se creó en Toyota, y se utiliza para controlar el avance del trabajo, en el contexto de una línea de producción. Actualmente está siendo aplicado en la gestión de proyectos software.

Las tres reglas de este método son las siguientes:

A su vez hay una regla 0 o inicial: el trabajo que se gestione estará dividido en bloques o tareas pequeñas. Como ejemplos de bloques o ítems a gestionar tenemos: los requisitos, las historias de usuario, una funcionalidad a ser desarrollada, etc.

A continuación, detallaremos las 3 reglas comentadas en la figura anterior:

Regla 1. Visualizar los estados:

Es muy importante analizar los estados del trabajo en nuestro proyecto. Todo proyecto consiste en una serie de fases para ir de una especificación hacia un producto que satisfaga la especificación. Cada fase nos acerca más al producto software. Y una de las formas de visualizar estas fases es implementando tableros, muros o pizarras con información visual para mejorar la comunicación en el equipo Kanban.

A continuación, representamos el ejemplo más sencillo de pizarra donde se muestran las diferentes fases. El espacio azul claro en cada columna es el espacio dejado para trabajar con cada ítem. Por lo tanto, existirán ítems "por hacer", ítems "en progreso" e ítems "hechos".

Y este es un ejemplo real de pizarra Kanban en una empresa de desarrollo software. Cada post-it representa un ítem.

Regla 2. Limitar el trabajo en progreso:

Una vez que tenemos identificados las posibles fases de las tareas de desarrollo software en el proyecto es necesario limitar el trabajo que puede realizarse en cada fase. Para ello, Kanban indica que sólo se debe empezar con algo nuevo cuando un bloque de trabajo o ítem anterior haya sido terminado.

En Kanban se limita el trabajo en progreso mediante el "WIP" (Work in Progress). Al incluir un número de WIP en alguna columna de nuestro tablero, estaremos indicando que en esa columna no se pueden añadir (y por lo tanto trabajar) más tareas de las indicadas por él.

Se tratará en mayor profundidad este tema a lo largo de la lección.

Regla 3. Medir los flujos de trabajo:

La tercera regla de Kanban intenta contestar a la pregunta ¿Cuánto tiempo ha pasado desde que el ítem ha ingresado en el ciclo de vida del proyecto hasta que se ha obtenido el resultado final? Si lo trasladamos a una gestión de proyectos software la pregunta sería: ¿Cuánto tiempo ha pasado desde que se ha especificado el requisito y se ha obtenido la funcionalidad en producción?

Debido a que la regla 2 limita la cantidad de trabajo se pueden dar casos de cuellos de botella, como, por ejemplo, haciendo que el equipo de desarrollo espere a que el equipo de pruebas termine con la funcionalidad anterior. Por ello, es necesario medir el flujo de trabajo y realizar mejoras en los límites del trabajo en progreso para cada fase. Mediante estas reglas Kanban consigue optimizar la capacidad de producción del equipo de desarrollo. Se tratará en mayor profundidad este tema a lo largo de la lección.

En todo momento lo que busca Kanban es una capacidad de producción ideal que maximice la productividad del equipo y la calidad del producto software entregado.

Normalmente, cuando estamos en tecnología, la unidad de trabajo del equipo o "bloque de trabajo", es la historia de usuario o las tareas. Un ejemplo sencillo de pizarra Kanban utilizada en Scrum puede ser el siguiente:

EJEMPLO I: FLUJO DE KANBAN

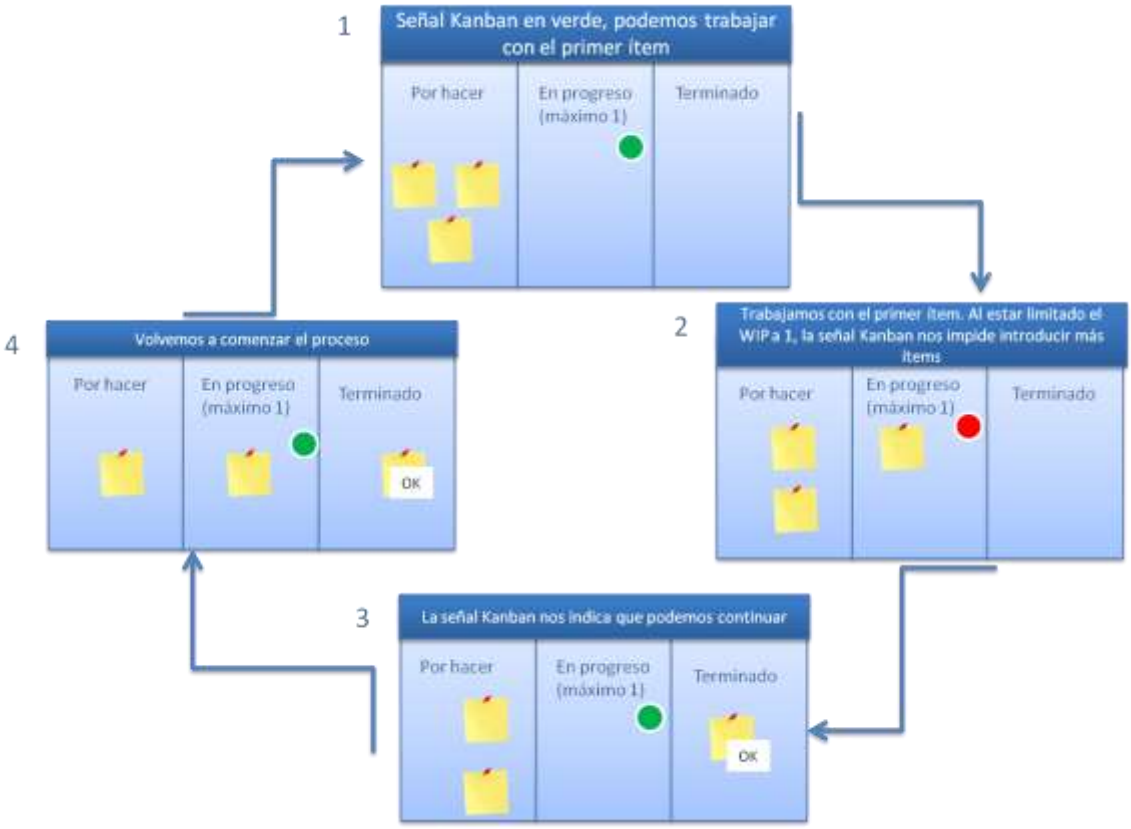
En el siguiente ejemplo vemos una representación simplificada de Kanban y cómo se limita el trabajo en cada fase a partir del WIP.

El jefe de proyecto de nuestro equipo de desarrollo ha implementado un flujo de trabajo representado con 3 fases. Éstos son "Por hacer", "En Progreso" y "Terminado". Además, el número "1" de la fase "En Progreso" nos indica que nuestro equipo no podrá trabajar en más de un ítem (en este caso historias de usuario) a la vez.

Principalmente se realiza un ciclo de 4 pasos:

1. **Comenzamos el proceso.** Podemos trabajar con una historia de usuario, ya que el estado del flujo "En Progreso" está libre.
2. **Trabajamos con la primera historia de usuario.** Al estar limitado el WIP a 1, no podemos introducir más historias de usuario, hasta que hayamos finalizado con la tarea actual.
3. **Una vez terminada la primera historia de usuario**, podemos incluir más historias de usuario de la fase "Por hacer".
4. Volvemos a comenzar el proceso.

Gráficamente representado a continuación:



Todo ello ayudará en la gestión del desarrollo ya que, entre otras cosas, mostrará los cuellos de botella, colas, variabilidad y pérdidas de eficiencia. Esto lo veremos más adelante.

EJEMPLO II: KANBAN Y UN SPRINT BACKLOG

Vamos a ver otro ejemplo de la aplicación de Kanban.

1- En el Sprint backlog hay 4 tareas, de las cuales pasamos a introducir dos tareas a progreso:

2- Dos días después habremos terminado la tarea 3.1:

3- Al quedar un hueco libre, se puede introducir la tarea 3.2:

4- Un día después termina la tarea 1, y podemos introducir directamente la tarea 4 (simplificamos en un paso)

5- El día 5 nuestro equipo termina 3.2:

6- Y finalizado el día 6 termina la iteración:

EJEMPLO III: KANBAN CON ELEMENTOS DE SCRUM

Para afianzar los conceptos y ver claramente las diferencias con Scrum vamos a continuar el ejercicio de la lección anterior. A continuación, se facilita el enunciado:

Supongamos que tenemos un equipo de desarrollo software el cual usa Kanban. Nuestro rol es de jefe de proyecto y un cliente nos ha pedido que desarrollemos una potente calculadora para móviles.

El cliente nos ha trasladado las historias de usuario (user stories). Nuestro equipo ya las ha agrupado por valor (dado por el cliente) y también ha estimado el tiempo que tardaremos, en días, en completarlas. El cliente desea la implementación de:

ID	REQUISITO	VALOR PERCIBIDO	DURACIÓN
1	Operaciones básicas (suma, resta, logaritmo, etc.)	100	3 Días
2	La representación gráfica de las funciones	50	7 Días
3	Resolución de ecuaciones	60	5 Días
4	Operaciones con matrices	55	3 Días
5	Una interfaz gráfica amigable para el usuario	20	2 Días

El primer paso que debemos hacer es dividir las historias de usuario en tareas de un tiempo homogéneo:

ID	REQUISITO	VALOR PERCIBIDO	DURACIÓN
1	Operaciones básicas (suma, resta, logaritmo, etc.)	100	3 Días
2	La representación gráfica de las funciones	50	7 Días
2.1	Estudio de funciones		2 Días
2.2	Dibujo Gráfico de funciones		3 Días
2.3	Pruebas		2 Días
3	Resolución de ecuaciones	60	5 Días
3.1	Investigación de algoritmos		2 Días
3.2	Implementación de algoritmos		3 Días
4	Operaciones con matrices	55	3 Días
5	Una interfaz gráfica amigable para el usuario	20	2 Días

Para formar el Sprint Backlog y simplificar el problema, supondremos que las historias de usuario son indivisibles. Además, las tareas a implementar en cada historia de usuario tienen que guardar el orden establecido.

Esto es: si se quiere implementar el requisito 3, ha de realizarse 3.1 y 3.2 en la misma iteración, y en ese orden.

Este es el muro Kanban que tenemos inicialmente

Según el muro del ejemplo vemos que:

- En el Sprint Backlog lo tenemos que cumplimentar con 4 tareas para este Sprint. Aquí se ve claramente que Kanban limita el WIP por estado en flujo de trabajo (columna del tablero); en cambio Scrum limita el flujo de trabajo por iteración, al indicar en el Sprint Backlog cuántas tareas se pueden realizar en esa iteración (recordemos que una vez comenzada una iteración en Scrum, no debería modificarse el Sprint Backlog).
- En la fase de "En progreso" podrán estar dos tareas desarrollándose paralelamente, indicado por el número 2.

  Ahora bien, tenemos que seleccionar las tareas que, para estas historias de usuario, maximizan el valor dado por el cliente (suponiendo que las historias de usuario son indivisibles).

  Para el primer Sprint, las historias de usuario que maximizan el valor percibido por el cliente son 1, 3 y 4. Por lo que nuestro muro inicial quedaría así:

De esta manera podemos ver que es fácil estimar en Kanban. Son tareas más o menos homogéneas en tiempo. Por lo que, realizando unos pequeños cálculos podemos

determinar nuestro tiempo estimado. En este caso, como el WIP de estado del flujo "En progreso" es 2, el tiempo máximo vendrá determinado por el promedio de tiempo de todas las tareas dividido 2 (ver fórmula).

Por tanto:

> 3 días (de la tarea 1) + 2 días (de la tarea 3.1) + 3 días (de la tarea 3.2) + 3 días (de la tarea 4)
>
> 2 (WIP "En progreso")

El resultado de esta operación es aproximadamente 6 días. Por lo tanto, para terminar las historias de usuario, el equipo tardará aproximadamente 6 días.

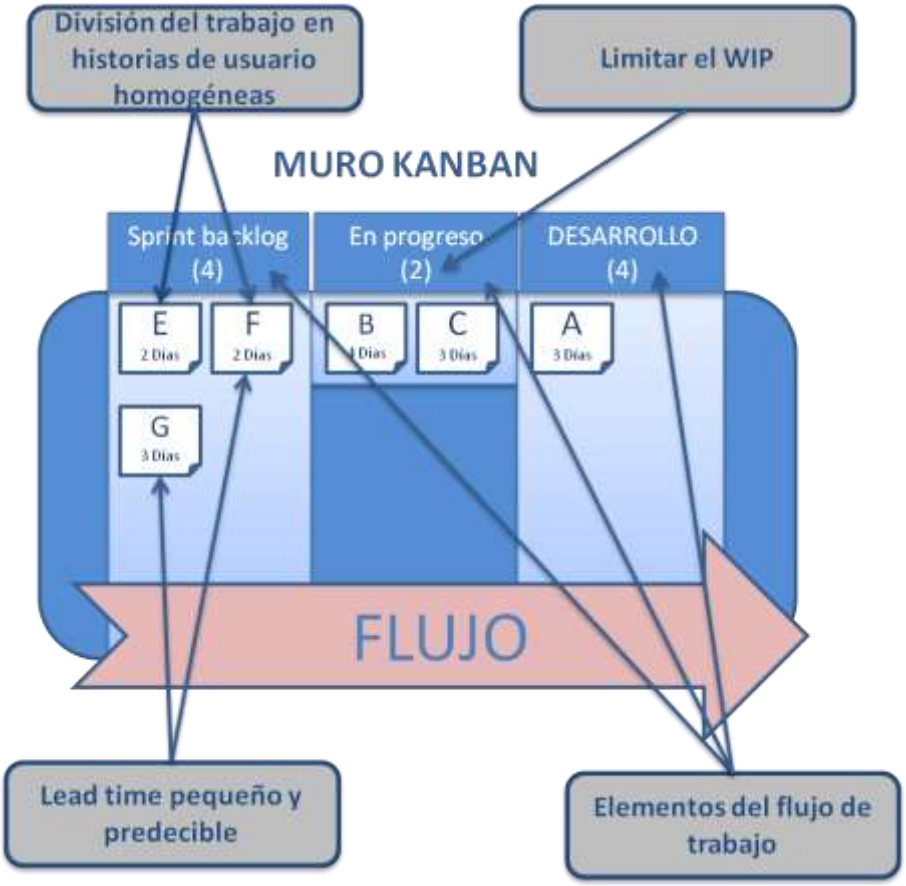
MIDIENDO EL FLUJO DE TRABAJO EN KANBAN

Las dos métricas más importantes para medir el flujo de trabajo en Kanban son el lead time y el cycle time.

¿Qué es el lead time? Para verlo de forma gráfica, el lead time es el tiempo que ha pasado desde que el ítem (historia de usuario, tarea, etc.) ha entrado en la pizarra Kanban hasta que ha pasado a la última fase (terminado, en producción, etc.). Este es el tiempo más importante desde el punto de vista del cliente quien es el que recibe el producto final que satisface sus especificaciones:

Asimismo, existen otros tiempos incluidos dentro del lead time. El cycle time, que es el tiempo que pasa desde que se empieza a trabajar con el ítem hasta que pasa a un estado finalizado. Es decir, es el tiempo que se ha estado trabajando sobre la historia de usuario, tarea, etc. Como se ve en la figura, se intentará hacer ambos tiempos similares mediante la evolución de los límites y de la misma pizarra Kanban.

- **Limitar el WIP- asignando límites concretos.** Determinando cuántos elementos pueden estar en progreso en cada estado del flujo de trabajo. A esto se añade otra idea tan razonable como que para empezar con una nueva tarea alguna otra tarea previa debe haber finalizado.



LOS CUELLOS DE BOTELLA Y EL WIP

Recordemos que en Kanban, el WIP ("Work In Progess") es el número máximo de tareas que pueden realizarse en cada fase del ciclo de trabajo.

Si una columna de un tablero Kanban (es decir, una fase del ciclo de producción) tiene, por ejemplo, un WIP 5, esto quiere decir que si el equipo (ojo, equipo, no las personas, porque el WIP de un Kanban afecta a una parte del proceso, en la que participan n personas) en esa fase está trabajando en cinco tareas no podrán trabajar en ninguna otra hasta que terminen alguna de estas cinco.

Un recurrido ejemplo que ilustra el WIP de un KANBAN es el de la impresora. El WIP de una impresora es 1, sólo se imprime una página a la vez. Así, si una página se atasca, y no se puede imprimir, se da la alarma inmediatamente, y se para el proceso, en vez de comenzar a imprimir otra hoja, lo que complicaría más el problema.

Aunque esto del WIP de un Kanban pueda parecer algo muy simple, su ajuste tiene un gran impacto en la productividad de un equipo. Y saber ajustar el mejor WIP en cada fase del proceso productivo no es una tarea fácil.

Por ejemplo, supongamos que utilizamos la pizarra descrita en el siguiente párrafo y existe un problema en las pruebas. En ese caso en algún momento se llega al máximo dado por el WIP en la etapa de desarrollo y el equipo no puede seguir trabajando.

En ese caso tenemos un cuello de botella en la etapa de pruebas por lo que los equipos de desarrollo pueden ayudar. De hecho, podían haberlo hecho antes, en el caso de que algún integrante del equipo quedara ocioso. Por lo tanto, si el equipo o el desarrollador no tiene posibilidad de continuar trabajando debe buscar el cuello de botella en las siguientes etapas.

Durante la utilización de Kanban los límites del trabajo en progreso (WIP) irán evolucionando. Para ello es indispensable tener en cuenta los síntomas de límites erróneos para converger a los valores óptimos.

¿Qué suele suceder cuando el WIP de un Kanban es muy bajo?

Si el WIP es muy bajo (imaginemos un WIP 1 en una de las fases para un equipo de 4 personas), las tareas se realizarán rápido (ya que podría tener hasta 4 personas trabajando sobre una tarea) pero probablemente tendré miembros del equipo ociosos (ya que normalmente no todas las personas podrán trabajar a la vez sobre la misma tarea).

Además, las tareas estarán en estado de "pendientes de empezar a ser realizadas" demasiado tiempo, en espera de que el equipo comience a trabajar sobre ellas. Y el "lead time" (una métrica Kanban que mide desde que se hace una petición hasta que se realiza la entrega) será mayor de lo necesario.

Un WIP ajustado, a la baja, hará que problemas que pudieran bloquear la finalización de una tarea se resuelvan lo más pronto posible, ya que hasta que no se resuelvan no se podrá comenzar a trabajar en otra cosa (filosofía Lean "soluciona los problemas cuanto antes").

A menor WIP mayor colaboración del equipo, más tareas se harán por más de una persona, teniendo en cuenta que hay un límite de personas que pueden trabajar a la vez sobre una misma tarea. Por ello, otro propósito de limitar el WIP es evitar el exceso de multitareas y la sobrecarga de una parte del proceso.

La esencia de los limites WIP de un KANBAN es enfocar al equipo en finalizar cosas, más que en comenzar cosas.

¿Qué suele suceder cuando el WIP de un Kanban es muy alto?

En contraposición, si el WIP de un KANBAN es muy alto (imaginemos un WIP 8 para un equipo de 4 personas) existe el riesgo de empezar muchas tareas y terminar pocas, de estar trabajando en varias cosas a la vez sin cerrar muchos temas.

Frente a cualquier contratiempo en una tarea, el equipo puede optar por comenzar con otra y demorar el terminar aquellas que presenten contratiempos.

> *En este sentido se observa cómo ajustar el límite WIP de un Kanban actúa como una señal de alerta, avisando de un problema antes de que se nos escape de las manos y siga avanzando (esto es filosofía Lean "soluciona los problemas cuanto antes"), ya que el equipo se centrará en resolver problemas que impiden cerrar tareas, ya que hasta que no lo hagan no podrán comenzar a trabajar en otras.*

¿Cómo saber el WIP de un KANBAN correcto?

Pues como te puedes imaginar no hay una regla exacta que lo calcule. Y además de depender del equipo, del producto que se desarrolle, etc., para un mismo equipo el WIP suele variar con el tiempo, lo que implica un ajuste y mejora continua (también en la filosofía Lean Kaizen).

Pero sí que hay algunas heurísticas a observar, para detectar si tenemos algún problema con nuestro WIP, por ejemplo:

- Si en una fase del ciclo de producción se observa que hay tareas sobre las que nadie trabaja durante mucho tiempo es porque el WIP de esa fase probablemente es alto. Esto también lo puedes observar en que el "lead time" será alto.
- Si en una fase hay gente parada durante mucho tiempo el WIP de un KANBAN probablemente es bajo. La productividad será baja. Y esto ocurre porque no todo el equipo puede trabajar a la vez sobre una tarea.

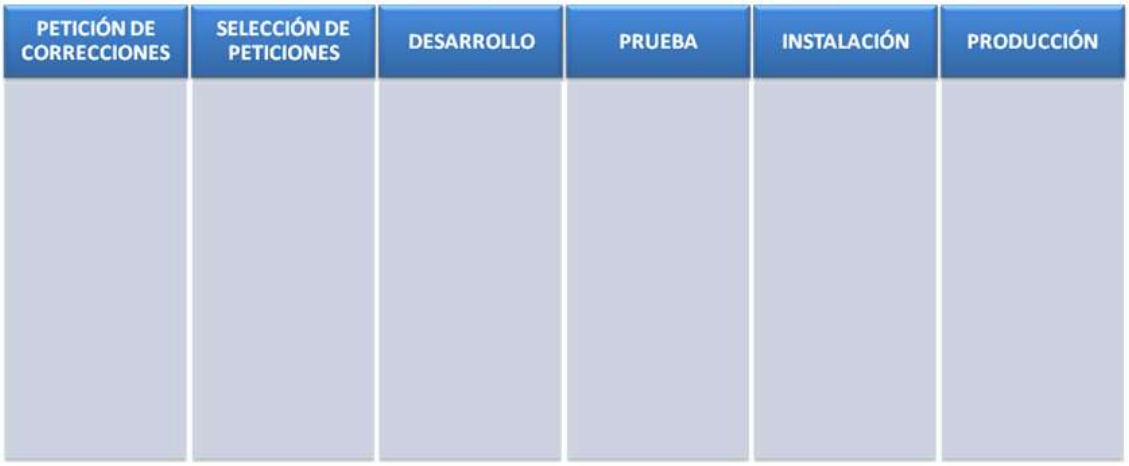
Otra cuestión es con qué WIP comenzar la primera vez. Hay quien usa la regla de 2n-1, donde n es el número de miembros del equipo, y el -1 se aplica para aumentar el grado de colaboración. Pero no es más que una regla, que no tiene mayor base que algo de sentido común. Al final, cada equipo debe buscar su WIP.

UNA PIZARRA MÁS COMPLEJA

Los ejemplos anteriores se han centrado en la parte operativa general. Hemos comentado que uno de los primeros pasos al implementar Kanban en la organización, en un equipo de desarrollo, de mantenimiento o de soporte es definir la pizarra.

Por lo tanto, será necesario definir las columnas de la pizarra de Kanban. Estas podrán ser de dos tipos: columnas de estado y columnas de espera o de buffer.

Para definir las columnas es indispensable pensar en el proceso actual que queremos organizar mediante Kanban. Por ejemplo, si quisiéramos organizar el desarrollo de parches para un producto software, se podrían incluir las siguientes columnas:



El flujo de trabajo es el siguiente: llega una petición de parche y el jefe de proyecto la selecciona enviándola a la columna de "peticiones seleccionadas". En el siguiente paso el equipo de desarrollo elegirá el siguiente parche a desarrollar. Una vez el parche se encuentre desarrollado quedará a la espera de ser elegido para realizar la actividad de pruebas. Por lo tanto, entre la la etapa de desarrollo y prueba puede existir un tiempo de espera.

Por eso es necesario dividir la columna de desarrollo y la columna de pruebas en dos partes, una para el trabajo que se está realizando (columnas de estado) y otra para el trabajo ya realizado (columnas de espera).

Lo más importante aquí es que el límite WIP es de la suma de ambas columnas. Es decir que si la columna "desarrollo" tiene un límite WIP de 4 no se podrá empezar nuevas tareas de desarrollo, aunque tenga los 4 ítems en la sub-columna "hechos". Esto es para evitar los cuellos de botella.

Recomendación: empezar de menos a más en cantidad de columnas. Como hemos visto se puede complicar fácilmente.