

Gestión Ágil de Proyectos Software

Javier Garzás

Juan Enríquez de S.

Emanuel Irrazábal



Prólogo de Mario Piattini

Ediciones



Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

Gestión Ágil de Proyectos Software

Javier Garzás Parra

Juan A. Enríquez de S.

Emanuel Irrazábal

De la edición: Kybele Consulting

MARCAS COMERCIALES: las marcas de los productos citados en el contenido de este libro (sean o no marcas registradas) pertenecen a sus respectivos propietarios. Kybele Consulting no está asociada a ningún producto o fabricante mencionado en la obra, los datos y los ejemplos utilizados son ficticios salvo que se indique lo contrario.

Kybele Consulting es marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa.

Sin embargo, Kybele Consulting no asume ninguna responsabilidad derivada de su uso, ni tampoco por cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene como objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa ni de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma. Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro, sin autorización previa y por escrito de Kybele Consulting: su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes intencionadamente reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

Editado por:

Kybele Consulting
C./ Francisco de Sales 2, P3, 1ºB, Villanueva del Pardillo, Madrid
Teléfono 91 815 40 24
Correo Electrónico: info@kybeleconsulting.com
Internet: www.kybeleconsulting.com
ISBN: 978-84-615-9003-2
Version: 1.1

A Eva - Javier

A Álex, mi hermano y a M^a Teresa y Ángel, mis padres - Juan

A Ramón Ismael y María del Carmen - Emanuel

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

Testimonios de algunos lectores

"Un gran esfuerzo de síntesis. Muy útil para JPs, CEOs, etc. que se inicien en este mundo"

Ignacio Cruzado

"Javier, Juan y Emanuel han afrontado con éxito el reto de escribir en español un libro necesario para conocer y entender la gestión ágil de proyectos desde la perspectiva de la gestión de empresa, la gestión de proyectos y el desarrollo de producto. Sabe a poco y ya estoy deseando que afronten el reto de nuevas publicaciones. ¡Enhorabuena!"

Ángel Águeda Barrero (socio - director de Evergreen PM)

"Según la RAE, "valor", en su primera acepción, significa "Grado de utilidad o aptitud de las cosas, para satisfacer las necesidades o proporcionar bienestar o deleite". En este libro se nos dan unas pautas para generar software con una alto grado de utilidad para nuestros clientes y que, realmente, se acerque a sus necesidades generando una sensación real de bienestar en su uso. "

Pedro García Repetto (Ministerio de Hacienda)

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

Índice de contenidos

1.	<i>Érase una vez un proyecto ágil</i>	1
1.1	La predictibilidad, ciclo de vida en cascada o desarrollo tradicional	2
1.2	Construir software no es como construir coches o casas	4
1.3	Frente a la predicción... adaptación, o el ciclo de vida iterativo e incremental	8
1.4	El proyecto ágil	12
2.	<i>El Manifiesto Ágil</i>	15
2.1	Los inicios del Manifiesto Ágil	15
2.2	Valores del Manifiesto Ágil	18
2.3	Los principios ágiles	20
2.4	Malas interpretaciones sobre el Manifiesto Ágil	21
3.	<i>Las historias de usuario</i>	25
3.1	Qué información contiene una historia de usuario	28
3.2	Malas interpretaciones del concepto de historia de usuario	30
3.2.1	Una historia de usuario no es una especificación de requisitos	30
3.2.2	Una historia de usuario no es un caso de uso	32
3.2.3	Una historia de usuario no es una tarea	34

Índice

3.3	Historias técnicas: otro tipo de historia de usuario	38
3.4	La calidad de una historia de usuario	38
3.5	Las dependencias entre historias de usuario	40
4.	<i>Scrum</i>	41
4.1	El equipo en Scrum	43
4.2	El Product Backlog	46
4.3	El Sprint	48
4.4	Las reuniones	52
4.5	Medir el progreso del proyecto	53
4.6	Beneficios de Scrum	55
5.	<i>Planificación y estimación ágil</i>	57
5.1	Asignar valor a las historias de usuario	58
5.2	Estimación de las historias de usuario	61
5.2.1	Algunos conceptos sobre la estimación software	61
5.2.2	La estimación de una historia de usuario	64
5.2.3	Peligros al estimar	70
5.3	Priorización de las historias de usuario	71
6.	<i>Lean Software Development y Kanban</i>	73
6.1	El Lean y los métodos ágiles: Lean Software Development	75

6.2 Kanban	77
6.2.1 Regla 1: Visualizar los estados	78
6.2.2 Regla 2: Limitar el trabajo en progreso	80
6.2.3 Regla 3: Medir los flujos de trabajo	83
6.3 Ejemplo de flujo de desarrollo en Kanban	84
6.4 Definición de un tablero más complejo	86
6.4.1 Las columnas de espera (buffer)	87
6.5 Kanban y los cuellos de botella	88
6.5.1 Teoría de las restricciones en Kanban	89
6.6 Comparando Kanban y Scrum	90
6.6.1 Scrum-ban	94
6.6.2 Trabajar varios proyectos en Scrum y Kanban	95
7. Métricas y seguimiento de proyectos ágiles	97
7.1 Tipos de indicadores	99
7.2 Indicadores de productividad	100
7.2.1 La velocidad	100
7.2.2 La aceleración	103
7.3 Indicadores de progreso de proyecto	105
7.3.1 Gráficos tipo BurnUp	105
7.3.2 Funcionalidades probadas y aceptadas (Running Tested Features, RTF)	106
7.4 Indicadores de valor entregado y retorno de inversión	109

Índice

7.4.1	EVM-Ágil	109
7.5	Gestión de riesgos: Impediment Backlog	113
7.6	Cuadro de mando ágil	114
7.7	Diagramas de Gantt en proyectos ágiles	116
7.8	Métricas asociadas al tablero Kanban	117
8.	<i>EXTreme Programming</i>	119
8.1	Las buenas prácticas de XP	120
8.2	El ciclo de vida de XP	125
8.3	Algunos datos sobre el uso de XP	129
9.	<i>Buenas prácticas de desarrollo para proyectos ágiles</i>	131
9.1	Integración continua y “builds” automáticos	132
9.1.1	Herramientas de integración continua	137
9.2	Pruebas en entornos ágiles	139
9.2.1	Pruebas unitarias	140
9.2.2	Pruebas de humo	143
9.2.3	Pruebas unitarias y pruebas de humo en proyectos ágiles	145
9.3	Refactorización	148
9.4	Gestión de la configuración software	151
9.4.1	Modelo de GCS aplicado con éxito en proyectos ágiles	152

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)	163
10.1 Modelos de procesos (CMMI e ISO/IEC 12207) y las prácticas ágiles	163
10.2 Relación entre Scrum y las áreas de proceso CMMI-DEV	166
10.3 Conclusiones	173
11. Contratos ágiles	175
11.1 Los principales tipos de contratos en entornos ágiles	176
11.1.1 Pago por tiempo o por iteración	176
11.1.2 Pago por punto función o punto historia	177
11.1.3 El contrato cerrado o fijo en proyectos ágiles	177
11.2 Los riesgos que asume cada parte en un contrato	179
11.3 Otros aspectos a tener en cuenta en un contrato	180
12. Los proyectos ágiles y el desarrollo global del software	183
12.1 Metodologías ágiles y GSD	185
12.2 Problemas y soluciones al implementar el desarrollo global de software con prácticas ágiles	186
13. La implantación de prácticas ágiles en la empresa	189
13.1 Primera etapa: conocimiento	190

Índice

13.2 ¿Qué debemos saber antes de aplicar metodologías ágiles? _____	191
13.3 Factores de éxito _____	192
13.4 Segunda etapa: aplicación _____	194
13.5 Tácticas de implementación _____	195
13.6 Tercera etapa: evaluación _____	196
13.7 Conclusiones y consideraciones sobre la implantación _____	197
A. <i>Herramientas para la gestión ágil</i> _____	199
A.1 Herramientas más usadas _____	200
A.2 Herramientas generales de gestión y seguimiento de proyectos _____	202
<i>Referencias</i> _____	215

Sobre los autores



JAVIER GARZÁS PARRA

(javier.garzas@urjc.es,
javier.garzas@kybeleconsulting.com)

Doctor (Ph.D.) (cum laude por unanimidad) e Ingeniero en Informática (premio extraordinario). Cursó estudios postdoctorales y fue investigador invitado en la Universidad Carnegie Mellon (Pittsburgh, Pennsylvania, EE.UU). Actualmente trabaja en la empresa Kybele Consulting S.L. (empresa spin off del grupo de investigación de la Universidad Rey Juan Carlos), es profesor Titular en la Universidad Rey Juan Carlos y edita el blog www.javiergarzas.com.

Cuenta con certificaciones de Auditor Jefe de TICs (Calificado por AENOR) para ISO 15504 SPICE – ISO 12207, Auditor ISO 20000 por ITSMF, Especialización en Enterprise Application Integration (premiado por Pricewaterhousecopers), CISA (Certified Information Systems Auditor), CGEIT (Certified in the Governance of Enterprise IT) y CRISC (Certified in Risk and Information Systems Control) por la ISACA, CSQE (Software Quality Engineer Certification) por la ASQ (American Society for Quality), Introduction CMMI-Dev y Acquisition Supplement for CMMI v1.2 (CMMI-ACQ) e ITIL V3 Foundation.

Actualmente es socio-director de KYBELE CONSULTING, liderando varios proyectos en administraciones y empresas como INFORMÁTICA DE LA COMUNIDAD DE MADRID (ICM), RENFE, DIRECCIÓN GENERAL DE TRÁFICO (DGT), MINISTERIO DE ADMINISTRACIONES PÚBLICAS, SISTEMAS TÉCNICOS DE LOTERÍAS (STL), AENOR, etc.

Sobre los autores

Comenzó su carrera profesional como consultor senior y responsable del centro de competencias en ingeniería del software, desde donde participa en proyectos para TELEFÓNICA MÓVILES CORPORACIÓN, INDRA tráfico aéreo o la automatización de la simulación de la rotativa de EL MUNDO. Más tarde fue responsable de calidad software y de proyectos de mCENTRIC. Posteriormente, DIRECTOR EJECUTIVO Y DE INFORMÁTICA de la empresa de desarrollo de ERPs para la gestión universitaria con mayor número de clientes en España. Experto en gestión y dirección de departamentos y fábricas software (realizando implantaciones de fábricas y mejoras en España, Colombia, Chile y Venezuela), con una amplia experiencia en ingeniería del software, calidad y mejora de procesos (participación en la mejora, evaluación o auditoría de procesos CMMI o ISO 15504 en más de 30 empresas).

Ha participado en numerosos proyectos de I+D nacionales e internacionales, ponencias, editado varios libros (destacando el primer libro en castellano sobre fábricas software) y publicado más de 60 trabajos de investigación. Evaluador de la ANEP (Agencia Nacional de Evaluación y Prospectiva) y experto certificado por AENOR y EQA para la valoración de proyectos I+D.



JUAN ÁNGEL ENRÍQUEZ DE SALAMANCA DE LA FUENTE

(juan.enriquez@kybeleconsulting.com)

Ingeniero en Informática por la Escuela Superior de Informática de Ciudad Real de la Universidad de Castilla-La Mancha. Actualmente trabaja en la empresa Kybele Consulting S.L. (empresa spin off del grupo de investigación Kybele de la Universidad Rey Juan Carlos). CISA (Certified Information Systems Auditor) por la ISACA.

Desde 2008 es CONSULTOR SENIOR de KYBELE CONSULTING, desarrollando funciones de consultor de calidad software. Ha participado en proyectos de control de la calidad software (proceso de aseguramiento de la calidad) para organizaciones y administraciones públicas como SISTEMAS TÉCNICOS DE LOTERÍAS (STL) y ha contribuido en el desarrollo de la plataforma KEMIS, proyecto financiado por el CDTI (IDI-20090175). Colabora a su vez con el GRUPO DE INVESTIGACIÓN KYBELE de la UNIVERSIDAD REY JUAN CARLOS desarrollando su investigación en el campo de la calidad software.

Desde septiembre de 2006 hasta mayo de 2008 trabajó en el GRUPO DE INVESTIGACIÓN ALARCOS de la UNIVERSIDAD DE CASTILLA - LA MANCHA elaborando funciones de diseño, desarrollo y mantenimiento de aplicaciones Web relacionadas con la medición de la calidad de datos y la visibilidad de los portales Web en diferentes tecnologías.

Posee varios artículos en revistas y conferencias bajo el área.

Sobre los autores



EMANUEL IRRAZÁBAL

(emanuel.irrazabal@urjc.es,
emanuel.irrazabal@kybeleconsulting.com)

Doctor (Ph.D.) (cum laude por unanimidad) e Ingeniero en Informática y Master oficial en Tecnologías de la Información y Sistemas Informáticos. Escuela Superior de Ingeniería Informática – Universidad Rey Juan Carlos. Mención de Calidad MEC.

Auditor Jefe Norma ISO 15504 por AENOR. CISA (Certified Information Systems Auditor) por la ISACA. Acreditado como Tecnólogo por el Programa Torres Quevedo.

Desde 2009 es CONSULTOR SENIOR de KYBELE CONSULTING, trabajando en varios proyectos en administraciones y empresas como SISTEMAS TÉCNICOS DE LOTERÍAS (STL), AENOR, SIEMENS, NEORIS, etc. y ha contribuido en el desarrollo de la plataforma KEMIS, proyecto financiado por el CDTI (IDI-20090175). Colabora a su vez con el GRUPO DE INVESTIGACIÓN KYBELE de la UNIVERSIDAD REY JUAN CARLOS desarrollando su investigación en el campo de la calidad software.

Desde abril 2008 hasta noviembre 2008 trabajando como responsable de pruebas en una empresa de desarrollo para la banca, implementando integración continua en su proceso de desarrollo (herramientas de Business Communication, análisis estático de código, pruebas unitarias, de stress y de aceptación). También ha trabajado como analista desarrollador y analista funcional de Web Applications con .Net 2005, Scrum, Test Driven Development, pruebas unitarias, funcionales, e integración continua para España y Estados Unidos.

Actualmente es también profesor asociado de la UNIVERSIDAD REY JUAN CARLOS y miembro del SC7/GT24 de AENOR.

Sobre los autores

Destacados

- Más de 20 proyectos de mejora y/o evaluación con CMMI y/o ISO 15504.
- Mejora de fábricas software en España, PYMES y Grandes Empresas.
- Artículos en revistas y conferencias bajo el área.
- Investigador del grupo de investigación KYBELE en la Universidad Rey Juan Carlos desde 2008.

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

Prólogo

El software juega un papel crucial en la sociedad, no sólo porque nuestra forma de vida depende cada vez más de su buen funcionamiento, sino también por la importancia cada vez mayor que tiene el macrosector de las TIC (Tecnologías de la Información y las Comunicaciones) en el PIB (Producto Interior Bruto), el crecimiento económico y el empleo tanto de los países desarrollados como de los emergentes.

Por ello no es de extrañar que a lo largo de sus más de sesenta años, la Ingeniería del Software haya propuesto diferentes técnicas y métodos para facilitar el desarrollo de software, que como señala Grady Booch: “*ha sido, es y probablemente será fundamentalmente difícil*”. Otro de los mayores expertos en este campo, Barry Boehm, destaca cómo la evolución de la Ingeniería del Software ha seguido un proceso de tesis, antítesis y síntesis que explicaría las diferentes propuestas y contrapropuestas que se han sucedido a lo largo de estas décadas.

Siguiendo este razonamiento, podríamos decir que si en la década de los cincuenta el desarrollo de software era “riguroso” ya que se llevaba a cabo como el del hardware (prácticamente “conectando cables”), en los sesenta se volvió mucho más “ligero” ya que, probablemente abusando de la maleabilidad que ofrece el *software*, se construyeron sistemas de manera muy flexible, quizás demasiado; dando lugar a los famosos códigos espaguetis y a los “superhéroes” que terminaban arreglando los problemas después de varias noches sin descanso. En los setenta, con la publicación del ciclo de vida en cascada por parte de Winston Royce, se vuelve a la “rigurosidad”, facilitando la gestión “formal” de los proyectos de software. Este enfoque se ha comparado muchas veces con utilizar una “lista de la compra” a la hora de ir al supermercado, al final a veces no compramos cosas que están en la lista porque no nos convencen, o bien compramos varias cosas que no estaban previstas... Aunque es cierto que si

Prólogo

hacemos la compra “mediante catálogo” (por correo, teléfono, o Internet) nos entregan exactamente lo que pensamos que necesitamos...

En los ochenta, como reacción al ciclo de vida en cascada, Barry Boehm propone el modelo en espiral, estableciendo un ciclo iterativo, que como señalan los autores de este libro, ya se utilizaba en otros ámbitos de la ingeniería. Este enfoque resultó muy útil en proyectos internos (en los que los desarrolladores y los clientes pertenecen a la misma organización), pero no tuvo el mismo éxito en proyectos con clientes externos que aún en la actualidad siguen prefiriendo pensar que tienen el proceso más “controlado” usando un ciclo de vida en cascada. Esta sensación de control por parte de los clientes (en muchos casos más ilusoria que real) se consolidará en la década de los noventa con la implantación de los procesos propuestos en los modelos de madurez de capacidad (CMM, CMMI) cuyos fundamentos publicó Watts Humphrey en 1989.

En la década de los 2000 se difunden los métodos ágiles (el “Manifiesto” es del año 2001), como reacción a los métodos “pesados” de la década anterior, y sobre todo a una manera excesivamente “burocrática” de entender el desarrollo de software, en el que primaba más cumplir “formalmente” con las especificaciones y entregar voluminosos documentos que “demostrarán” el trabajo realizado, que conseguir un software útil al cliente. De hecho, la calidad del software que se empezó interpretando como el estricto “cumplimiento de los requisitos” ha terminado cambiando su enfoque al de la “satisfacción del cliente”, que resulta de responder a sus necesidades y expectativas.

Esta obra nos propone que adoptemos un enfoque ágil para la gestión de proyectos, que seguro contribuirá a mejorar la calidad de los mismos y a conseguir clientes más satisfechos. Hay que tener en cuenta que en la actualidad el “paradigma ágil” ya ha madurado en todos los planos: en el científico, ya que se ha desarrollado una importante investigación sobre las mejores prácticas y métodos del desarrollo ágil; industrial, ya que disponemos de productos tanto

de software libre como propietario que ayudan a implantar las prácticas ágiles; y comercial, ya que éstas están siendo cada vez más utilizadas por los desarrolladores de software, incluso en grandes proyectos globales. Además, el enfoque ágil aporta un aspecto muy importante en el desarrollo de software, permite “empoderar” a los miembros del grupo de desarrollo, convertirles en un verdadero “equipo” e involucrarles junto con los “clientes” en los proyectos; y recordemos que a la postre la “P” más importante en nuestro campo, no es la de “Proyecto”, “Plataforma”, “Proceso” o “Producto”, sino la de “Persona”, ya que el desarrollo y la evolución de software siguen siendo muy intensivos en mano de obra y todo lo que suponga motivar al personal se reflejará de forma evidente en el resultado obtenido.

En esta década, de los años 2010, ya se están empleando métodos que sintetizan lo mejor de los enfoques “pesados” tradicionales y de los ágiles, buscando un equilibrio entre el seguimiento de un proceso de desarrollo muy “burocrático” y la inexistencia del mismo, combinando la adaptabilidad de las aproximaciones ágiles con la formalidad y documentación de los métodos rigurosos. De hecho, igual que los autores proponen, con mucho acierto, combinar las técnicas ágiles con los enfoques Lean o Kanban; si el lector está utilizando (con éxito) técnicas como el modelo entidad/interrelación, los diagramas de flujo, los modelos de clases, etc. será bueno que los integre en su metodología de desarrollo a la vez que se dota de una gestión de proyectos más ágil.

Esta obra le permitirá comprender de manera práctica y rigurosa las principales técnicas y, quizás lo más importante, la “filosofía” de la gestión ágil de proyectos, que sin duda le resultará útil en la gestión de sus proyectos, no sólo de sistemas de información. De hecho, en mi grupo de investigación la hemos utilizado con éxito para proyectos de I+D e incluso para la certificación de un laboratorio de calidad.

En cuanto a los autores, a Javier Garzás tuve la fortuna de tenerle como alumno, no sólo de la Ingeniería Informática sino también de doctorado, por lo

Prólogo

que conozco muy bien su capacidad de trabajo y de innovación. Además le admiro por ser una de las pocas personas que siendo profesor de universidad, también tienen espíritu emprendedor y empresario, como demuestra el éxito de Kybele Consulting durante estos años, con la que tengo la fortuna de poder colaborar en proyectos muy interesantes. La Universidad española debería impulsar más este tipo de perfiles, si quiere tener un lugar relevante en la sociedad y devolver, al menos en parte, lo que la sociedad le aporta. A Juan Enríquez de Salamanca también tuve la oportunidad de tenerlo como alumno de la Escuela Superior de Informática de Ciudad Real y como tecnólogo en el Grupo de Investigación Alarcos, y me alegra ver la prometedora carrera profesional que está desarrollando. De Emanuel Irrazábal, originario de la provincia de Corrientes (Argentina) en la que viví –como la serie de televisión de los noventa- *aquellos maravillosos años* de mi niñez, también he podido constatar en estos últimos años su profesionalidad. A los tres quería agradecerles haberme invitado a prologar esta obra, que me hubiese gustado poder escribirla yo pero... últimamente sólo me da tiempo para escribir los prólogos de buenos libros como éste.

Por último, parafraseando al pintor argentino Raúl Soldi, podemos decir que “**las tecnologías de desarrollo de software progresan, pero la Ingeniería del Software evoluciona**”, y en este sentido las técnicas ágiles ofrecen a los departamentos o factorías de desarrollo de software una magnífica oportunidad para evolucionar y responder así al reto de conseguir una mayor calidad de los sistemas de información y, sobre todo, que las cuantiosas inversiones en software aporten el correspondiente valor a las organizaciones.

Ciudad Real, Junio de 2012
Mario Piattini Velthuis
Instituto de Tecnologías y Sistemas de Información
Universidad de Castilla-La Mancha

Prefacio

"La transición de un paradigma en crisis a otro nuevo del que pueda surgir una nueva tradición de ciencia normal, está lejos de ser un proceso de acumulación, al que se llegue por medio de una articulación o una ampliación del antiguo paradigma. Es más bien una reconstrucción del campo, a partir de nuevos fundamentos, reconstrucción que cambia alguna de las generalizaciones teóricas más elementales del campo, así como también muchos de los métodos y aplicaciones del paradigma. Cuando la transición es completa, la profesión habrá modificado su visión del campo, sus métodos y sus metas."

Thomas S. Kuhn

The Structure of Scientific Revolutions (1962)

Las metodologías ágiles se han convertido hoy en día en una de las principales estrategias de desarrollo software. Tanto ha sido su auge en los últimos años que se han extendido, y han sido adoptadas, por otras áreas de la gestión de proyectos no estrictamente relacionadas con el software. Y aún les queda camino por recorrer, sobre todo después de haber encontrado la unión con la filosofía Lean y ciertos métodos de producción industrial, perfectos compañeros de viaje para seguir evolucionando en nuevas técnicas de desarrollo y gestión del software.

Prefacio

Sin embargo tan rápido crecimiento no ha sido gratuito, y ha llevado consigo cierta confusión, malinterpretación e incluso efectos negativos en proyectos software. Aspectos menos positivos, que aún se perciben en el día a día de los proyectos software.

En los últimos años se han publicado multitud de libros y artículos sobre agilidad, los cuales exponen los principios y las buenas prácticas propuestas por la filosofía ágil. Si bien son muy pocos los que están escritos en castellano, y los que tienen como objetivo y enfoque ofrecer una visión general de la agilidad, que permita fácilmente introducir a los profesionales no tan familiarizados con la misma.

La presente obra ofrece una visión amplia sobre diferentes factores que se deben tener en consideración para la construcción de software de calidad. Se persiguen los siguientes objetivos:

- Presentar de forma clara los conceptos fundamentales relacionados con la agilidad.
- Dar a conocer las principales metodologías y técnicas.
- Tratar aspectos muy importantes para gestionar correctamente un proyecto ágil, la calidad de la información o la gestión del conocimiento.

La selección de los temas incluidos en esta obra se ha basado sobre todo en la experiencia práctica, combinada con rigor científico, proporcionando una panorámica actual y completa sobre problemáticas y soluciones asociadas a la gestión ágil.

CONTENIDO

La obra consta de 13 capítulos y un anexo en el cual presentamos un resumen de las principales herramientas para la gestión ágil. El primer capítulo introduce las principales características de un proyecto de desarrollo software ágil. En el capítulo 2 se analizan los valores y principios ágiles del Manifiesto Ágil.

A lo largo del capítulo 3 se detallan las historias de usuario, sus características y sus diferencias respecto a los requisitos o los casos de uso. Los capítulos 4 y 5 tratan, respectivamente, las prácticas relacionadas con el marco de trabajo Scrum y con la planificación/estimación ágil.

El capítulo 6 recoge la relación entre el Lean Software Development y los métodos ágiles, detallando las reglas de la técnica Kanban y su relación con Scrum; mientras que el capítulo 7 trata acerca de las métricas y el seguimiento de proyectos ágiles de desarrollo software, indicando tipos de indicadores y cuadros de mando.

Al igual que en el capítulo 4 se detallaban las prácticas de Scrum, en el capítulo 8 se detallan las prácticas relacionadas con la metodología de desarrollo eXtreme Programming. Y en el capítulo 9 se resumen las buenas prácticas de desarrollo relacionadas con proyectos ágiles, como la integración continua, las pruebas unitarias o la refactorización.

En los capítulos 10 y 11 se introducen dos temas directamente relacionados con los proyectos software como es, por un lado la relación entre las metodologías ágiles y los modelos de procesos (como CMMI-DEV o ISO 12207), y por otro lado, los contratos ágiles.

Los siguientes dos capítulos giran alrededor de la implantación de las prácticas ágiles en una organización. En el capítulo 12 se explica la relación de las metodologías ágiles con el desarrollo global de software; y en el capítulo 13 se describen los pasos para la implantación de prácticas ágiles en la empresa.

Prefacio

ORIENTACIÓN A LOS LECTORES

Nuestro propósito al presentar este libro ha sido dirigirnos a una audiencia amplia que comprende:

- Participantes en seminarios o cursos monográficos sobre agilidad.
- Profesionales informáticos que estén trabajando en el área del desarrollo de Sistemas de Información
- Directivos que tengan entre sus responsabilidades el desarrollo y mantenimiento de sistemas.
- Usuarios avanzados, que tengan interés en adquirir conocimientos sobre las técnicas y metodologías más utilizadas para asegurar la calidad de los sistemas de información
- Analistas o consultores que, aun teniendo conocimientos de la materia, quieran abordarla de forma más sistemática.

Debido a la diversidad de la audiencia, el estudio de esta obra puede realizarse de maneras muy distintas dependiendo de la finalidad y conocimientos previos del lector. Si bien recomendamos a todos primero leer el tema 1.

OTRAS OBRAS RELACIONADAS

Aquí os dejamos una breve reseña de otros libros en los que ha participado Kybele Consulting:



Fábricas del Software: Experiencias, tecnologías y organización. 2^a Edición actualizada.

Garzás Parra, Javier / Piattini Velthuis, Mario G.

La presente obra reúne las contribuciones de los mayores especialistas en aspectos relacionados con la fabricación de software, por lo que se ofrece una visión amplia sobre diferentes factores que se deben tener en consideración para la puesta en marcha y la gestión de una fábrica de software. Además, incluye la experiencia práctica de 13

fábricas de software, grandes y pequeñas, de España, Argentina, México y Venezuela.



Medición y estimación del software: técnicas y métodos para mejorar la calidad y la productividad.

García Rubio, Félix Oscar / Garzás Parra, Javier / Genero Bocco, Marcela Fabiana / Piattini Velthuis, Mario G.

Esta obra presenta de forma clara y precisa los conceptos fundamentales sobre la medición de software, ofrece un tratamiento sistemático de los

Prefacio

principales estándares y métodos de medición internacionales, resume las principales métricas (útiles y válidas) existentes en la actualidad, expone en profundidad las principales técnicas de estimación de software y presenta cuestiones relacionadas con el control estadístico de procesos.

SITIOS WEB RELACIONADOS

El mundo de la tecnología se mueve tan rápido que lo escrito pronto queda anticuado, por eso, si quieras mantenerte al día en estos temas te recomendamos seguir:

- www.javiergarzas.com
- www.fabricasdesoftware.es
- www.iso15504.es
- www.kybeleconsulting.com

REDES SOCIALES RECOMENDADAS

- | | |
|---|---|
|  | @calidadsoftware |
|  | Ingeniería del Software |
|  | Calidad en el Software y los Sistemas de Información (CSSI)
ISO 15504.es SPICE - Calidad Software - España y Latinoamérica |

AGRADECIMIENTOS

Nuestro agradecimiento a los asistentes a los diferentes webinars, seminarios y conferencias que hemos organizado sobre diferentes aspectos de la calidad software y proyectos ágiles, especialmente a los participantes del emprendimiento CIO AGIL, por habernos transmitido en qué temas debería incidir el libro. Con sus inquietudes han ayudado a clarificarnos qué debíamos dejar claro y qué debíamos enfatizar en esta obra.

Agradecer también a Kybele Consulting el apoyo en la realización de este libro, en su promoción y edición.

Por otro lado, nos resultaría imposible citar a todas las personas que con sus sugerencias y comentarios han contribuido a que sea posible la realización de este libro. Aún así, queríamos destacar la labor de revisión, las sugerencias y los comentarios realizados por:

- Ignacio Cruzado
- Ángel Águeda Barrero, socio - director de Evergreen PM
- Pedro García Repetto, Ministerio de Hacienda

Por último, queremos dar especialmente las gracias a D. Mario Piattini por sus valiosas sugerencias, y por escribir el prólogo de esta obra.

*Javier Garzás Parra
Juan Á. Enríquez de Salamanca de la Fuente
Emanuel Irrazábal*

Móstoles, Madrid, mayo de 2012

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

1. Érase una vez un proyecto ágil

Capítulo

1

1. Érase una vez un proyecto ágil

“Estamos todavía en la infancia de darle nombre a lo que está sucediendo en los proyectos de desarrollo software” – A. Cockburn

Con el tiempo, muchos profesionales de la ingeniería del software nos hemos dando cuenta de que construir software no siempre sigue los mismos principios de gestión de proyectos que sigue construir cosas físicas como puentes, edificios o cadenas de montaje de coches (Garzá, 2011). Y probablemente muchos proyectos nunca los seguirán.

Aunque, desde hace años, desde su nacimiento, la gestión de proyectos software ha intentado imitar la gestión de proyectos que realizan otras disciplinas, como la arquitectura, las fábricas o la ingeniería civil, hasta el punto de heredar y adaptar al mundo del software muchos de sus roles (p.e. arquitectos *software*), tipos de organizaciones (p.e. fábricas de *software*) y prácticas.

Pero, sin duda, la práctica más controvertida que se ha querido adaptar al software, la más destacada para nuestro contexto, siendo hoy en día una de las más discutidas y polémicas, es la llamada **predictibilidad** (M. Fowler, 2008), también conocida como gestión de proyectos dirigida por la planificación, desarrollo tradicional o incluso también conocida como desarrollo pesado.

1. Érase una vez un proyecto ágil

1.1 La predictibilidad, ciclo de vida en cascada o desarrollo tradicional

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. A este tipo de gestión de proyectos 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 y 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

1. Érase una vez un proyecto ágil

no se modifique; de hecho, notaciones como UML¹ 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** (ver Figura 1).

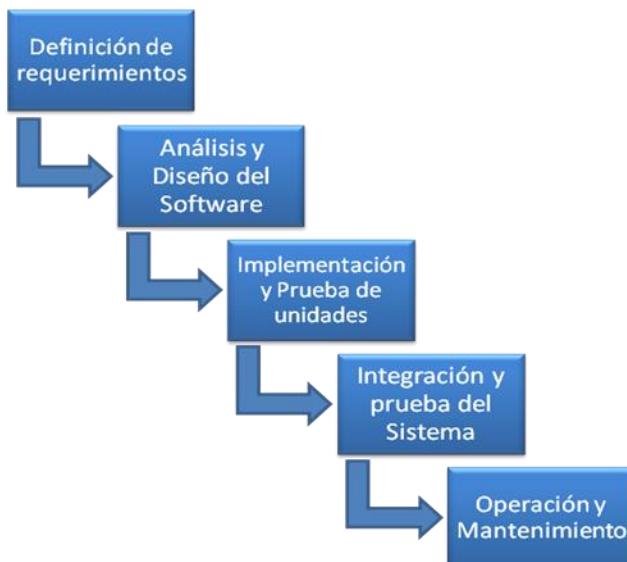


Figura 1. Ejemplo de ciclo de vida predictivo o en cascada

El problema es que en ingeniería del software, a diferencia de la arquitectura u otras ingenierías tradicionales, no se puede separar tan claramente las fases de requisitos, diseño y construcción. Y raramente cada una de estas fases se puede hacer de una única vez. Y es que construir software no sigue exactamente las mismas reglas que construir cosas físicas.

¹ UML, por sus siglas en inglés que significan Unified Modeling Language es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema.

² En este caso se utiliza la palabra prototipo como sinónimo de un producto

1. Érase una vez un proyecto ágil

1.2 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 prototipos².

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

² En este caso se utiliza la palabra prototipo como sinónimo de un producto software con características que pueden ser utilizadas por el cliente

1. Érase una vez un proyecto ágil

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

1. Érase una vez un proyecto ágil

y usos de lenguajes de diseño como UML, o metodologías como RUP³ o Métrica v3⁴.

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.

Ejemplos de errores clásicos en ingeniería del software derivados de pasar por alto las diferencias de crear software con la creación de productos físicos:

- El “mítico hombre-mes” (Brooks, 1975): Añadir a un proyecto ya comenzado y con retraso más programadores para intentar que avance más rápido, de la misma manera que en la construcción se incrementa la mano de obra para acelerar la construcción.
- Medir el avance del proyecto “midiendo” el número de líneas de código. Similar a medir el avance de una construcción según los ladrillos utilizados.

Diferenciar el cómo se construye software del cómo se construyen los productos físicos es uno de los pilares de las metodologías ágiles (M. Fowler, 2005). De hecho es un tema del que se ha escrito mucho .

³ RUP, por sus siglas en inglés que significan Rational Unified Process es un proceso de desarrollo de software utilizado junto con UML y que constituye una metodología para el análisis, diseño, implementación y documentación de proyectos software orientados a objetos.

⁴ Métrica v3 es una metodología de planificación, desarrollo y mantenimiento de sistemas de información, mayormente promovida y utilizada en el ámbito de las administraciones públicas.

1. Érase una vez un proyecto ágil

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

Desde su nacimiento, la ingeniería del software se ha estado preguntando si debería construir el software siguiendo principios similares a los de otras disciplinas como la arquitectura o la fabricación clásica. Y aún hoy no existe una respuesta consensuada. Ya en octubre de 1968, durante la primera conferencia sobre ingeniería software, organizada por la OTAN en Alemania, había debates al respecto:

H.A. Kinslow, hablando sobre el diseño en el software comentaba: “El proceso de diseño es iterativo. [...]. Para mí el diseño consta de:

- Diagramas de flujo hasta que creas que entiendes el problema.
- Escribir código hasta que te das cuenta de que no es así [no entiendes el problema].
- Volver atrás y rehacer el diagrama de flujo.
- Escribir más código e iterar hasta que sientas que estás en la solución correcta.”

Y por su parte, Naur:

“Los diseñadores software están en una posición similar a los arquitectos o ingenieros civiles”

1. Érase una vez un proyecto ágil

1.3 Frente a la predicción... adaptación, o el ciclo de vida iterativo e incremental

Frente a realizar cada fase del proyecto una única vez, una fase tras otra sin, en teoría, vuelta atrás, o en cascada, aparece el ciclo de vida incremental en el que el software se va creando poco a poco, por partes. En vez de una única fase monolítica de requisitos, otra única de diseño y una final de codificación, el software final se construye después de varias secuencias de requisitos, diseño y codificación, cada una creando una parte del software final. Se van desarrollando partes del software, para después integrarlas a medida que se completan. A este ciclo de vida se le conoce como incremental.

Un ejemplo de un desarrollo incremental puede ser la agregación de módulos en diferentes fases. Dividir el software final en partes e ir desarrollando cada una. Gráficamente vemos en la Figura 2 como cada vez se van desarrollando partes, que se van integrando para formar el producto final.



Figura 2. Desarrollo incremental

Pero no sólo el ciclo de vida incremental es el que propone una manera alternativa al desarrollo en cascada. Hay muchas otras propuestas (como el ciclo de vida en espiral), donde una de las más destacadas es el ciclo de vida iterativo. El **desarrollo iterativo** se centra en mejorar y revisar reiterativamente el producto ya creado. En cada ciclo se revisa y mejora el trabajo.

Por ejemplo, cuando se desarrolla rápidamente un producto, muchas veces no se hace con todos los detalles ni la mejor calidad, pero se hace con la idea de que

1. Érase una vez un proyecto ágil

el usuario vea cuanto antes un prototipo, para que en sucesivas iteraciones dicho producto se vaya mejorando y afianzando. En el ejemplo de la siguiente figura se puede apreciar la mejora a través de sucesivas iteraciones. Es importante señalar que en este ciclo de vida no hay adición de funcionalidades en el producto, en cada iteración hay revisión y mejora.

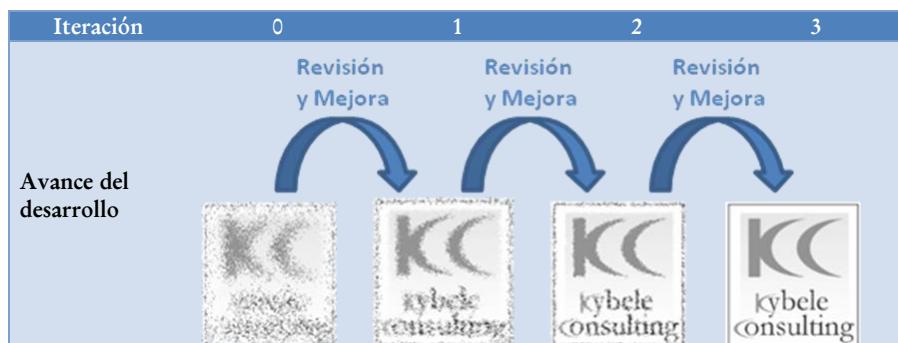


Figura 3. Desarrollo iterativo

De la unión de los dos anteriores nace el ciclo de vida **iterativo e incremental**, que, como veremos, es característico de los proyectos ágiles. En este tipo de ciclo de vida, al final de cada iteración se consigue una versión operativa del software, que añade nuevas funcionalidades y mejoras sobre la versión anterior. Al ir desarrollando prototipos, se obtiene un “feedback” continuo por parte del usuario sobre un producto operativo.

En la Figura 4, a diferencia de las anteriores, se puede observar cómo al producto creado en la primera iteración se le van añadiendo nuevas formas, tonos, colores (mejoras) en cada iteración sucesiva. Además, las nuevas modificaciones se aplican sobre el trabajo creado desde la primera iteración, ya que es una versión del producto final. Por tanto, cada iteración mejora el producto.

1. Érase una vez un proyecto ágil

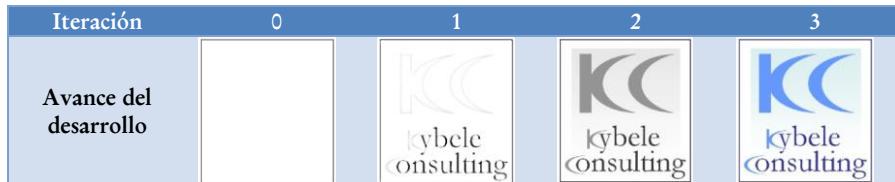


Figura 4. Desarrollo iterativo e incremental

Las ventajas que ofrece un desarrollo iterativo e incremental frente al cascada son varias, dependiendo del contexto en el que se implemente el proceso. Estas ventajas son:

- **Detección de problemas y riesgos** en momentos tempranos del proyecto. Al tener pronto una versión, se puede estudiar antes si el proyecto es viable.
- **Mayor visión del avance del desarrollo**, ya que se obtiene un producto operativo real desde etapas más tempranas del ciclo de desarrollo.
- **Pruebas de aceptación y “feedback” del usuario** sobre un prototipo operativo. De esta manera, se puede orientar el desarrollo hacia el cumplimiento de sus necesidades, y realizar todas las adaptaciones identificadas para cumplir con los objetivos planteados. Evitamos así una fase monolítica de especificación de requisitos.
- **El aprendizaje y experiencia del equipo**, iteración tras iteración, mejora exponencialmente el trabajo, aumenta la productividad y permite optimizar el proceso a corto plazo. El trabajo iterativo deja una experiencia en el equipo que permite ir **ajustando y mejorando las planificaciones**.

CURIOSIDADES: El ciclo de vida iterativo e incremental es incluso ;más antiguo que el cascada! (J. Garzás, 2010).

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).

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. Aunque el proyecto X-15 no era un proyecto exclusivamente de software, es importante mencionarlo porque algunos de los participantes en el mismo (y su experiencia en dicho ciclo de vida) comenzaron a utilizarlo en la NASA en 1960 para el desarrollo software. Concretamente fue en un proyecto llamado Mercury, del que a su vez algunos participantes en el mismo trabajarían después en IBM Federal Systems Division, donde también se aplicó el ciclo de vida iterativo e incremental al desarrollo software. El proyecto Mercury (1960) trabajó con iteraciones diarias, realizando revisiones técnicas a los cambios, y aplicando la técnica de planificar y escribir las pruebas antes de cada micro incremento.

La primera referencia que describe y recomienda el desarrollo iterativo es de 1968, un informe de Brian Randell y F.W. Zurcher que trabajaban en el IBM T.J. Watson Research Center.

1. Érase una vez un proyecto ágil

El ciclo iterativo e incremental difiere del ciclo en **cascada**, principalmente, en que en este último las fases del ciclo de vida se realizan (en teoría) **una única vez**, y el inicio de una fase no comienza hasta que termina la fase que le precede, como se representaba en la Figura 1. Mientras que de manera general, podemos decir que en este tipo de ciclos de vida iterativos e incrementales las fronteras entre cada fase del proyecto son mucho más difusas (no hay, por ejemplo, un punto claro en el que se termina el diseño y empieza la codificación) y hay un alto grado de solapamiento entre las fases (Piattini et al., 2003)

Claro que, si iterar es una buena práctica... ¿Qué sucede cuando las iteraciones se llevan al máximo? ¿Cuándo intentamos crear el máximo número de prototipos? ¿Cuándo intentamos que los tiempos de cada iteración sean mínimos? Que estaríamos empezando a hablar de un proyecto ágil.

1.4 El proyecto ágil

Comentaba Ambler (Ambler, 2008), que 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 autoorganizados. Es decir, un proyecto ágil es un desarrollo iterativo más la segunda gran característica implicada directamente por la iteración extrema: equipos colaborativos y autoorganizados.

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

1. Érase una vez un proyecto ágil

pueden diseñar y programar. E implica autoorganización, 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 autoorganizados, 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.

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

2. El Manifiesto Ágil



2. El Manifiesto Ágil

“Estamos descubriendo mejores maneras de desarrollar software, haciéndolo y ayudando a que otros lo hagan.” – Firmantes del Manifiesto Ágil

El 12 de febrero de 2001, 17 destacados y conocidos profesionales de la ingeniería del software escribían en Utah el Manifiesto Ágil (K. Beck et al., 2001). Entre ellos estaban los creadores de algunas de las metodologías⁵ 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.

2.1 Los inicios del Manifiesto Ágil

Antes de la elaboración del manifiesto, un gran número de grupos de trabajo ya habían desarrollado ideas similares con el objetivo de promover el desarrollo

⁵ El término metodología es utilizado de manera genérica para referirnos a Scrum, siendo este un marco de trabajo tal y como lo indican sus creadores.

2. El Manifiesto Ágil

software mediante ciclos iterativos, con equipos trabajando de manera colaborativa y autoorganizada. Muchas de estas ideas provenían de la comunidad del software orientado a objetos, los cuales apoyaban el enfoque de desarrollo iterativo. El Manifiesto Ágil empezó a gestarse en el año 2000, con la intención de reunir estos diferentes enfoques, aunque todavía no se tenía una palabra común (que posteriormente sería la palabra “ágil”).

En ese mismo año 2000 Kent Beck organizó un taller centrado en su metodología, eXtreme Programming (XP). Uno de los debates surgidos en el taller estaba relacionado con la amplitud del movimiento XP y la posibilidad de transformarse en un movimiento más amplio. Esto dejaba abierta las puertas para un nuevo taller que aunara esfuerzos hacia una iniciativa más amplia.

La idea inicial de este nuevo taller (llamado “Snowbird”) que dio lugar al Manifiesto Ágil, fue la de crear un punto de encuentro para construir una mejor comprensión de los enfoques actuales del desarrollo software y de las nuevas ideas surgidas. Uno de los que promovieron la realización de un manifiesto fue Robert Martin, el cual estaba dispuesto a conseguir un manifiesto que podría ser utilizado para unir a la industria detrás de este tipo de técnicas.

Durante el transcurso de este taller se decidió finalmente utilizar la palabra “ágil” como nombre que unificaba los enfoques de los diecisiete participantes, y se empezó a trabajar en su contenido. Se concretaron los valores ágiles y se inició con la sección de principios, concretada más tarde en una wiki colaborativa. Jim Highsmith describe la elección de la palabra “ágil” en (Highsmith, 2001), donde se pueden encontrar también las notas de la reunión. La única preocupación respecto al término ágil fue por parte de Martin Fowler, de origen británico, quien pensaba que la mayoría de los estadounidenses no sabrían cómo pronunciar la palabra “agile”.

2. El Manifiesto Ágil

Ese mismo año los diecisiete participantes del taller volvieron a encontrarse en el congreso Object-Oriented Programming, Systems, Languages & Applications (OOPSLA). Durante el congreso los autores recalcaron que la motivación del taller que había generado el Manifiesto Ágil era sólo unificar y compartir enfoques. Por lo tanto los creadores del manifiesto no tenían intención de reclamar el liderazgo de la comunidad ágil que se estaba formando como consecuencia del manifiesto. El siguiente paso, con el esfuerzo de muchos de estos autores fue la creación de la Agile Alliance, una organización sin ánimo de lucro, dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a las organizaciones para que adopten dichos conceptos.

Actualmente, el Manifiesto Ágil es uno de los documentos más influyentes de la ingeniería del software y uno de los que más ha influido en la manera de desarrollar. Y aunque muchos de sus consejos y principios son muy antiguos, previos al manifiesto, éste supone una manera accesible y sintetizada de recoger una emergente y efectiva filosofía de desarrollo software. Y que ha dado para cientos de debates, inspiraciones, mejoras del desarrollo, libros, artículos, para rechazar el uso de otras prácticas de ingeniería software (como CMMI⁶ y similares), erróneas justificaciones para por ejemplo, no documentar nada en los proyectos, etc.

El contenido del Manifiesto Ágil se divide en cuatro valores y doce principios. Veamos a continuación una descripción de los valores o postulados del manifiesto ágil (los principios sobre los que se basan los métodos alternativos a los tradicionales, menos rígidos y "pesados").

⁶ CMMI, por sus siglas en inglés que significan Capability maturity model integration, es un modelo para la mejora y evaluación de procesos para el desarrollo, mantenimiento y operación de sistemas de software.

2. El Manifiesto Ágil

2.2 Valores del Manifiesto Ágil

El Manifiesto Ágil⁷ se caracteriza principalmente por sus cuatro valores. Cada uno de estos valores son concretizados de diferente manera por las metodologías ágiles.

- “**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. Así mismo compromete al equipo de desarrollo y a los individuos que lo componen. Como indica Jeff Sutherland en (J. Sutherland, 2010), cuando no existen problemas de comunicación el equipo puede trabajar hasta 50 veces mejor que el promedio de la industria.

Ahora bien, solamente con aumentar la frecuencia de la comunicación o el feedback no se eliminan los problemas de comunicación. El comportamiento del equipo es fundamental y Jeff Sutherland enumera alguna de sus claves (J. Sutherland, 2010): el respeto del trabajo de cada miembro del equipo, la transparencia de la información y las decisiones, compromiso de cada miembro con el equipo y los objetivos del equipo.

- “**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

⁷ En este libro hemos tomado la traducción al castellano que se encuentra en: agilemanifesto.org/iso/es/

2. El Manifiesto Ágil

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.

En este caso también se abre una nueva vía de debate, respecto de la pregunta: ¿Qué significa “software que funciona”? El equipo de desarrollo debe estar de acuerdo en este significado, y por ejemplo, puede incluir que el software supere la ejecución de pruebas o que pueda ser utilizado por un usuario final.

- **“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.

La colaboración diaria con el cliente o un representante es una de las claves por las cuales los proyectos ágiles tienen un ratio de éxito mayor que los proyectos tradicionales.

- **“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. Uno de los objetivos principales de cualquier proyecto ágil es proveer valor al cliente. Y para ello es necesario que el cliente obtenga de forma iterativa pequeñas funcionalidades. De esta manera se pueden producir cambios y mejoras que aumenten el valor retornado al cliente por parte del producto software resultante.

2. El Manifiesto Ágil

CURIOSIDADES

El manifiesto ágil utiliza la palabra “over”

Si repasamos la versión original del Manifiesto Ágil, éste utiliza la palabra "over" que significa "por encima de, sobre", para expresar las preferencias del Manifiesto (p.e "Individuals and interactions over processes and tools" o "Working software over comprehensive documentation"). De hecho, los autores recalcan que "aunque valoramos los elementos de la derecha (de la palabra over), valoramos más los de la izquierda". Por eso usaron la palabra "por encima de" y no "en vez de".

2.3 Los principios ágiles

De los cuatro valores anteriormente descritos en el Manifiesto Ágil surgen los **doce principios del manifiesto**. Mediante estos principios se puede identificar lo que diferencia a un proceso ágil de un proceso tradicional. Los principios, explicados en el Manifiesto Ágil, 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.

2. El Manifiesto Ágil

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.

2.4 Malas interpretaciones sobre el Manifiesto Ágil

A continuación se enumerarán algunas malas interpretaciones que se pueden obtener de la lectura del manifiesto o los principios ágiles.

- **Ausencia total de documentación.** Es necesario documentar de manera ágil, pero documentar. Como indica Bran Selic en (Bran, 2009) "frecuentemente escucho a los desarrolladores decir que no les gusta documentar, que no lo encuentran útil, pero... ¿No era el objetivo principal de documentar el ayudar a otros?". Como se verá en la sección 8.2, pág. 125, algunas metodologías ágiles como eXtreme Programming, o destacados autores como por ejemplo Martin Fowler (M. Fowler, 2008), indican que es necesario realizar un diseño, aunque sea

2. El Manifiesto Ágil

ligero (es decir, documentación), antes de la codificación. Lo importante es que la documentación esté al servicio del desarrollador, del equipo y en definitiva del proyecto. La semántica de los lenguajes de programación está muy lejos de la semántica del dominio de aplicación. La amplitud de esta brecha es la razón por la que todavía se necesitan programadores. Finalmente afirma Bran Selic, "debiera haber documentación de diseño ágil, que mantenga y enlace el diseño y la codificación. Y el esfuerzo realizado en documentar debiera ser proporcional al tamaño del diseño".

- **Ausencia total de planificación.** Planificar y ser flexible es diferente a improvisar. Un equipo que desarrolla con metodologías ágiles poseen técnicas de estimación como por ejemplo, el "Planning Poker" (pág. 66). La planificación tiene una componente "ágil" intrínseca. No se concibe una planificación rígida, o 100% precisa. Y tampoco es su objetivo final. La planificación ayuda a anticipar decisiones y gestionar los riesgos. De hecho, en las principales metodologías ágiles se realizan reuniones de planificación. Se logra una planificación "ágil" incorporando a los clientes, teniendo en cuenta la historia reciente del equipo de desarrollo y utilizando técnicas que favorezcan el aporte de cada punto de vista.
- **El cliente debe hacer todo el trabajo y será el jefe de proyecto.** El cliente entra a formar parte del equipo de desarrollo, pero tiene un rol propio dentro del mismo. Es importante tener en cuenta que en la mayoría de las metodologías ágiles no existe el rol tradicional de "jefe de proyecto". En cambio, los equipos se autogestionan. Este es uno de los desafíos principales a la hora de implantar prácticas ágiles.
- **El equipo puede modificar la metodología sin justificación.** De mezclar culturas organizacionales flexibles o rígidas con proyectos cambiantes o estables, están apareciendo cuatro métodos de desarrollo (Garzás, 2010). Los métodos formales (más apropiados para organizaciones rígidas con proyectos estables) y los métodos ágiles (más apro-

2. El Manifiesto Ágil

piados en organizaciones flexibles con proyectos cambiantes), a los que se añaden los iterativos-formales (más apropiados para organizaciones rígidas con proyectos cambiantes) y los métodos ágiles optimizados o con algo de diseño “up front” (más apropiados en organizaciones flexibles con proyectos estables).

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

3. Las historias de usuario



3. *Las historias de usuario*

“Gran parte de los procedimientos actuales de adquisición de software reposan en el supuesto de que se puede especificar de antemano un sistema satisfactorio” — Fred Brooks

Las tareas de cualquier proyecto de desarrollo software tienen 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).

El concepto de historia de usuario tiene sus raíces en la metodología “eXtreme Programming” o programación extrema (capítulo 8). 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 (capítulo 4).

Las historias de usuario se representan normalmente en posit 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

3. Las historias de usuario

muestra en la Figura 5, aunque también se pueden gestionar por medio de herramientas como las descritas en el anexo de este libro.



Figura 5. Ejemplo de historia de usuario en písit

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

1. Creación de la tarjeta (o písit). Una descripción escrita con un título asociado que sirve como identificación de la funcionalidad. También puede contener la estimación (apartado 5), 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 písit, estas aclaraciones (o confirmaciones) se pueden especificar a la vuelta del mismo, como se muestra en la siguiente figura:

3. Las historias de usuario

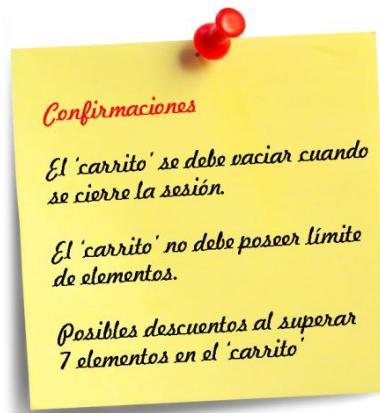


Figura 6. Ejemplo de aclaraciones representadas en un pósit

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.

A parte de por 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.

3. Las historias de usuario



Figura 7. Ejemplo real de uso del pósit con tareas e historias de usuario

3.1 Qué información contiene 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 siguiente figura:

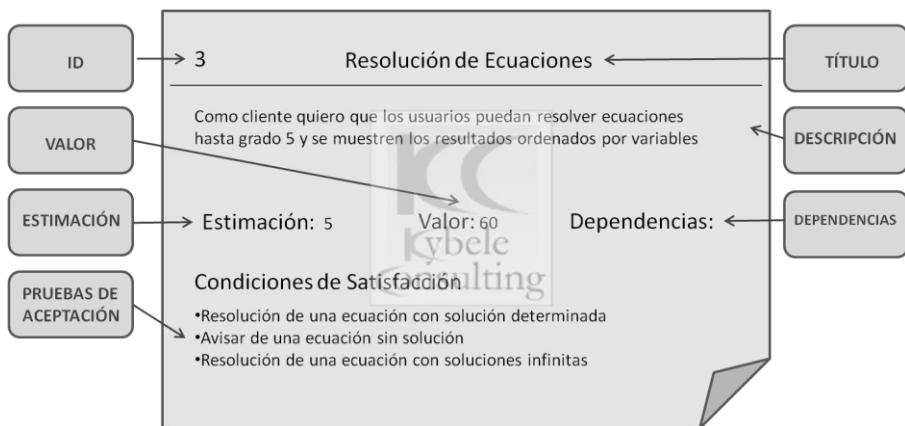


Figura 8. Historia de usuario

3. Las historias de usuario

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

- **ID:** identificador de la historia de usuario.
- **Título:** título descriptivo de la historia de usuario.
- **Descripción:** descripción sintetizada de la historia de usuario. Si bien el estilo puede ser libre, la historia de usuario debe responder a tres preguntas: ¿Quién se beneficia? ¿Qué se quiere? y ¿Cuál es el beneficio? Cohn (M. Cohn, 2004) recomienda seguir el siguiente patrón: **Como [rol del usuario], quiero [objetivo], para poder [beneficio].** Con este patrón se garantiza que la funcionalidad se capture a un alto nivel y que se está describiendo de una manera no demasiado extensa.
- **Estimación:** estimación del tiempo de implementación de la historia de usuario en unidades de desarrollo, conocidas como puntos de historia (estas unidades representarán el tiempo teórico de desarrollo/persona que se estipule al comienzo del proyecto). Este concepto se desarrolla más adelante en el capítulo 5, pág. 57.
- **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. En el apartado 5.1 se describen técnicas para calcular este valor.
- **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 las historias de las que depende.
- **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.

3. Las historias de usuario

Cohn en (M. Cohn, 2004) 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. Al igual puede ocurrir con documentos de especificaciones de seguridad, normativas, etc.

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

3.2 Malas interpretaciones del concepto de historia de usuario

Es bastante común a la hora de comenzar a implantar metodologías ágiles que no quede claro el concepto de historia de usuario; sobre todo por el empeño en intentar asemejar las historias de usuario con requisitos, casos de uso o tareas. En los tres siguientes epígrafes se intenta aclarar qué es y qué no es una historia de usuario.

3.2.1 Una historia de usuario no es una especificación de requisitos

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

3. Las historias de usuario

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 pruebas, normativas, convenciones de codificación a seguir, etc. Este es uno de los principales problemas que suelen encontrarse los equipos de desarrollo a la hora de trabajar con metodologías ágiles. Por un lado se ven seducidos por la idea ágil, pero a la hora de ponerla en práctica encuentran problemas sobre donde ubicar gran parte de la información (especificaciones, necesidades, normativas, no funcionalidades, criterios de interfaz, etc.) que típicamente se encontraban en la especificación de requisitos. Este es el motivo por el que los equipos, al comenzar a trabajar con una metodología ágil, suelen

3. Las historias de usuario

deducir de manera incorrecta que esta información debe ahora ubicarse en las historias de usuario.

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 capítulo 2, pág. 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.

3.2.2 Una historia de usuario no es un caso 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), las historias de usuario están más cerca de la captura de requisitos (aquella fase 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. Andrew Stellman (Stellman, 2009) especifica los siguientes aspectos clave que diferencian una historia de usuario de un caso de uso:

3. Las historias de usuario

- **Las historias de usuario describen necesidades.** Describen necesidades generalmente del día a día de los usuarios de manera breve y mediante lenguaje de negocio, es decir, lenguaje entendible por el usuario y/o cliente.
- **Los casos de uso describen el comportamiento a incluir en el software para cumplir dichas necesidades.** Deben contener los suficientes detalles, expresados de manera clara y no ambigua, para que el desarrollador consiga entender qué es lo que el software debe realizar exactamente.
- **Las historias de usuario son fáciles de entender por parte del cliente o del usuario.** El lenguaje de las historias de usuario es interpretable por el cliente y los usuarios ya que carecen de detalles técnicos más específicos con los que no están familiarizados.
- **Los casos de uso describen de manera completa la interacción entre el software y los usuarios (u otros sistemas).** Cuando se redacta un caso de uso, se diseña la solución funcional para cubrir una necesidad del usuario y/o cliente.

Generalmente, cuando un proyecto comience a seguir una metodología ágil, se deberían olvidar completamente los casos de uso y el equipo debería centrarse en la realización de historias de usuario. Según Cockburn (A. Cockburn, 2008), 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?

3. Las historias de usuario

- 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.

3.2.3 Una historia de usuario no es una tarea

Las historias de usuario también se suelen asemejar con tareas. No obstante, una historia de usuario es una descripción breve y de alto nivel de una funcionalidad o necesidad que debe tener el software final; mientras que una tarea es una descripción sencilla de una pequeña parte del trabajo necesario para completar una historia de usuario. Generalmente en los proyectos ágiles, el desarrollo de una historia de usuario suele dar lugar a varias tareas.

Sin embargo, la actividad de dividir estas historias de usuario en tareas no es fácil. De hecho, en una gran parte de los proyectos a los que nos hemos enfrentado, parece que el tema de descomponer las historias de usuario en tareas no queda del todo claro.

3. Las historias de usuario

Por ello, a continuación se muestran algunas buenas prácticas descritas en (Kuphal, 2011) que se pueden tener en cuenta a la hora de descomponer las historias en tareas.

1. Intentar que el tamaño de las tareas sea de entre medio día hasta un máximo de 3 o 4 días de trabajo de un sólo miembro del equipo. Tareas más pequeñas suelen conllevar grandes pérdidas de tiempo a la hora de planificarlas. Por otro lado, las tareas de más de 3 o 4 días de trabajo, se pueden dividir en otras tareas, si es posible, con el fin de no ocupar demasiado tiempo y ser completadas de manera eficiente.
2. Crear tareas que una vez completadas generen un producto entregable. Por ejemplo, en vez de tareas como “construir la interfaz de usuario”, “construir la lógica de negocio” o “construir la capa de persistencia”, una división que se recomendaría para este tipo de funcionalidad sería “implementar el módulo de inserción de un nuevo usuario”, “implementar el módulo de actualización de un usuario”, o “implementar el módulo de eliminación de un usuario”. De esta manera, cuando se complete una tarea se podrá generar un producto entregable. A su vez estas tareas no dependen de la finalización de otras para ser probadas
3. No dedicar excesivo tiempo a estudiar todos los detalles de cada tarea. Suele ocurrir que una vez definida la tarea, el siguiente paso sea su estimación, y que para ello el equipo quiera definir y conocer todos los detalles posibles de la misma. Aunque, efectivamente, este análisis ayuda a realizar estimaciones más precisas, puede demorar en gran medida el proceso de división de las historias de usuario en tareas.
4. En el conjunto de tareas, deben estar incluidas tareas de pruebas y su posible automatización.

3. Las historias de usuario

Una vez descrito el conjunto de buenas prácticas y para completar este tema, Hiren Doshi en (Doshi, 2010) muestra un listado de ejemplos de tareas en las que se puede dividir una historia de usuario y que mostramos a continuación:

- **Diseño de la historia de usuario:** especialmente para generar discusión sobre cómo se va a implementar la historia de usuario.
- **Implementación de una historia:** esta tarea puede incluir la definición de las interfaces y los métodos necesarios para cubrir la necesidad expresada en la historia. Pueden existir múltiples tareas de este tipo.
- **Pruebas unitarias asociadas a la historia (por ejemplo TDD⁸):** esta tarea debiera ser obligatoria para cada historia.
- **Pruebas de aceptación asociadas a la historia:** tarea definida para desarrollar las pruebas de aceptación automáticas que facilitarán la validación de la historia por parte del cliente y/o usuario.
- **Requisitos no funcionales:** para cada historia, se deben tener en cuenta requisitos de seguridad, rendimiento, escalabilidad, etc. Estos atributos de calidad pueden dar lugar a nuevas tareas.
- **Revisiones de código:** el código siempre debe haber sido revisado, convirtiéndose la revisión en otra tarea.
- **Refactorización del código:** esta tarea se debe incluir para cada historia con el fin de evitar una complejidad excesiva del código.
- **Emular interfaces:** en el caso de que se demore en exceso la implementación de una interfaz, éstas se deben emular con el fin de comenzar a trabajar.
- **Pruebas exploratorias:** pruebas ad-hoc para una historia.

⁸ TDD (Test-Driven Development), es una técnica que promueve implementar las pruebas antes, o a la vez, que se programa.

3. Las historias de usuario

- **Corrección de errores:** dependiendo de la historia, se debe reservar tiempo para la resolución de errores o incidencias.
- **Verificación de errores:** dependiendo de la historia, se debe reservar tiempo para verificar la corrección de los errores.
- **Demo:** demostración interna al equipo de la historia. Esta pequeña demostración se suele realizar una vez finalizada la historia en la reunión diaria.
- **Actualizar la wiki o el repositorio de documentos:** con el diseño y los resultados de la historia.
- **Documentación de usuario final:** la generación de los manuales de usuario, instalación, etc. también debe estar ubicada en alguna tarea.

Hay que tener en cuenta que el anterior es solamente un listado de ejemplo, y que dependiendo de la historia de usuario habrá elementos de esta lista que puedan aplicar o no. A su vez, en los proyectos ágiles pueden surgir otros tipos de tareas que no se encuadren en esta lista y que sean igualmente válidos.

El control de la relación entre las historias de usuario y sus tareas se realiza frecuentemente con hojas de cálculo aunque también se podría utilizar cualquier otra herramienta de gestión de tareas. A continuación, se simula una hoja de cálculo en donde se muestra esta relación entre historias y tareas:

Historias de usuario	Tareas
	Diseño del módulo del carrito.
Como cliente, quiero poder añadir elementos a un carrito para poder revisarlos antes de la compra.	Implementación del modulo del carrito.
	Pruebas de aceptación del módulo del carrito.
...	...

Tabla 1. Relación entre historias de usuario y tareas con hoja de cálculo

3. Las historias de usuario

3.3 Historias técnicas: otro tipo de historia de usuario

Dentro de las historias de usuario nos podemos encontrar con un tipo particular, las denominadas “historias técnicas”. Las historias técnicas reúnen las tareas que hay que hacer, pero que carecen de interés para el cliente: la actualización de una base de datos, la refactorización de un mal diseño, o automatizar pruebas. Estas serían mejoras internas acordadas en las reuniones de retrospectiva (ver apartado 4.4, pág. 52).

En los proyectos ágiles, en el caso de utilizar las historias técnicas, es conveniente diferenciarlas de manera adecuada de las historias de usuario. De esta manera se podrá analizar la cantidad de tiempo utilizado en una iteración para aspectos directamente relacionados con la entrega de valor al cliente y las tareas técnicas.

3.4 La calidad de una historia de usuario

Para asegurar la calidad de una historia de usuario, Bill Wake describió en 2003 un método llamado **INVEST** (Wake, 2003). El método se usa para comprobar la calidad de una historia de usuario revisando que cumpla las características descritas en la Tabla 2.

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.
---	---

3. Las historias de usuario

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 mas 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”). 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 2. Descripción del INVEST

3. Las historias de usuario

3.5 Las dependencias entre historias de usuario

En el apartado 3.4 se han descrito las características de calidad que se recomienda que tengan las historias de usuario, una de ellas es que sean independientes. La independencia entre historias es fundamental para facilitar actividades como priorizar, añadir o eliminar historias de una entrega o de la planificación de una iteración. Las historias de usuario deberían ser atómicas, de tal manera que se pudieran comenzar y terminar sin tener en cuenta al resto de historias.

Sin embargo, definir todas las historias de usuario de manera que sean independientes unas de otras no es una tarea fácil. Por ejemplo, puede darse el caso que aparezcan dos historias de usuario: “abrir sesión” y “cerrar sesión”, en donde para poder cerrar la sesión del usuario debe haber sido abierta previamente. En este caso, se podría solucionar uniendo las dos historias de usuario en una: “Acceso del usuario al sistema”. Por tanto, combinar historias de usuario es una forma de evitar que existan dependencias entre ellas, pero esto no es siempre posible.

Otra recomendación para evitar dependencias es incluir en cada historia de usuario un “corte vertical” de la aplicación, es decir, que se abarque la interfaz de usuario, el dominio de negocio y el acceso a los datos necesarios para cubrir la funcionalidad que describe (al igual que se ha descrito como buena práctica en la descomposición de historias en tareas del apartado 3.2.3). En caso contrario, si se dividen las historias de usuario por capas (el desarrollo de la interfaz, dominio y acceso a datos se realiza por separado), se obtendrán con total seguridad dependencias entre ellas.

Importante: aunque lo deseable es conseguir que todas las historias de usuario que se definen en un proyecto sean independientes, no es una actividad fácil. De hecho, la experiencia muestra que es prácticamente inevitable que acaben existiendo dependencias.



4. Scrum

“Aunque la implementación parcial de Scrum es posible, el resultado no es Scrum” -
K. Schwaber y J. Sutherland

Scrum (K. Schwaber, 2004) es una metodología ágil que proporciona un marco de trabajo 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.

Cada año (VersionOne Inc., 2011) realiza una encuesta con el objetivo de observar el estado de las prácticas ágiles en la industria del software. En base a sus resultados se obtienen datos muy interesantes, entre otros por la destacada participación en la misma (en la última encuesta, la correspondiente al 2011, recibieron 6042 respuestas). Entre los datos más destacados del informe del año 2011, se observa que más de un 65% de las organizaciones que aplican prácticas ágiles usan Scrum, normalmente acompañado de otras como Kanban (apartado 6.2, pág 77) o eXtreme Programming (capítulo 7, pág. 97).

4. Scrum

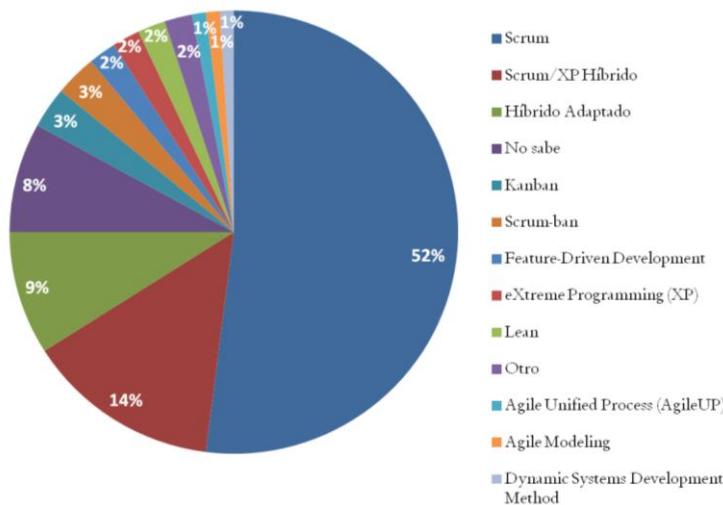


Figura 9. Metodologías Ágiles más utilizadas

El conjunto de buenas prácticas de Scrum se aplica esencialmente a la gestión de proyectos. Por otro lado, es un “framework”, por lo que es necesaria su adaptación en cada organización, o incluso a cada equipo. (J. Sutherland & Schwaber, 2011).

Scrum es una metodología ágil, cuyo objetivo es obtener resultados rápidos, adaptándose a los cambios de las necesidades de los clientes. Las características principales de Scrum pueden resumirse en dos:

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

Scrum se basa en entregas parciales priorizadas por el beneficio que aporta al receptor final del software.

Para ello, como metodología ágil que es, Scrum define un ciclo de vida iterativo e incremental, mejorando la gestión de los riesgos y aumentando la comunicación. Como se indica en la “Scrum Guide” (K. Schwaber & Sutherland, 2010), 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 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. En parte, la transparencia se logra mediante la comunicación de la información. Para obtener esta información es necesario conocer lo que sucede. Una de las características fundamentales de Scrum son las reuniones (ver apartado 4.4, pág. 52). Por ejemplo, en Scrum se realizan reuniones cortas diarias.
- **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. Esto se relaciona directamente con el pilar de inspección. Una buena metodología necesita de un mecanismo de mejora continua, es decir, de control, para adaptarse y mejorar.

A continuación se detallarán los diferentes elementos de Scrum a partir del estudio de su ciclo de vida.

4.1 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.

4. Scrum

Cada equipo Scrum tiene tres roles(K. Schwaber & Sutherland, 2010):

- **Scrum Master:** es el responsable de asegurar que el equipo Scrum siga las prácticas de Scrum. Sus principales funciones son:
 - Ayudar a que el equipo y la organización adopten Scrum.
 - Liderar el equipo Scrum, buscando la mejora en la productividad y calidad de los entregables.
 - Ayudar a la autogestión del equipo.
 - Gestionar e intentar resolver los impedimentos con los que el equipo se encuentra para cumplir con las tareas del proyecto.
- **Propietario del Producto (Product Owner):** es la persona responsable de gestionar las necesidades que serán satisfechas por el proyecto y asegurar el valor del trabajo que el equipo lleva a cabo. Su aportación al equipo se basa en:
 - Recolectar las necesidades o historias de usuario.
 - Gestionar y ordenar las necesidades (representadas por las historias de usuario, descritas en el capítulo 3).
 - 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:
 - **Autogestionado:** el mismo equipo supervisa su trabajo. En Scrum se potenciarán las reuniones del equipo (ver apartado 4.4), aumentando la comunicación. No existe el rol clásico de jefe de proyecto. El Scrum Master tiene otras responsabilidades vistos en el apartado anterior.

- **Multifuncional:** no existen 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 han conseguido realizar proyectos Scrum con equipos distribuidos gracias a herramientas de trabajo colaborativo (Hossain et al., 2009).
- **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 decrece 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.

A estos tres roles se les considera "totalmente comprometidos", tanto con el proyecto como con el proceso de desarrollo Scrum seguido por la organización. Por ello en Scrum reciben el nombre de roles "cerdo".

Existen también otros implicados que no forman parte directamente del proceso de desarrollo pero que deben ser tenidos en cuenta como por ejemplo, los expertos del negocio, "stakeholders" o los usuarios, y que en Scrum reciben el nombre de roles "gallina".

4. Scrum

CURIOSIDADES

Los roles “cerdo” y “gallina” en Scrum

Se utilizan los roles “cerdo” y “gallina” en Scrum para diferenciar el grado de implicación. Esto proviene del siguiente chiste:

“Un cerdo y una gallina daban un paseo junto a la carretera. En un momento dado, la gallina dice al cerdo: este es un buen sitio para abrir un restaurante. Y el cerdo le dice: “buena idea ¿Cómo podría llamarse? La gallina le responde: “huevos con jamón”. El cerdo le replica: “no creo que vaya a asociarme contigo, yo estaría comprometido pero tú sólo estarías involucrada.”

4.2 El Product Backlog

La pila de producto o **Product Backlog** (a partir de ahora 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 (aunque como se vio en el apartado 3.2.1, pág. 30 no son exactamente requisitos)**. 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 pre establecido en Scrum para ordenar las historias de usuario, el más aceptado es partir del valor que aporta al negocio la implementación de la historia de usuario. El responsable de ordenar el Product Backlog es el Product Owner, aunque también puede ser ayudado o recibir asesoramiento de otros roles como, por ejemplo, el Scrum Master y el equipo de desarrollo.

Finalmente, el descubrimiento y la descripción de los elementos del Product Backlog es un proceso continuo. Las necesidades ya no están congeladas desde el principio, sino que se descubren y se detallan a lo largo de todo el proyecto. Este descubrimiento no se limita a las etapas iniciales de desarrollo, sino que abarca todo el proyecto en Scrum.

Tal y como se describe en (Pichler, 2010) 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.
- 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.

4. Scrum

- 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.

Trabajar con un Product Backlog no significa que el equipo Scrum no puede crear otros productos de trabajo útiles, incluyendo un resumen de los distintos usuarios, diagramas para ilustrar las reglas de negocio o los prototipos de la interfaz de usuario. Estos documentos no sustituyen al Product Backlog, sino que explican su contenido.

4.3 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.

En un reciente estudio realizado a más de 6.000 empresas se han obtenido las duraciones promedio de las iteraciones: el 27% realizaba iteraciones de entre 8 y 14 días, el 17% entre 15 y 21 días, el 9% entre 22 y 30 días y el 6% entre 1 y 7 días.

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

Aún 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 10.

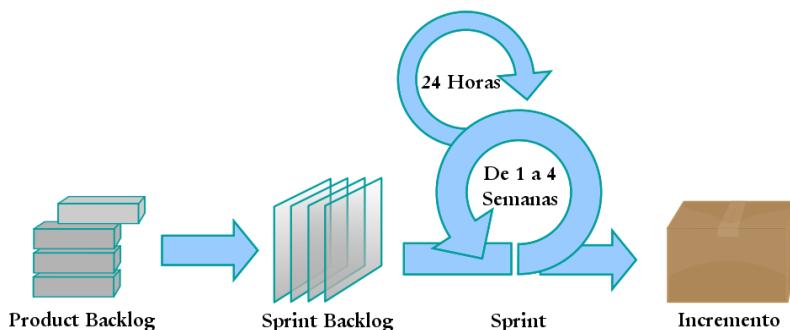


Figura 10. Ejemplo de proceso Scrum

4. Scrum

Importante: Aunque todos los Sprints dan como resultado un incremento del producto software, no todos implican un paso a producción. Es responsabilidad del Product Owner y los clientes decidir el momento en el que los incrementos son puestos en producción.

Una posibilidad para realizar 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 11, cada historia de usuario del Product Backlog es dividida en tareas.



Figura 11. 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 el apartado 5.2.2.1, pág. 66).

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.

Muchas veces se piensa que cada Sprint es un mini ciclo de vida en cascada. Esto no es del todo así, en realidad el Sprint mantiene una serie de características que lo diferencian de un ciclo de vida en cascada. Las tres principales son:

- El diseño es continuo, así como la integración durante el desarrollo o las pruebas: estas actividades se realizan de manera continua a lo largo del Sprint. En un ciclo de vida en cascada se realizan de manera secuencial, como si se tratase de compartimentos estancos. En la parte superior de la gráfica se observan las tareas secuenciales para un ciclo de vida en cascada (obtención de requisitos, análisis, diseño, etc.). Para el caso de un Sprint las actividades no respetan ninguna secuencia.
- El equipo de trabajo: las personas que integran un equipo de desarrollo realizarán diferentes tareas (estimación, diseño, desarrollo, pruebas, etc.). No existen equipos especializados, como, por ejemplo, en el ciclo de vida en cascada. Es decir, para una tarea de análisis en el ciclo de vi-

4. Scrum

da en cascada existe la figura del analista; lo mismo sucede con el diseño, el desarrollo o las pruebas. Durante un Sprint, en cambio, las tareas de análisis, diseño, desarrollo o pruebas puede realizarlas cualquier integrante del equipo.

4.4 Las reuniones

Las reuniones son un pilar importante dentro de Scrum. Se realizan a lo largo de todo el Sprint como muestra la Figura 12, representadas en los cuadros con color gris. 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 Product Backlog 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 encontrados.
- **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.

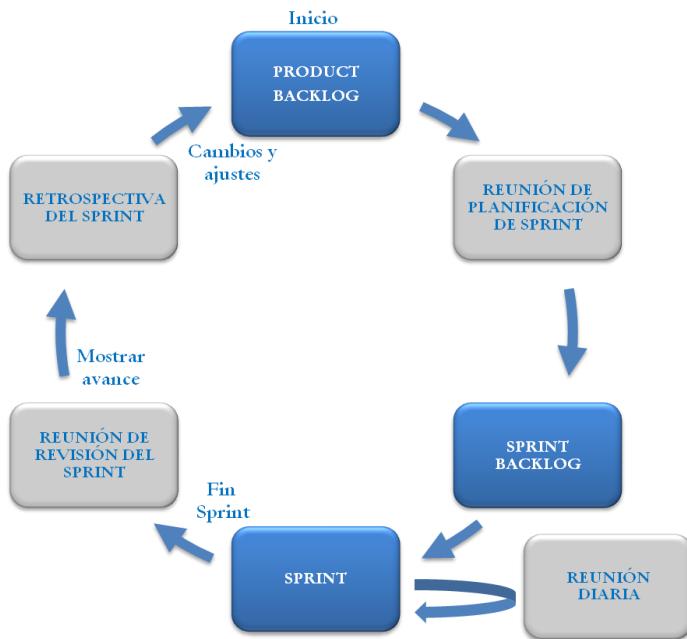


Figura 12. Reuniones de Scrum

4.5 Medir el progreso del proyecto

En el capítulo 7 hablaremos más detenidamente sobre las prácticas ágiles para realizar mediciones en proyectos de desarrollo software. Aquí hablaremos sobre uno de los gráficos más importantes con los que **se mide el progreso real**

4. Scrum

del proyecto de desarrollo software con Scrum. 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).

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 13, 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.

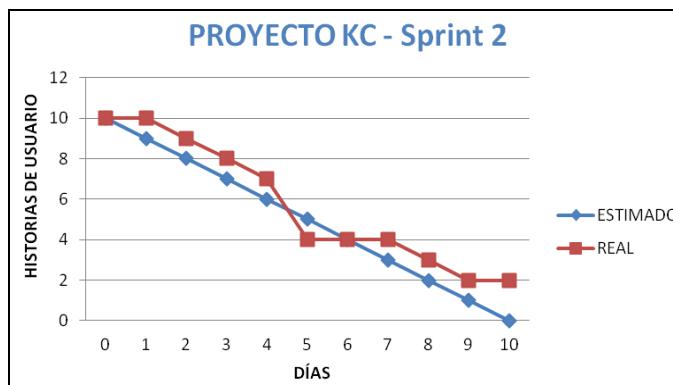


Figura 13. Gráfico BurnDown

4.6 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.

A continuación se van a describir las principales ventajas:

- **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**

4. Scrum

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

5

5. Planificación y estimación ágil

“Predecir es muy difícil, y sobre todo el futuro” - Niels Bohr

Dentro de la planificación de proyectos, el papel que juegan la estimación y la priorización de las actividades a desarrollar es crucial. El primer paso en la planificación (también en el mundo ágil) es la creación de la lista de funcionalidades a desarrollar (en el caso de Scrum, Product Backlog), es decir, la definición de las necesidades a realizar en el proyecto. Como se ha explicado en el capítulo 3, en proyectos ágiles estas necesidades se pueden dividir en objetivos expresados como historias de usuario (user stories) (M. Cohn, 2004), cada una aportando valor al negocio de manera incremental e individual.

A la hora de priorizar las historias de usuario se deben tener en cuenta principalmente dos de sus posibles campos:

- **Valor:** valor (puede no ser numérico) que aporta la historia de usuario al cliente y/o usuario. El objetivo del equipo es maximizar el valor y la satisfacción percibida por el cliente y/o usuario en cada iteración.
- **Estimación:** estimación del coste de implementación en unidades de desarrollo (estas unidades representarán el tiempo teórico de desarrollo/persona que se estipule al comienzo del proyecto).

5. Planificación y estimación ágil

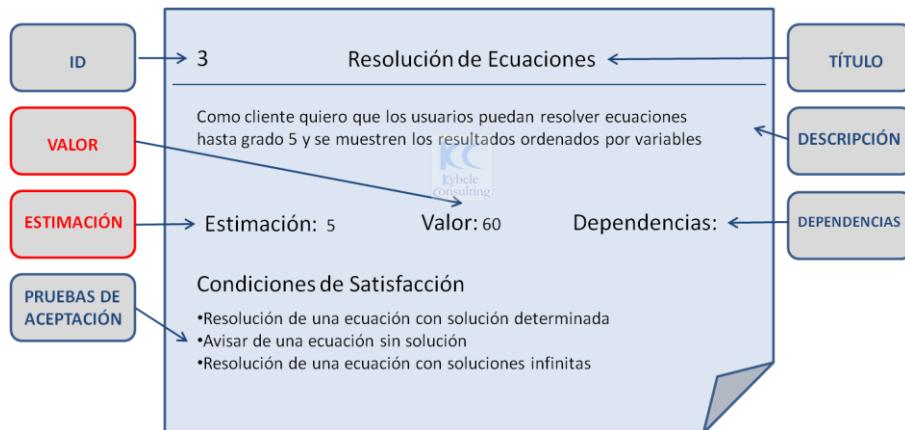


Figura 14. Valor y estimación en la historia de usuario

5.1 Asignar valor a las historias de usuario

Como se ha comentado con anterioridad, los ítems de la lista de necesidades deben tener un valor asignado. Dicho valor es asignado por el representante del cliente/usuario en el proyecto teniendo en cuenta las siguientes variables:

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

5. Planificación y estimación ágil

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 (Figura 15). Dentro de cada grupo resultará más fácil realizar una ordenación relativa por valor numérico. Todo ello servirá para que en cada iteración se entregue el producto al cliente maximizando su valor, como se explicará en el apartado 5.3.



Figura 15. Ejemplo de escala cualitativa

Existen otro tipo de ponderaciones bastante divulgadas, como por ejemplo la técnica MoSCoW (Clegg & Barker, 1994). Su fin es obtener el entendimiento común entre cliente/usuario y el equipo del proyecto sobre la importancia de cada historia de usuario. La clasificación que proporciona esta técnica es la siguiente:

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

5. Planificación y estimación ágil

- **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.

Tras esto cada historia de usuario se pondera individualmente. Dicha ponderación puede ser llevada a cabo de varias formas:

- Estimando el retorno de inversión (ROI⁹), en unidades monetarias.
- A través de los riesgos. A más riesgos, menor valor.
- De forma arbitraria poniendo un mínimo (normalmente 0) y un máximo, y ponderando a partir de dichas cifras.

Importante: cómo a lo largo de un proyecto las personas pueden variar, aunque se mantengan sus roles (por ejemplo, el representante del cliente o usuario en el equipo pueden desempeñarlo diferentes personas), es muy importante institucionalizar las palabras y sus significados: ¿Cuándo una historia de usuario es imperativa? ¿Cuándo es importante? De esta manera se consigue homogeneizar la estimación a lo largo del proyecto, y, por lo tanto, aprender de las mismas.

⁹ ROI, por sus siglas en inglés que significan Return On Investment, es un ratio que compara el beneficio o la utilidad obtenida en relación a la inversión realizada.

5.2 Estimación de las historias de usuario

Antes de explicar las técnicas usadas a la hora de estimar una historia de usuario, se considera importante realizar una introducción acerca de la estimación en proyectos software así como la explicación de los conceptos más importantes que se deben conocer en los procesos de estimación de proyectos basados en metodologías ágiles.

5.2.1 Algunos conceptos sobre la estimación software

En el año 2006 Steve McConnell realiza uno de los estudios más reconocidos en el ámbito de la estimación software (este estudio puede consultarse en (S. McConnell, 2006), concluyendo que la mayoría de los errores que se producen más frecuentemente en los proyectos software están relacionados con aspectos de estimación. Entre estos errores destacan los siguientes:

- **Calendario optimista:** la tendencia al estimar es hacerlo de manera optimista. Esta tendencia es general y disminuye con la experiencia. Si se hiciese un histórico de estimaciones, se observaría que normalmente estimar por encima (de manera pesimista) es mucho menos frecuente que estimar por debajo (de manera optimista).
- **Expectativas no realistas:** muchas veces las tareas sobre las que se estima son ambiguas. De esta manera aunque el equipo seguramente llegue a un consenso común, el cliente puede mantener otra expectativa presionando para obtener el resultado que él desea.
- **Confundir estimaciones con objetivos:** al estimar tenemos que tener en cuenta nuestra capacidad actual real y el histórico de productividad. Por ejemplo, se quiere terminar un proyecto en 3 meses (objetivo) suponiendo que el equipo lo forman 2 programadores y que el 20% del tiempo están respondiendo incidencias de otros proyectos. Con estos supuestos, la estimación se tendrá que hacer de acuerdo a los recursos disponibles y a los requisitos del proyecto y puede ser factible o no.

5. Planificación y estimación ágil

- **Omisión de estimar tareas necesarias:** las pruebas, la documentación, las reuniones, las tareas relacionadas con la gestión de configuración o la calidad, puede que no se planifiquen. Siempre se debe tener en cuenta que estas tareas también consumen tiempo.

Otro aspecto a resaltar es que generalmente el nivel de precisión de la estimación es distinto a medida que avanza el proyecto. Al inicio, cuando sólo se tienen requisitos, el error con el que se trabaja es mayor que cuando se estima en la fase de diseño. Una figura muy útil para entender este tema es el “cono de incertidumbre” (S. McConnell, 2006), que muestra cómo las estimaciones son más precisas según progresá el proyecto. Al inicio del proyecto, como se muestra en la Figura 16, las estimaciones que se realicen podrán ser hasta 4 veces por encima (es decir, se estima más tiempo del necesario) o por debajo (menos tiempo del necesario) de los valores reales (representados en la figura con el valor 1x del eje vertical). Por otro lado, conforme avanza el proyecto estas estimaciones se irán acercando progresivamente a esos valores reales (1x o una precisión total).

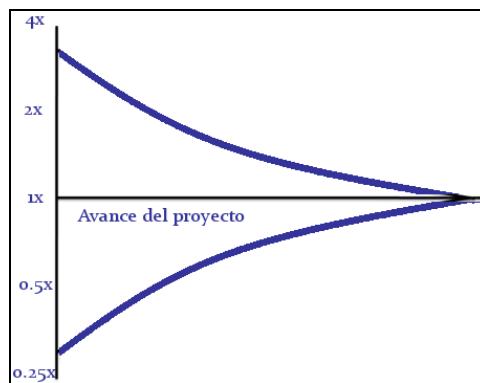


Figura 16. El cono de incertidumbre en la estimación software (McConnell, 2009)

5. Planificación y estimación ágil

La frecuencia con que se producen los errores comentados anteriormente y la falta de precisión de las primeras estimaciones realizadas, han provocado que las prácticas de estimación no se hayan divulgado demasiado dentro de la gestión de proyectos. De hecho, la estimación software es uno de los campos en la ingeniería del software más teorizados y menos puestos en práctica lo que ha provocado que se hayan generado una serie de mitos que se discuten a continuación:

MITO ✗	RESPUESTA ✓
✗ Las técnicas son complicadas.	✓ No es necesario conocer profundamente los mecanismos internos de la técnica de estimación para aplicarla.
✗ Las técnicas reemplazan el juicio experto.	✓ Realmente lo complementan. Las decisiones las siguen tomando las personas. En caso de seguir una técnica de estimación, lo harán con más información.
✗ Son actividades “tradicionales”, “antiguas”.	✓ Existen técnicas de estimación ágiles y rápidas como el Planning Poker (lo veremos a continuación, en el punto 5.2.2.1).
✗ Los resultados no tienen nada que ver con la realidad.	✓ Las técnicas son ajustables, y dependen mucho de las entradas.
✗ Sólo se estima al inicio del proyecto.	✓ Una de las fortalezas de una buena estimación es que se realice periódicamente, como apoyo al seguimiento. Para ello se necesita una técnica de estimación automatizada.

Tabla 3. Mitos acerca de la estimación

5. Planificación y estimación ágil

Lo que sí debe quedar claro es que la estimación no intenta adivinar el tamaño del producto software o la cantidad de jornadas que dura el proyecto.

Aunque como se ha comentado anteriormente, las prácticas de estimación no se han divulgado demasiado en los proyectos de desarrollo de software, la aparición de técnicas ágiles de estimación ha conseguido que cada vez se consideren más imprescindibles como ayuda a la planificación de proyectos ágiles. Y como la gestión de proyectos ágiles es una actividad adaptativa, no es extraño que las técnicas de estimación en un proyecto ágil tengan también características ágiles.

Para poder entender la estimación ágil deben quedar claros los siguientes 3 conceptos (M. Cohn, 2005):

- Estimación del tamaño: técnica que proporciona una estimación de alto nivel para los elementos de trabajo. Se suele usar una medida neutral como por ejemplo los puntos de historia (story points). Se explica con más detenimiento en el apartado 5.2.2.
- Velocidad: indicador que describe el número de puntos de historia que el equipo de trabajo puede desarrollar en una iteración.
- Estimación del esfuerzo: técnica que proporciona una traducción del tamaño (medido por ejemplo en puntos de historia) en el esfuerzo requerido por el equipo para completar el trabajo (expresado normalmente en días u horas).

5.2.2 La estimación de una historia de usuario

A hora de asignar una estimación a cada historia de usuario se suelen utilizar unidades de medida que reflejan el esfuerzo necesario del equipo para desarrollar la historia de usuario. Las unidades de medida más utilizadas son las siguientes:

5. Planificación y estimación ágil

- Puntos de historia (story points): es una unidad que describe cuánto esfuerzo requiere desarrollar una historia de usuario. Es una medida que se adapta al equipo, y suele ser específica del mismo, es decir, que diferentes equipos pueden dar significados diferentes a este indicador.

Una vez que el equipo define sus propios puntos de historia, esa medida se mantiene consistente para ese mismo equipo cada vez que se estimen historias de usuario. Esta consistencia de los puntos de historia es fundamental para crear predictibilidad, propósito principal de las estimaciones.

Por ejemplo, se puede decidir estimar todas las historias en una escala de 1 a 10 puntos de historia. Una posibilidad es fijar una historia como referente (por ejemplo en 5 puntos de historias) y a partir de esa, estimar las demás. Cuando se conozca el tiempo que el equipo tarda en desarrollar un punto de historia, estos valores se podrán traducir en horas/persona o días/persona. De esta manera, el número de puntos de historia desarrollados en una iteración se conoce como la velocidad del equipo y se debe medir con el objetivo de afinar las estimaciones a lo largo de las diferentes iteraciones (la medición en proyectos ágiles se trata en el capítulo 7 de este libro).

Otras unidades utilizadas para asignar una estimación a las historias de usuario pueden ser en función del tiempo que se tarda en desarrollarlas:

- Tiempo ideal: tiempo que se dedicará para desarrollar una historia de usuario sin tener en cuenta posibles interrupciones en el trabajo.
- Tiempo real: tiempo que se dedicará en desarrollar una historia de usuario teniendo en cuenta todas las posibles interrupciones, los trabajos del equipo dedicados a otros proyectos, etc.

El tema de cómo afectan las interrupciones y otros aspectos en las estimaciones se tratan.

5. Planificación y estimación ágil

5.2.2.1 Técnica de estimación: Planning Poker

El Planning Poker (Haugen, 2006) (M. Cohn, 2005) es una técnica cuyo nombre ha sido acuñado por James Grenning y popularizado por Mike Cohn. Se la utiliza como una técnica para calcular una estimación del esfuerzo o del tamaño relativo de las tareas en un desarrollo software. El Planning Poker está basado en el consenso y 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.

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 siguiendo la secuencia de Fibonacci o alguna otra secuencia similar. El objetivo de esta numeración es reflejar la incertidumbre inherente en la estimación con un conjunto lo suficientemente amplio de números. Esto hace que no necesitemos un gran número de cartas. De esta manera, en caso de dudas estaremos obligados a elegir el número que se acerca más (por ejemplo, si estimamos que una historia pueden ser 7 puntos, pero en las cartas no se dispone del número 7, elegiremos la carta del número 8). A continuación se muestra un mazo de cartas numerado con una secuencia similar a la de Fibonacci bastante usada en los proyectos ágiles:

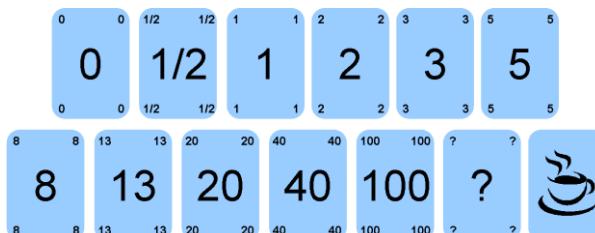


Figura 17. Ejemplo de mazo de cartas de Planning Poker

Como se puede observar en la Figura 17 en cada mazo suelen existir 2 cartas especiales: la carta de la interrogación que indica que no se conoce toda la

5. Planificación y estimación ágil

información necesaria o que no se tiene la suficiente experiencia para dar una estimación de la historia de usuario; y la carta de la taza de café, que puede ser descubierta por algún miembro del equipo para solicitar un pequeño descanso en el proceso de estimación.

En la Figura 18, se puede ver el proceso de estimación a través de la técnica de Planning Poker:

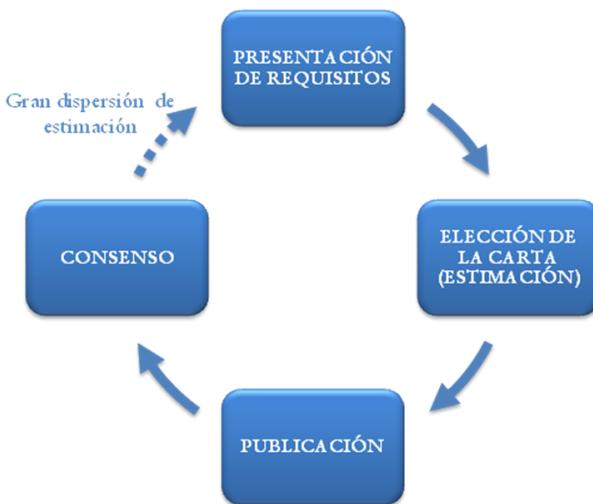


Figura 18. Etapas del Planning Poker

1. Se presentan las historias de usuario (o sus tareas) a estimar una por una, haciendo una descripción de las mismas 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 su mazo de cartas y escoge la carta (o la suma de un conjunto de cartas) que representa su estimación del trabajo para implementar la necesidad en cuestión. Como se ha comentado,

5. Planificación y estimación ágil

las unidades de estimación utilizadas pueden variar y deben estar definidas previamente.

3. 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 20) se vuelve al discutir la historia de usuario y se vuelve a realizar el proceso de estimación.
4. Por último, si no existe una gran dispersión, se llega por consenso a un acuerdo en la estimación de la historia de usuario.

Generalmente la dispersión en las estimaciones es síntoma de que la información que maneja 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 la historia para que en la siguiente ronda de estimación, la dispersión de las estimaciones sea mucho menor y se pueda llegar fácilmente a un consenso.

Importante: Usando la técnica 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. Esta técnica lleva a estimaciones respaldadas por todos los involucrados y basadas en consensos.

5.2.2.2 Ejemplo de cómo convergen las estimaciones iniciales en Planning Poker

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

5. Planificación y estimación ágil

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			

Tabla 4. Estimaciones Planning Poker

En la tabla anterior se observa que la estimación final de la iteración es de 22 puntos de historia. A continuación se analizan cada uno de los casos:

- 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, el resto converge a su estimación.

5. Planificación y estimación ágil

- 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: Planning Poker es una técnica dinámica de estimación que favorece la participación de todos los asistentes, reduce el tiempo de estimación de una funcionalidad, y lo más importante, consigue alcanzar una convergencia en la estimación de las historias de usuario.

5.2.3 Peligros al estimar

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

Durante el desarrollo, entre un 25% y 35% del tiempo se invertirán en otras actividades (imprevistos, llamadas, correo, etc.) como muestra la Figura 19. Asimismo es bueno recordar que el trabajo se expande hasta ocupar todo el tiempo (Parkinson, 1957) del que se disponía para su realización (esto es conocido como la ley de Parkinson (Parkinson, 1957)). Estos dos aspectos significan que si se realiza una estimación por encima (es decir se planifican 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. En cambio, si se estima por debajo (se planifican 3 días reales para una tarea de 3 días ideales) se producirán retrasos.

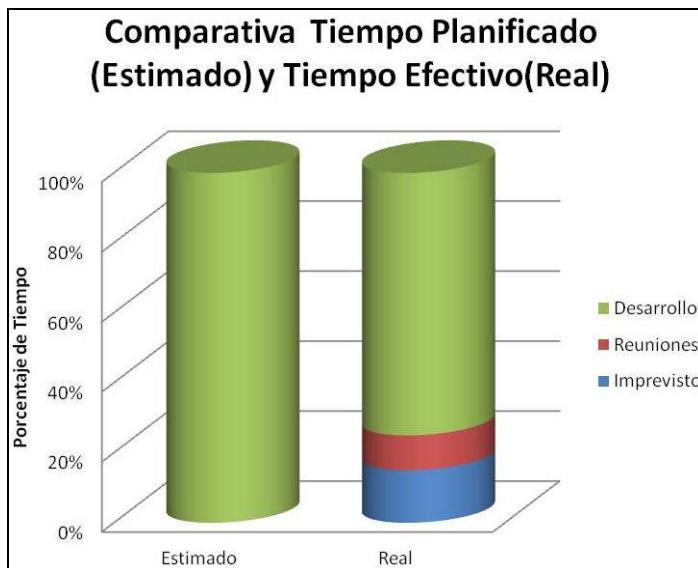


Figura 19. Comparativa tiempo planificado y tiempo efectivo

5.3 Priorización de las historias de usuario

Cuando ya se dispone de la lista completa de historias de usuario estimadas por el equipo de trabajo y con un valor dado por el cliente y/o usuario, llega el momento de priorizar aquellas historias de usuario que se van a desarrollar en cada iteración.

Estas historias de usuario a desarrollar se seleccionan en la reunión de planificación de la iteración en la que deben estar presentes el representante del cliente o usuario así como el equipo de trabajo al completo. De esta manera el representante del cliente o usuario propone una serie de historias a desarrollar en la iteración y especifica todos los detalles aclarando las dudas que puedan surgir al equipo de trabajo.

5. Planificación y estimación ágil

La selección de las historias de usuario se realiza por consenso teniendo en cuenta los campos (descritos al inicio de este capítulo) “valor”, para asegurar que se le da el máximo valor de negocio al usuario al final de la iteración; y “estimación”, que permite planificar las historias de usuario que por tiempo se pueden encuadran en la iteración.

Importante: El objetivo principal es que al final de cada iteración se entregue el máximo valor posible al cliente y al usuario final.

Sin embargo hay que tener en cuenta que concentrar el máximo valor entregado en las primeras iteraciones del proyecto, puede provocar en el cliente/usuario un pensamiento de que el equipo reduce su rendimiento y/o productividad en el resto de iteraciones (ya que observa menos avances al final de las iteraciones conforme el proyecto va avanzando). Idealmente se deben planificar las historias de usuario para que una iteración aporte el máximo valor de negocio al cliente teniendo en cuenta que todas las iteraciones sean lo más homogéneas posible.



6. Lean Software Development y Kanban

“Las reglas de Kanban son simples.
Pero, como en el ajedrez, sólo porque las reglas sean simples no significa que el juego sea simple” -
Henrik Kniberg

“Reduce todo hasta su expresión más sencilla, pero no más” - Albert Einstein

Anque como se comentó en el capítulo 1, comparar la ingeniería del software con la fabricación de productos físicos puede no ser siempre lo más acertado, existen determinadas técnicas o estrategias de la fabricación tradicional adaptables y que aportan beneficios a la ingeniería del software.

En los años 50, la industria japonesa estaba recuperándose de la segunda guerra mundial y logró crecer con gran éxito al aplicar a sus fábricas de automóviles los conceptos de calidad en la producción creados por los principales gurús estadounidenses (Harvey, 2004), de entre los que destaca Deming. La paradoja es que siendo métodos idealmente originados por estadounidenses, fueron aplicados por los japoneses, convirtiendo a Japón en el líder de la industria automovilística por encima de los EEUU. Es en esta época cuando surge el término “Lean” o “Lean Manufacturing” (cuya traducción es fabricación esbelta) que tiene su origen en Toyota. De hecho, “Lean” es sinónimo de

6. Lean Software Development y Kanban

“Toyota Production System”, una estrategia de fabricación aplicada con mucho éxito en Japón y ahora muy famosa en el mundo del software. El artífice del Lean fue Taiichi Ohno (1912 – 1990) y su estrategia se basó en los siguientes principios:

- **Calidad perfecta a la primera:** búsqueda de cero defectos, detección y solución de los problemas en épocas tempranas del proceso.
- **Minimización del despilfarro:** eliminación de todo aquello de lo que se puede prescindir: sobreproducción, tiempos de espera, inventarios y transportes innecesarios, defectos y/o procesos inadecuados.
- **Mejora continua:** este principio es conocido como Kaizen, se basa en que “todo puede mejorar”, tanto cada uno de los pasos del proceso como la producción en sí. Este principio representa un avance consistente y gradual en dónde se dinamizan los esfuerzos del equipo para mejorar con un mínimo coste y conservando el margen de utilidad y el precio competitivo. Todo esto cumpliendo con las especificaciones de entregar en tiempo y lugar exacto, así como entregar cantidad y calidad sin excederse.
- **Procesos "pull":** producir sólo lo necesario en base a los productos que son solicitados por el cliente final, evitando ocupar máquinas, equipos y personas en producciones cuya demanda no es inmediata.
- **Flexibilidad:** producir rápidamente diferentes mezclas de gran variedad de productos, sin sacrificar la eficiencia debido a volúmenes menores de producción
- **Desarrollo de una relación a largo plazo con los proveedores,** a partir de los acuerdos a los que se llegue para compartir información y compartir el riesgo de los costes.

6. Lean Software Development y Kanban

6.1 El Lean y los métodos ágiles: Lean Software Development

El desarrollo ágil es un paraguas que incluye varias metodologías (Scrum, XP, etc.). Todas ellas tienen en común que siguen en mayor o menor medida los valores y principios del manifiesto ágil presentado en el capítulo 2 de este libro. Por otro lado, hay incluso quien afirma (M. Fowler, 2008) que las ideas de Lean y las ideas ágiles 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 (M. Fowler, 2008), como por ejemplo Mary y Tom Poppendieck. Mary, esposa de Tom, trabajó en una fábrica que usaba el método Lean, y Tom era desarrollador software. De ahí que Mary y Tom Poppendieck sean los pioneros en la aplicación del Lean al software; de hecho fueron los autores del libro que ha inspirado las ideas del Lean aplicado al desarrollo software (Poppendieck & Poppendieck, 2003).

Todo esto es paradójico ya que aunque la filosofía ágil rechaza que el proceso de desarrollo software sea un proceso de fabricación industrial tradicional (Garzás, 2011), los defensores de los proyectos ágiles han tenido una gran influencia de los métodos de fabricación de Toyota.

No obstante, la adaptación del “Lean Manufacturing” al desarrollo software ágil se produjo y se conoce como la metodología “Lean Software Development” que fue desarrollada por los mencionados Mary y Tom Poppendieck (Poppendieck & Poppendieck, 2003).

Lean Software Development puede resumirse en siete principios (explicados a continuación), que se describen por Mary y Tom Poppendieck en (Poppendieck & Poppendieck, 2003), y que son una adaptación al desarrollo software de los principios Lean:

6. Lean Software Development y Kanban

- **Eliminar desperdicios** (Eliminating Waste). Eliminar todo lo que no añade valor: código y funcionalidades innecesarias, retraso en el proceso de desarrollo de software, requisitos poco claros, burocracia, comunicación interna lenta, documentación innecesaria, etc.
- **Amplificar el aprendizaje** (Amplifying Learning). El desarrollo de software es un proceso de aprendizaje continuo. El mejor enfoque para encarar una mejora en el ambiente de desarrollo es ampliar el aprendizaje. Incrementando la retroalimentación mediante reuniones cortas, tanto los clientes como el equipo de desarrollo, logran aprender sobre el alcance del problema y buscan soluciones para un mejor desarrollo.
- **Decidir lo más tarde posible** (Deciding as Late as Possible). En el desarrollo de software los mejores resultados se alcanzan con un enfoque basado en que se pueden retrasar ciertas decisiones tanto como sea posible, hasta que éstas se puedan basar en hechos y no en suposiciones y/o pronósticos inciertos.
- **Entregar lo más rápido posible** (Delivering as Fast as Possible). Cuanto antes se entregue el producto final sin defectos considerables, más pronto se pueden recibir comentarios que se incorporan en la siguiente iteración. Además cuanto más cortas sean las iteraciones, mejor es el aprendizaje y la comunicación dentro del equipo. Los clientes siempre valorarán la entrega rápida de un producto de calidad.
- **Capacitar y potenciar al equipo** (Empowering the Team). Los roles deben cambiar: a los directivos se les enseña a escuchar a los desarrolladores, de manera que éstos puedan explicar mejor qué acciones podrían tomar, así como ofrecer sugerencias para mejorar. En este punto se debe tener cuidado en considerar a las personas como recursos. Las personas necesitan algo más que una lista de tareas, necesitan motivación y un propósito para el cual trabajar: un objetivo alcanzable dentro de la realidad, con la garantía de que el equipo puede elegir sus propios compromisos.

6. Lean Software Development y Kanban

- **Construir con calidad** (Building Quality In). En el proceso de desarrollo se debe instaurar la meta de encontrar y corregir los defectos tan pronto como sea posible. Para asegurar la detección de estos defectos, es conveniente poseer un completo conjunto de pruebas a lo largo de todo el ciclo de vida del desarrollo y ejecutarlas automáticamente y de manera periódica.
- **Ver el todo** (Seeing the Whole). Los sistemas de software no son solamente la suma de sus partes, sino también el producto de sus interacciones. Cuanto más grande sea el sistema, más serán las organizaciones que participan en su desarrollo, más partes son las desarrolladas por diferentes equipos y mayor es la importancia de tener bien definidas las relaciones entre los diferentes proveedores con el fin de producir una buena interacción entre los componentes del sistema.

Por último, destacar que solamente cuando se aplican todos los principios de Lean se obtiene una verdadera ventaja de los mismos (Poppendieck & Poppendieck, 2003), siempre y cuando se combinen con un fuerte "sentido común".

6.2 Kanban

Dentro de la metodología Lean, surgió (también en Toyota) una técnica para gestionar el avance del trabajo, en el contexto de una línea de producción, que se denominó **Kanban**. Kanban es una palabra japonesa que significa “tarjetas visuales” (“kan” significa visual y “ban” tarjeta). La primera aplicación al desarrollo software de esta técnica, apoyándose en el paradigma Lean y en la teoría de restricciones (apartado 6.5), la realizó David Anderson en 2004 en lo que denominó “sistema Kanban de desarrollo software” (Anderson, 2009).

Kanban se basa en el desarrollo incremental, dividiendo el trabajo en partes (historias de usuario, tareas, etc.). Una de las principales aportaciones es que utiliza técnicas visuales para ver la situación de cada parte. Cada parte del trabajo se escribe en una tarjeta y se pega en un tablero. Las tarjetas suelen tener

6. Lean Software Development y Kanban

información variada, si bien, aparte de la descripción, debieran tener la estimación de la duración de la misma. Kanban se compone de las tres reglas (H. Kniberg & Skarin, 2010) que se muestran en la Figura 20 (la división previa del trabajo en partes se considera la regla 0 de Kanban):



Figura 20. Pilares del Kanban

6.2.1 Regla 1: Visualizar los estados

Es muy importante analizar los estados del trabajo en el proyecto. Todo proyecto consiste en una serie de fases hacia un producto que satisface una especificación. Una de las formas de visualizar estas fases es implementando tableros (también conocidos como muros o pizarras) con información visual para mejorar la comunicación en el equipo Kanban.

El tablero debe tener tantas columnas como estados por los que puede pasar la tarea (por ejemplo, en espera de ser desarrollada, en análisis, en diseño, etc.). A continuación se representa un ejemplo sencillo de tablero donde se muestran unas posibles fases que podrían aparecer. El espacio azul claro en cada columna permite ubicar los distintos ítems. Por lo tanto, en el ejemplo de la figura, existirán ítems "por hacer", ítems "en progreso" e ítems "terminados" (Figura 21).

6. Lean Software Development y Kanban

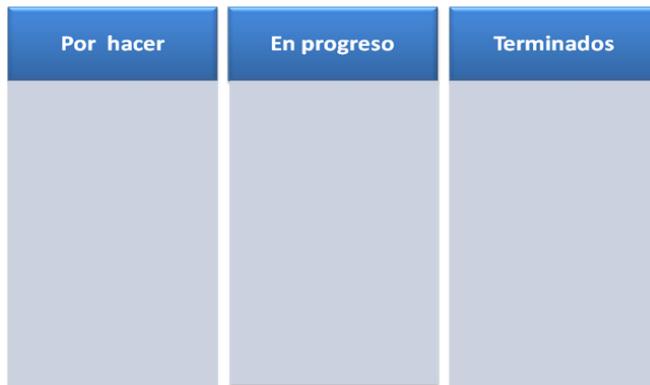


Figura 21. Ejemplo de tablero Kanban

Las diferentes fases del ciclo de producción o flujo de trabajo por el que pasa un ítem se deben decidir según el proyecto, no hay nada fijo. De esta manera en un tablero Kanban se puede representar cualquier ciclo de vida de desarrollo como, por ejemplo, el que muestra la Figura 22:

PETICIÓN DE TAREAS	SELECCIÓN DE TAREAS (5)	DESARROLLO (4)	PRUEBA (1)	TERMINADO
M N	H I J K L	D E F G 	C	A B

Figura 22. Ejemplo de tablero Kanban. Las tareas (tarjetas) van de la A a la N

El objetivo de esta visualización es que quede claro a todo el equipo el trabajo a realizar, en qué se está trabajando, que todo el mundo tenga algo que hacer y que permanezca clara la prioridad de las tareas.

6. Lean Software Development y Kanban

6.2.2 Regla 2: Limitar el trabajo en progreso

Quizás una de las principales ideas de Kanban es que el trabajo en curso debería estar limitado, es decir, que el número máximo de tareas que se pueden realizar simultáneamente en cada fase debe ser algo conocido. A ese número máximo de tareas se le llama límite “Work In Progress” o más conocido por sus siglas: WIP.

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 en esa fase está trabajando en cinco tareas no podrá trabajar en ninguna otra hasta que termine alguna de estas cinco. El ejemplo clásico que lo ilustra es el de la impresora. El WIP de una impresora es 1, sólo una página se imprime 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.

En la Figura 22 el WIP se ha colocado entre paréntesis debajo del nombre de cada fase. Se puede observar que el límite WIP para las pruebas es de 1 y para el desarrollo, de 4.

Como veremos a continuación, aunque el concepto de WIP pueda parecer algo muy simple, su ajuste tiene un gran impacto en la productividad un equipo, y saber ajustar el mejor WIP en cada fase del proceso productivo no es una tarea fácil.

6.2.2.1 Qué sucede cuando el WIP 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 existirán 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 tiempo que estarán en el tablero será mayor de lo necesario.

Un WIP ajustado a la baja, hará que cualquier problema que pudiera bloquear la finalización de una tarea se resuelva lo más pronto posible, ya que hasta que no se solucione no se podrá comenzar a trabajar en otra actividad.

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.

6.2.2.2 Qué sucede cuando el WIP es muy alto

Pero si el WIP 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 muchas actividades a la vez.

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

En este sentido se observa como **ajustar el límite WIP actúa como una señal de alerta** y avisa de un problema antes de que se nos escape de las manos y siga avanzando (esto es filosofía Lean “soluciona los problemas cuanto antes”).

A raíz de todo lo anterior hay que tener presente que **la esencia de los límites WIP es enfocar al equipo en finalizar tareas... más que en comenzar tareas.**

6. Lean Software Development y Kanban

6.2.2.3 Cómo saber el WIP correcto

Evidentemente **no hay una regla exacta que lo calcule**. Y además de depender del equipo, del producto que desarrolle, etc., para un mismo equipo **el WIP suele variar con el tiempo**, lo que implica un ajuste y mejora continua.

Pero sí que hay algunas heurísticas a observar, para detectar si tenemos algún problema con nuestro WIP (ver Tabla 5), 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 tiempo que estará una tarea en el tablero será alto.
- Si en una fase hay personas ociosas durante mucho tiempo el WIP 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? Se utiliza 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. Como conclusión final, cada equipo debe buscar su WIP.

Síntomas de WIP demasiado bajo:	Síntomas de WIP demasiado alto:
<ul style="list-style-type: none">• Miembros del equipo ociosos.• Las tareas están en estado “pendientes de empezar a ser realizadas” demasiado tiempo.	<ul style="list-style-type: none">• Hay tareas en las que no trabaja nadie.• Muchas tareas abiertas, y se cierran pocas.

Tabla 5. Síntomas de WIP incorrecto

6.2.3 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 se traslada a una gestión de proyectos software la pregunta sería: **¿Cuánto tiempo ha pasado desde que se ha pedido una funcionalidad y se ha entregado?**

Como el WIP limita la cantidad de trabajo se pueden originar casos de cuellos de botella, como por ejemplo, puede ocurrir que el equipo de desarrollo tenga que esperar 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 WIP para cada fase**. Mediante estas reglas, Kanban consigue **optimizar la capacidad de producción** del equipo de desarrollo.

Por tanto, en Kanban es muy importante medir lo que se conoce como el “**lead time**” (tiempo medio para completar un elemento) y es necesario optimizar el proceso para que el lead time sea tan pequeño y predecible como sea posible. Para verlo de forma gráfica, el **lead time** (Figura 23) es el tiempo que ha pasado desde que el ítem entra en el tablero Kanban hasta que llega a la última fase (terminado, en producción, etc.). Este es el **tiempo más importante desde el punto de vista del cliente**, receptor del producto final.

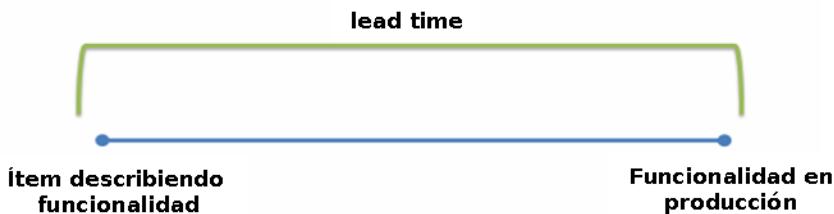


Figura 23. Lead time

6. Lean Software Development y Kanban

Asimismo, se utilizan otros tiempos incluidos dentro del lead time (Figura 24) como por ejemplo el denominado **cycle time**, que es el tiempo que pasa desde que se empieza a trabajar con el ítem hasta que llega a un estado finalizado; es decir es el **tiempo que se ha estado trabajando sobre la funcionalidad, tarea, etc.** Como se observa en la figura, lo ideal es conseguir que ambos tiempos sean similares mediante la evolución de los límites y del mismo tablero Kanban.

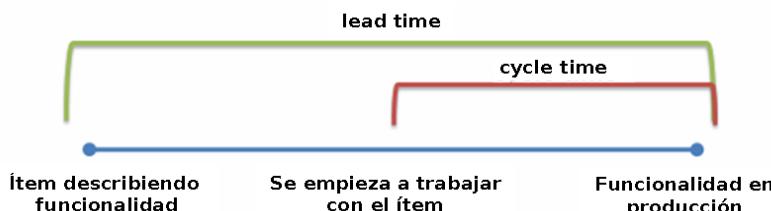


Figura 24. Partes del lead time

6.3 Ejemplo de flujo de desarrollo en Kanban

El jefe de proyecto de un equipo de desarrollo ha implementado un flujo de trabajo representado con **3 fases** que son “pendiente”, “en progreso” y “finalizado”. Por otro lado, un $WIP=2$ de la fase “En Progreso” indica que **el equipo no podrá trabajar en más de dos ítems** (en este caso tareas) a la vez. Prácticamente se realiza un ciclo de 4 pasos (ver Figura 25).

1. **Comienza el proceso.** Se puede comenzar a trabajar con 2 tareas ya que el estado del flujo “En Proceso” está libre.
2. **Se trabaja con las 2 primeras tareas.** Al estar limitado el WIP a 2, no se pueden introducir más tareas hasta que hayamos finalizado con alguna de las dos actuales.
3. Una vez terminada la primera tarea (o ambas en este caso), se pueden incluir más tareas de la fase “Pendiente”.
4. **Se vuelve a comenzar** el proceso.

6. Lean Software Development y Kanban

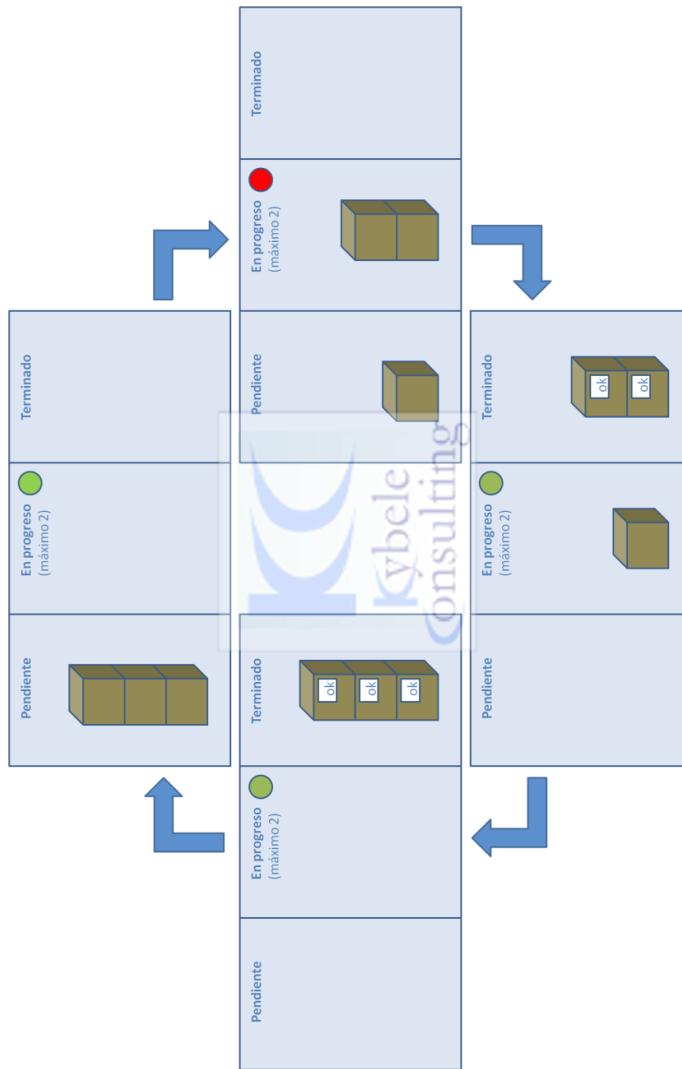


Figura 25. Ejemplo de flujo de desarrollo en Kanban

6. Lean Software Development y Kanban

6.4 Definición de un tablero más complejo

Como se ha comentado, uno de los primeros pasos al implementar Kanban en la organización, en un equipo de desarrollo, de mantenimiento o de soporte, es **definir el tablero Kanban**. El tablero tiene un fuerte efecto sobre la cultura de la organización. Aumenta la visualización y la comunicación mejorando la colaboración entre equipos. Ahora los equipos pueden ver como sus acciones influyen o interrumpen el flujo de trabajo general.

Para poder definir dicho tablero, será necesario **definir las columnas** que va a contener. 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 se quiere organizar. Por ejemplo, si se quiere organizar el desarrollo de correcciones para un producto software, se podrían incluir las columnas de la Figura 26.

PETICIÓN DE CORRECCIONES	SELECCIÓN DE PETICIONES	DESARROLLO	PRUEBA	INSTALACIÓN	PRODUCCIÓN

Figura 26. Tablero Kanban: Ejemplo de desarrollo de correcciones

El flujo de trabajo sería el siguiente: llega una petición de corrección y el jefe de proyecto la selecciona enviándola a la columna de "selección de peticiones". En el siguiente paso el equipo de desarrollo elegirá la siguiente corrección a desarrollar. Una vez que la corrección se encuentre desarrollada, quedará a la espera de ser elegida para realizar la actividad de pruebas.

6. Lean Software Development y Kanban

Por lo tanto, entre la etapa de desarrollo y prueba puede existir un tiempo de espera.

Por tanto, por lo expuesto en el párrafo anterior, es necesario dividir la columna de desarrollo y la columna de pruebas en dos partes (Figura 27), una para el trabajo que se está realizando (columnas de estado) y otra para el trabajo ya realizado (columnas de espera).

PETICIÓN DE PARCHES	SELECCIÓN DE PETICIONES	DESARROLLO		PRUEBA		INSTALACIÓN	PRODUCCIÓN
		ACTUAL	FINALIZADOS	ACTUAL	FINALIZADOS		

Figura 27. Tablero Kanban: Ejemplo de desarrollo de parches con subdivisiones

Lo más importante aquí es que el límite WIP es la suma de ambas columnas. 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 subcolumna "finalizados". Esta forma de trabajar sirve para evitar los cuellos de botella.

6.4.1 Las columnas de espera (buffer)

Es frecuente tener **columnas de espera (buffer)** para asegurar la fluidez entre dos flujos consecutivos del proceso.

Por ejemplo, puede ocurrir que la velocidad de desarrollo no sea exactamente igual que la velocidad de realización de pruebas. Por lo tanto, allí, sí es necesario un buffer para que absorba la variabilidad.

6. Lean Software Development y Kanban

La necesidad de este buffer disminuirá una vez que el proceso vaya madurando. Por tanto, con una correcta visualización de este tipo de columnas en la pizarra se puede establecer la necesidad real de cada una de estas columnas, y, **con el tiempo, ir ajustándolas.**

Finalmente, las columnas del tipo “to do”, o, en el caso del ejemplo de la Figura 27, “petición de correcciones” también son columnas de espera, ya que el cliente no siempre está disponible para decirnos qué hacer.

El objetivo final al crear un tablero Kanban es tener un proceso fluido a lo largo del proyecto minimizando el lead time. Por lo tanto, es necesario preguntarse a menudo: ¿Qué columna debería tener el tablero Kanban? Se debería **comenzar con una pizarra simple e ir agregando columnas en caso de ser necesario.**

6.5 Kanban y los cuellos de botella

Como se ha comentado en el apartado 6.2.2, una de las claves para la utilización correcta de Kanban es la **definición y evolución de los límites de trabajo en progreso o desarrollo (WIP)**. Supongamos que se utiliza el tablero descrito anteriormente y existe un problema en las pruebas (Figura 28). 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.**

6. Lean Software Development y Kanban

TABLERO KANBAN

PETICIÓN DE PARCHES	SELECCIÓN DE PETICIONES	DESARROLLO		PRUEBA		INSTALACIÓN	PRODUCCIÓN
		ACTUAL	FINALIZADOS	ACTUAL	FINALIZADOS		
G H	F		B C D E	A			

Figura 28. Cuello de botella en Kanban

La Figura 28 es un claro ejemplo de cuello de botella en la etapa de pruebas. En estos casos se recomienda que los miembros del equipo de desarrollo puedan ayudar; de hecho podían haberlo hecho antes, en el caso de que algún integrante del equipo quedara desocupado. 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** y ayudar en lo posible.

6.5.1 Teoría de las restricciones en Kanban

La teoría de restricciones (o teoría de limitaciones) es una filosofía de gestión que fue creada por Eliyahu M. Goldratt en 1984 en su libro titulado *The Goal*. La teoría de las restricciones, aplicada a la fabricación, se basa en la premisa de que una planta de fabricación será tan rápida como el proceso más lento de la cadena. De esta manera el valor de una cadena de fabricación es el valor del eslabón más débil de la misma (Anderson, 2003). Por tanto, la teoría de restricciones tiene como principal objetivo detectar los cuellos de botella del proceso y una vez identificados, se deben enfocar todos los esfuerzos de gestión e inversión económica en aliviar dichos cuellos de botella con el fin de mejorar el rendimiento del proceso.

6. Lean Software Development y Kanban

La esencia de la teoría de las restricciones se basa en cinco puntos correlativos de aplicación:

1. Identificar las restricciones (o cuellos de botella) del sistema o proceso.
2. Decidir cómo resolver esos cuellos de botella.
3. Subordinar todo a la decisión anterior, es decir, alinear a la organización y al sistema en su conjunto para apoyar la decisión tomada anteriormente.
4. Solventar el cuello de botella.
5. Si en los pasos anteriores se consigue resolver el cuello de botella, regresar al paso 1.

Mediante los tableros Kanban, los cuellos de botella se hacen visibles fácilmente como se observa en la Figura 28, donde se encuentra un claro ejemplo de cuello de botella en la etapa de pruebas, facilitando de esta manera la aplicación de la teoría de restricciones. Esto se produce porque en algún momento se ha llegado al máximo dado por el WIP en la etapa de desarrollo y el equipo no puede seguir trabajando.

En este momento, solventar el cuello de botella se debe convertir en la principal prioridad del equipo de trabajo (premisa de la teoría de restricciones). En estos casos se recomienda que todos los miembros del equipo puedan ayudar (en el ejemplo, aunque no pertenezcan al equipo de pruebas); de hecho podían haberlo hecho antes de que se originase el cuello de botella, en el caso de que algún integrante del equipo quedara desocupado.

6.6 Comparando Kanban y Scrum

Kanban no es una técnica específica de desarrollo software, su objetivo principal es gestionar de manera general como se van completando tareas. Sin embargo, en los últimos años se ha utilizado en la gestión de proyectos de desarrollo software, a menudo con metodologías ágiles como Scrum (lo que se

6. Lean Software Development y Kanban

conoce como “Scrum-ban”, el cual que se verá con más detalle en el apartado 6.6.1). Las diferencias que proporciona Kanban a los proyectos ágiles respecto al resto metodologías ágiles serían las siguientes:

- Kanban es **más adaptativo** lo cual significa que conlleva menores restricciones a la hora de adoptar las técnicas en la organización, facilitando al jefe de proyecto **más opciones y posibilidades de personalizar la metodología utilizada en el proyecto**.
- Aunque Kanban se basa en el ciclo iterativo, el tiempo fijo por iteración no es obligatorio. Se da **mayor prioridad a tener las tareas de la iteración terminadas**, lo cual no coincide con métodos ágiles como Scrum que sí obligan a terminar la iteración en el tiempo estimado.
- Kanban ofrece **la posibilidad de introducir nuevas tareas una vez comenzada una iteración** para adaptarse a las nuevas necesidades de los clientes.
- En Kanban el tamaño del equipo no está especificado y **puede estar especializado** en cada estado del proceso de la iteración.
- Por último, tanto los diagramas como la **documentación se usarán en la medida de su utilidad**. Se documentará y se realizarán diagramas siempre y cuando mejoren la eficiencia y la calidad del producto.

Como se acaba de detallar, Kanban presenta diferencias con otras metodologías ágiles sobre todo por la flexibilidad que ofrece. En (H. Kniberg & Skarin, 2010) se enumeran las diferencias principales entre Scrum y Kanban, que se han reflejado en la Tabla 6.

6. Lean Software Development y Kanban

Características	Scrum	Kanban
Iteraciones con tiempo fijo.	Obligatorio	Opcional
Compromiso del equipo con el trabajo de una iteración.	Obligatorio	Opcional
Estimación	Obligatoria	Opcional
Métrica utilizada para la mejora del proceso y de la tarea de planificación	Velocidad	Lead time
Roles	Product Owner, Scrum Master y el Equipo de desarrollo	
Equipos multifuncionales	Obligatorio	Opcional, se permiten equipos especializados
Tamaño de las tareas	Deben ser finalizadas en un Sprint	Sin restricciones
Gráficos obligatorios	El gráfico Burn-Down	No existe un gráfico en particular recomendado
Limitación del trabajo en progreso	Limitado implícitamente a partir del Sprint	Limitado explícitamente a partir del WIP
Agregar una tarea en una iteración	No es posible	Sí, en tanto exista capacidad (dada por el WIP)
Las tareas de una iteración pertenecen a:	Un equipo específico	Se pueden compartir entre equipos o personas
Momento en que son borradas las pizarras	Al finalizar cada Sprint	Nunca

Tabla 6. Diferencias entre Scrum y Kanban (H. Kniberg & Skarin, 2010)

6. Lean Software Development y Kanban

Como se puede ver en un ejemplo de (H. Kniberg & Skarin, 2010) indicado en la Figura 29 Scrum limita implícitamente a 4 las tareas que pueden ser realizadas (WIP implícito), ya que sólo existen 4 tareas en el Sprint Backlog. Pero no **limita el máximo de tareas por columna**. En cambio, la mayoría de los equipos ven útil limitar las tareas por columna, y cuando esto se hace el tablero Scrum se convierte en un tablero Kanban.



Figura 29. Scrum limita la cantidad de tareas por iteración, Kanban limita la cantidad de trabajo en curso

Tanto Scrum como Kanban utilizan un sistema de planificación que se corresponde con el principio del “justo a tiempo” (traducción del inglés just in time) de la filosofía Lean¹⁰.

Las similitudes entre Scrum y Kanban son las siguientes:

- Ambas tienen en cuenta la filosofía Lean y el agilismo.
- Ambos utilizan la planificación
- Ambos limitan el trabajo en progreso, aunque de manera diferente.
- Ambos utilizan un mecanismo transparente de mejora del proceso

¹⁰ Just in time: principio de origen japonés para aumentar la productividad en las fábricas. Su objetivo es producir los elementos que se necesitan, en las cantidades que se necesitan y en el momento en que se necesitan, evitando pérdidas por acciones de gestión o fabricación de elementos innecesarios.

6. Lean Software Development y Kanban

- Ambos se enfocan en entregar productos software lo más temprano posible.
- Ambos se basan en equipos autoorganizados.
- Ambos requieren dividir el trabajo en tareas más pequeñas.
- Ambos optimizan sus planificaciones basándose en datos empíricos (velocidad o lead time).

6.6.1 Scrum-ban

Una vez comparado Kanban con Scrum y teniendo en cuenta las experiencias al implementarlas y visitar empresas que utilizan prácticas ágiles cabe recalcar un mensaje: **muchos equipos que utilizan Kanban poco a poco descubren (o redescubren) el valor de las prácticas de Scrum**. De hecho, a veces los equipos utilizan Kanban en lugar de Scrum pensando que las buenas prácticas de Scrum no se ajustaban a su realidad. Estos mismos equipos, descubren que las prácticas Scrum exponían los problemas en sus desarrollos en lugar de generarlos. El verdadero problema era que habían estado utilizando Scrum desde un punto de vista demasiado "académico" en lugar de analizar sus buenas prácticas y adaptarlo a su contexto.

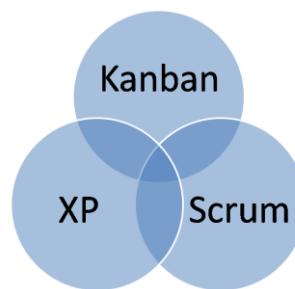


Figura 30. Las técnicas como Kanban, Scrum y otras metodologías ágiles tienen una filosofía y prácticas comunes

La utilización de Kanban con la metodología ágil Scrum da lugar a lo que se

6. Lean Software Development y Kanban

denomina **Scrum-ban**. Scrum-ban es una metodología que consiste en que partiendo de la implantación de Kanban, se llevan a cabo una secuencia de mejoras evolutivas, incorporando prácticas ágiles de Scrum gradualmente.

Básicamente se sigue el flujo de trabajo continuo como lo define Kanban, pero se incluyen elementos de Scrum de manera progresiva como por ejemplo, las reuniones diarias de 15 minutos, pequeñas retrospectivas con el afán de mejorar el proceso, etc.

De esta manera Scrum-ban supone una vía más accesible para introducir a los equipos de desarrollo al mundo de las **metodologías ágiles**, permitiendo tener un periodo de transición, con el objetivo de que los desarrolladores y demás “stakeholders” se sientan cómodos con la nueva metodología.

Scrum-ban puede ayudar a su vez a aquellas **organizaciones donde ya se ha adoptado Scrum para conseguir una mejora de eficiencia**, gracias al sistema de flujo continuo propio de Kanban, limitando el número de trabajos en curso (WIP), permitiendo cambios en el Sprint Backlog, etc. sin renunciar a sus técnicas habituales.

6.6.2 Trabajar varios proyectos en Scrum y Kanban

Tanto en Scrum como en Kanban y dependiendo la complejidad de la organización puede darse el caso de gestionar más de un proyecto en un mismo tablero y con un mismo equipo. Existen varias opciones para realizar esta práctica. Una estrategia, que pueden seguir los proyectos que utilicen Scrum está en focalizar a cada equipo en un proyecto determinado, y, por lo tanto, en un conjunto de historias de usuario determinada (Figura 31- izquierda). Otra estrategia es incluir historias de usuario de más de un proyecto en cada iteración (Figura 31- derecha).

6. Lean Software Development y Kanban

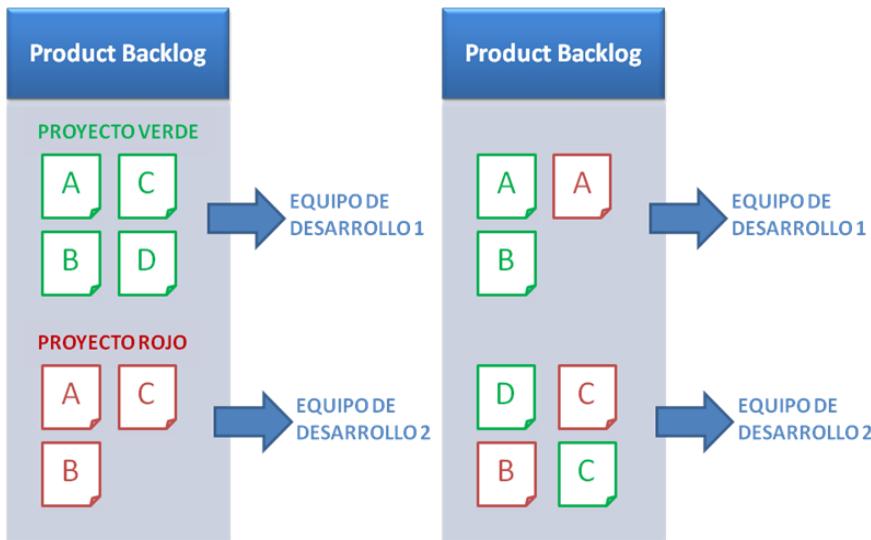


Figura 31. Product Backlog con historias de usuario de dos proyectos

Esto se puede dar de la misma manera en Kanban. Se pueden realizar tareas de más de un proyecto software en el mismo tablero Kanban. Por tanto, para mejorar la gestión de este tipo de tableros es conveniente distinguir diferentes colores en las tareas de los diferentes proyectos (Figura 32- izquierda), o mediante compartimentos o swimlines (Figura 32 – derecha).

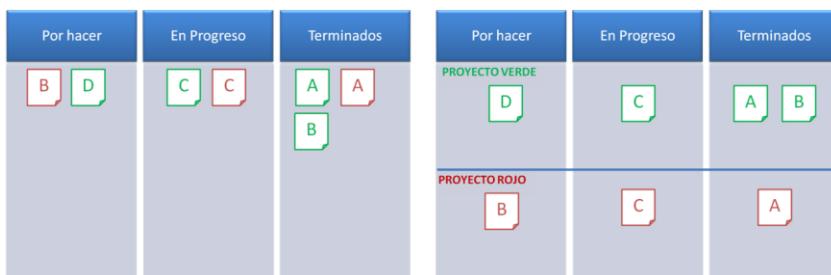


Figura 32. Es necesario distinguir las áreas de diferentes proyectos

7. Métricas y seguimiento de proyectos ágiles



7. *Métricas y seguimiento de proyectos ágiles*

“¡Dime cómo me medirás y te diré cómo me comportaré!” – Eli Goldrat

La información es la materia prima de la toma de decisiones, proporcionando criterios objetivos de gestión y seguimiento. Una de las principales actividades y que favorece la toma de decisiones en la gestión y seguimiento de proyectos ágiles es la medición de todo el proceso y del producto resultante. Sin embargo, un problema común en los proyectos tradicionales es la tendencia a medir en exceso, sin tener en cuenta su utilidad, generando costes, pérdidas de tiempo, lentitud en el desarrollo del proyecto, etc. De acuerdo con los principios del manifiesto ágil (ver apartado 2.3), en el caso de este tipo de proyectos el objetivo debe ser medir lo mínimo necesario para entregar el máximo valor al cliente.

Ahora bien, la pregunta sería: ¿Cuáles son las mediciones apropiadas? A lo largo de este capítulo se hablará sobre sus principales características y se describirán un conjunto de métricas e indicadores relacionados con proyectos ágiles.

Antes de entrar en detalle, se quieren dejar claros los conceptos de indicador y métrica ya que frecuentemente estos términos son usados indistintamente de manera incorrecta. Una métrica es una medida cuantitativa del grado en que un

7. Métricas y seguimiento de proyectos ágiles

sistema, componente o proceso posee un atributo dado (por ejemplo, número de pruebas unitarias en el producto), mientras que un indicador es la agregación o interpretación de una métrica o un conjunto de métricas para proporcionar una visión profunda de cualquier aspecto del proyecto (por ejemplo, productividad del equipo). Cualquier indicador que se obtenga en un proyecto debería ayudar a tomar decisiones y estar relacionado directamente con un objetivo. Hartmann y Dymond (Hartmann & Dymond, 2006) enumeran un conjunto de recomendaciones a la hora de elegir un indicador ágil:

- Reforzar los principios ágiles. Los indicadores deberán ser “ágiles” y no sólo por medir un proyecto que ha seguido una metodología ágil. Un indicador comunicado de manera deficiente, complejo de calcular o poco flexible es un ejemplo de indicador poco “ágil”.
- Medir resultados y no salidas. Por ejemplo se podría realizar la medición del trabajo que todavía falta para terminar una iteración; pero sin embargo, medir la cantidad de líneas de código desarrolladas, no proporcionaría ningún tipo de información valiosa para el equipo o el cliente.
- Seguir tendencias y no cifras. Al igual que en el caso de la estadística, es necesario trabajar con información agregada. No se debería trabajar con una granularidad menor que la proporcionada por el nivel de “equipo” o “iteración”.
- Trabajar con poca cantidad de indicadores. Si se trabaja con demasiada información, esta puede ocultar tendencias importantes.
- Fácil de recolectar. Debería ser tan fácil como presionar un botón para obtener listados de datos.
- Proveer información de manera periódica y frecuente. Para lograr diagnósticos rápidos, evitar desviaciones profundas o mejorar la motivación, los indicadores deben actualizarse diariamente, semanalmente o al final de cada iteración.

7. Métricas y seguimiento de proyectos ágiles

Como conclusión, los indicadores deberían estar siempre relacionados con el valor para el cliente producido por el equipo. De esta manera se pretende obtener software de calidad, con un desarrollo rápido y que aporte valor al cliente.

7.1 Tipos de indicadores

En general, un indicador puede estar conformado por métricas relacionadas con el desarrollo del proyecto y/o con el resultado (es decir, el producto software). Los indicadores asociados a la gestión y seguimiento de proyectos se centran principalmente en dar una visión de los siguientes aspectos:

- Productividad y resultados del proyecto: cualquier medición real de la productividad en el desarrollo de software tiene que estar basada en el valor de negocio entregado.
- Situación financiera: el cliente debe poder conocer cómo retorna su inversión y tomar decisiones fundadas sobre el proyecto al respecto.
- Valor y calidad percibida por el cliente: es recomendable que esté basado en un único indicador clave con el que el cliente pueda medir todos los objetivos del proyecto, de forma que permita guiar la toma de decisiones y la forma de invertir en el proyecto.
- Riesgos: el objetivo es identificar y evaluar los riesgos que surgen a lo largo del proyecto, con el fin de prevenir que ocurran, protegerse contra ellos o mitigar sus consecuencias.

Por último hay que destacar que en los equipos ágiles es común que los indicadores (y sus métricas) surjan ante una necesidad del equipo, como una forma de autocontrol para mejorar. Al incorporar nuevos indicadores, se debe intentar minimizar el coste e impacto que se añade al trabajo ordinario del equipo, así como minimizar el coste de recolección, proceso y presentación de

7. Métricas y seguimiento de proyectos ágiles

dichos indicadores. Por todo ello y como hemos dicho anteriormente, se debe escoger un número óptimo de indicadores y métricas que permita tomar decisiones sobre los resultados del proyecto y las necesidades del cliente.

7.2 Indicadores de productividad

A continuación se describirán algunos de los principales indicadores de productividad utilizados en proyectos ágiles de desarrollo software.

7.2.1 La velocidad

Este indicador es obligatorio en el seguimiento de proyectos ágiles y supone la cantidad de trabajo que un equipo puede hacer en una iteración, indicando el ratio con el cual el equipo convierte historias de usuario en incrementos potencialmente entregables. En el inicio de la iteración el equipo estima el trabajo, que será medido tomando algún tipo de escala. Esta escala es arbitraria, y puede tomar numerosas unidades: historias de usuario, tareas técnicas, puntos de historia, 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.

La principal importancia de esta medida es su consistencia ya que una vez conocida la velocidad del equipo se utilizará para estimar en la siguiente iteración, siendo así una herramienta de motivación para ser igual o más productivos en las sucesivas iteraciones.

En este campo de la gestión ágil, cuando se sigue la metodología Scrum toman gran importancia los gráficos BurnDown que son una representación del

7. Métricas y seguimiento de proyectos ágiles

trabajo que queda por hacer y a su vez, permiten comparar el progreso del proyecto o del Sprint con la planificación inicial. Al ser un gráfico propio de Scrum, normalmente 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 BurnDown se actualiza diariamente, representando el trabajo pendiente y el realizado hasta el momento, por lo que proporciona información muy útil para predecir las desviaciones. La Figura 33 muestra un ejemplo de este tipo de gráfico en el que la línea roja muestra la evolución real de la iteración mientras que la línea azul clara representa la estimación que se realizó al inicio.

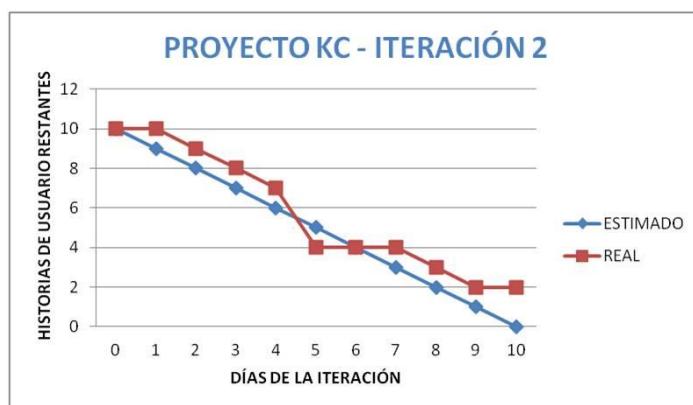


Figura 33. Ejemplo de gráfica de velocidad (gráfico BurnDown)

Por tanto la implantación de este tipo de gráficos (ya sea BurnDown u otro similar) proporciona al proyecto las siguientes posibilidades:

- Capacidad para predecir el trabajo restante para concluir una iteración.
- La posibilidad de extraer y utilizar la información en las planificaciones de las siguientes iteraciones.

7. Métricas y seguimiento de proyectos ágiles

Es necesario resaltar que no se deben tomar estos gráficos como un elemento de presión. Durante la iteración en curso, el equipo no debe intentar obtener un rendimiento parecido al esperado si con ello dejan de lado los aspectos de calidad. Idealmente la información proporcionada por el gráfico se utilizará para planificar mejor las futuras iteraciones.

7.2.1.1 Histórico de la velocidad del equipo

La medición de la velocidad del equipo se puede utilizar para evitar que el proceso de estimación sea poco realista. Teniendo en cuenta el histórico de la velocidad de desarrollo del equipo, se puede comprobar si las estimaciones han sido demasiado optimistas o pesimistas (M. Cohn, 2005).

En el siguiente ejemplo, se observa un histórico de la velocidad media por iteración de un equipo de trabajo (Figura 34). La velocidad suele venir determinada por el número medio de puntos de historia que ha realizado el equipo en las iteraciones de un proyecto. En el eje vertical se encuentra la cantidad media de puntos de historia desarrollados por iteración (o Sprint) y en el eje horizontal se indican los proyectos realizados.

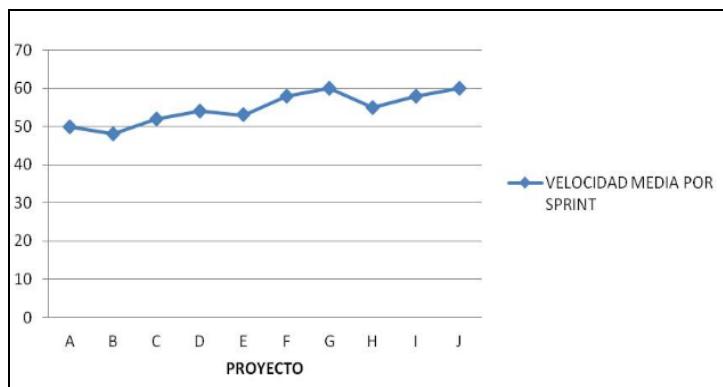


Figura 34. Histórico de velocidad del equipo de desarrollo

7. Métricas y seguimiento de proyectos ágiles

Esta información se puede utilizar desde dos puntos de vista dentro del proceso de estimación:

- **Antes del inicio de una estimación:** para que el equipo conozca su capacidad real de desarrollo.
- **Después de realizar la estimación:** para constatar que la estimación no se ha realizado de manera optimista o pesimista.

El histórico de la velocidad de desarrollo del equipo puede servir para realizar estimaciones posteriores más precisas. Esto también se conoce como método de estimación por analogía. Se utiliza la experiencia de los proyectos anteriores planificados, comparando el proyecto a estimar con proyectos similares terminados previamente.

7.2.2 La aceleración

Otro aspecto muy relevante a medir en un proyecto ágil respecto la productividad del equipo es la aceleración (Ambler, 2008). Este indicador viene determinado por las tasas de crecimiento (o decrecimiento) de las velocidades del equipo de desarrollo respecto a un periodo de referencia (ver Figura 35).

$$\text{Aceleración} = \frac{(\text{velocidad iteración (n)} - \text{velocidad iteración (0)})}{\text{velocidad iteración (0)}}$$

Figura 35. Fórmula de la aceleración

La inclusión de la medición de la aceleración en el seguimiento de un proyecto ágil reporta grandes ventajas:

- Es un indicador fácil de calcular, se escoge un rango de tiempo y se observan las tasas de crecimiento y decrecimiento de las velocidades.

7. Métricas y seguimiento de proyectos ágiles

- Es un indicador que no se falsea, ya que al ser equipos autogobernados, es una información importante para llevar bien a cabo su autogestión.
- Su automatización es sencilla, especialmente si se trabaja con herramientas como hojas de cálculo.

A continuación se muestra un ejemplo del cálculo de la aceleración en el que se supone que se tienen las velocidades de dos equipos de trabajo (X e Y) en proyectos donde se aplican metodologías ágiles. Se supone también que los equipos están compuestos por 8 desarrolladores y están trabajando en aplicaciones idealmente iguales en esfuerzo. De las últimas cinco iteraciones en la siguiente tabla se muestran las velocidades obtenidas por cada uno de los equipos.

ITERACIÓN	1	2	3	4	5
EQUIPO X	11	13	12	14	13
EQUIPO Y	58	62	60	63	63

Tabla 7. Velocidades por iteración de los dos equipos

Lo primero que hay que destacar es que Y en la primera iteración tiene una velocidad de 58 y X de 11. ¿Esto quiere decir que Y produce más rápido que X? ¡No!, no se pueden comparar las medidas de los dos equipos directamente ya que son unidades diferentes, y por tanto son estimaciones de tamaño y esfuerzo diferentes. Por ejemplo X podría usar como unidad de medida la referencia “historias de usuario terminadas / nombre” y el equipo Y “funcionalidades terminadas de historia de usuario / persona”.

Otro aspecto que se observa a simple vista es que entre las iteraciones 1 y 5 ha habido un incremento de la velocidad en ambos equipos, luego se puede afirmar que actualmente los dos equipos son más productivos. Ahora bien, debido a que usan unidades de medida diferentes: ¿Cuál ha sido el equipo que más ha incrementado su productividad en el transcurso de las 5 iteraciones?

7. Métricas y seguimiento de proyectos ágiles

Para responder a esta pregunta, aplicamos la fórmula de la aceleración (Figura 35):

$$\text{Aceleración} = \frac{(\text{Velocidad iteración 5} - \text{Velocidad iteración 1})}{\text{Velocidad iteración 1}}$$

$$\text{Aceleración } X = \frac{(13 - 11)}{11} = 0.18$$

$$\text{Aceleración } Y = \frac{(63 - 58)}{58} = 0.08$$

Por tanto, se puede concluir que el equipo X ha conseguido un mayor incremento de la productividad (esto podría deberse a que por ejemplo adoptaron buenas prácticas de desarrollo). El análisis de los aspectos que desembocaron en este aumento de la productividad, se puede extrapolar al resto de equipos de una organización y de esta manera se podría conseguir un aumento de la productividad general.

7.3 Indicadores de progreso de proyecto

A continuación se describirán algunos de los principales indicadores de progreso utilizados en proyectos ágiles de desarrollo software.

7.3.1 Gráficos tipo BurnUp

Para complementar los gráficos tipo BurnDown existen otro tipo de gráficos (en Scrum, denominados BurnUp) que reflejan cuánto avanzó el equipo hasta el momento, por lo general medido en el número de puntos de historia que se llegaron a completar hasta el momento, en relación al esfuerzo total necesario para terminar el proyecto. El gráfico BurnUp permite tener una perspectiva clara de cuánto se tiene por delante para completar el producto.

7. Métricas y seguimiento de proyectos ágiles

Por ejemplo, en Scrum la reunión de retrospectiva del Sprint (ver apartado 4.4) es un buen momento para comunicar esta información al cliente, y es el momento adecuado para actualizar este gráfico de acuerdo a las historias entregadas durante la última iteración. En la Figura 36 se muestra un ejemplo de este gráfico representando en el eje vertical las historias de usuario terminadas y en el eje horizontal los días de la iteración.

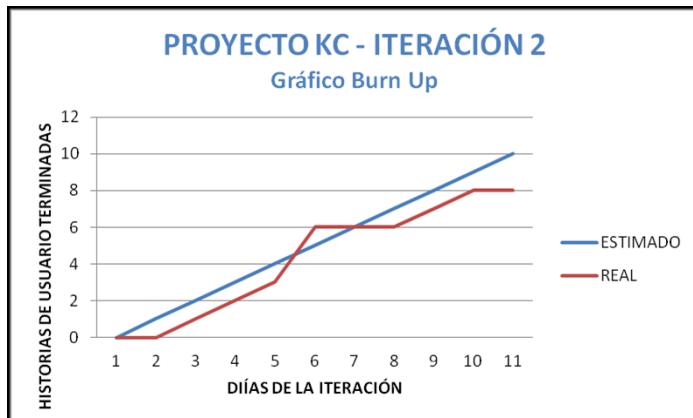


Figura 36. Gráfico BurnUp

A diferencia de los gráficos tipo BurnDown, la principal característica de los gráficos tipo BurnUp es que miden el progreso actual del proyecto.

7.3.2 Funcionalidades probadas y aceptadas (Running Tested Features, RTF)

Running Tested Features (RTF) mide la cantidad de funcionalidades que han pasado las pruebas de aceptación. Este indicador sugerido por Ron Jeffries (R. Jeffries, 2004), suele provocar un aumento de la agilidad y la productividad del equipo. Se construye en los siguientes pasos simples:

7. Métricas y seguimiento de proyectos ágiles

- Se identifican las funcionalidades con valor para el negocio. En este caso es importante definir las historias de usuario con buen un nivel de granularidad.
- Por cada funcionalidad se crean una o más pruebas de aceptación automáticas. El problema en este sentido es lograr la automaticidad de dichas pruebas y tener una cobertura suficiente.
- Cada vez que se agregan nuevas funcionalidades al entregable se vuelven a aplicar las pruebas de aceptación de todas las funcionalidades.

Tras finalizar este proceso y a lo largo de las diferentes iteraciones se va configurando un gráfico que indica la cantidad de funcionalidades que han pasado todas las pruebas de aceptación. Como se puede ver en la Figura 37, el primer gráfico (superior) representa una situación ideal, donde las funcionalidades que aportan valor se van introduciendo a lo largo de todas las iteraciones. Sin embargo, en el segundo gráfico (medio) las funcionalidades con valor para el negocio se han empezado a introducir hacia el final. Por último en el tercer gráfico (inferior), se puede ver que entre la quinta y la sexta iteración han existido problemas en el desarrollo, como por ejemplo, errores en funcionalidades entregadas previamente. Esto ha provocado que se haya visto disminuido el número de funcionalidades entregadas que han pasado las pruebas de aceptación.

7. Métricas y seguimiento de proyectos ágiles

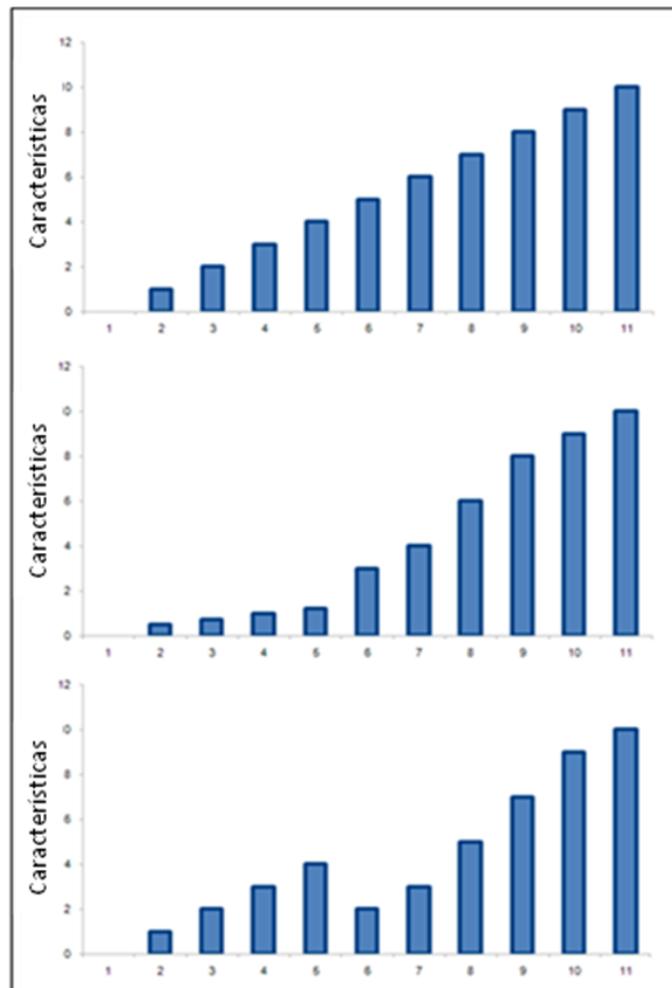


Figura 37. Gráficos RTF

7. Métricas y seguimiento de proyectos ágiles

7.4 Indicadores de valor entregado y retorno de inversión

En este caso se describirán algunos de los principales indicadores relacionados con la medición del valor entregado al cliente por parte del equipo de desarrollo.

7.4.1 EVM-Ágil

EVM-Ágil o AgileEVM (Sulaiman et al., 2006) no es más que la adaptación de una técnica tradicional de gestión de proyectos denominada EVM (Earned Value Management). El término “Earned Value” hace referencia a la medición del rendimiento del proyecto en relación con su línea base, es decir, con lo planificado. El resultado de esta técnica es indicar posibles desviaciones de tiempo y costes.

El objetivo principal del EVM-Ágil es dar a conocer a los clientes y al equipo de desarrollo en qué grado obtienen el valor esperado, cuál es el retorno de su inversión y con qué velocidad. El cambio para llegar a EVM-Ágil es la inclusión de los puntos de historia como medida para realizar los cálculos. Por lo tanto, en el EVM-Ágil se utilizarán las fórmulas de EVM con las medidas de un proyecto ágil. Algunas de las ventajas de usar EVM-Ágil son:

- Es sencillo y aprovecha las mediciones que se realizan habitualmente en las metodologías ágiles, especialmente en Scrum.
- Los indicadores que se obtienen aportan valor al equipo, los clientes, etc.

Para aplicar esta técnica es necesario realizar una planificación inicial de las iteraciones, las historias de usuario de cada iteración y el valor de cada historia de usuario.

7. Métricas y seguimiento de proyectos ágiles

MITO: “En los proyectos ágiles está prohibido definir un alcance”.

Aunque Scrum o eXtreme Programming no definen de manera explícita un alcance (descontando el Product Backlog), existen otras metodologías ágiles que lo hacen, por ejemplo Feature-Driven Development (FDD) o Crystal Clear.

Lo más importante y el primer paso de esta técnica es definir la unidad mínima de valor. Como ya se ha comentado, para los proyectos ágiles la unidad mínima de valor son los puntos de historia (en este capítulo se nombran con las siglas SP) que representan la cantidad de valor que aporta una historia al cliente/usuario.

Una vez definida la unidad mínima de valor, el siguiente paso es conocer las métricas históricas de proyectos anteriores, para compararlo con las mediciones actuales:

- BV: la velocidad histórica o cuantos puntos de historia se esperan completar por iteración.
- BC/SP: el coste histórico promedio de cada punto de historia.

El siguiente paso es calcular las métricas propias de EVM-Ágil en base a las mediciones del proyecto en curso:

- SR: el número de iteraciones planificadas para la siguiente versión o release del producto que se está construyendo.
- SPC: la cantidad de puntos de historia completados en la iteración.
- TSPC: el total de puntos de historia completados hasta el momento; es decir la suma de SPC de todas las iteraciones.
- VA: la velocidad actual, que se calcularía como TSPC/SR.

7. Métricas y seguimiento de proyectos ágiles

- CA: el coste actual de la iteración.
- TCA: el coste actual de todas las iteraciones hasta el momento; es decir la suma de CA de todas las iteraciones.
- CA/SP: el promedio del coste actual por punto de historia.

En base a estos cálculos se pueden obtener indicadores derivados. Los dos indicadores más propagados en la gestión y seguimiento de proyectos son:

- Rendimiento del Coste (CPI). La razón entre el coste histórico promedio de cada SP y el coste actual por cada SP. Es un indicador de la cantidad del valor pagado por el cliente que ha sido ganado (o recuperado) por el propio cliente. Si el índice CPI es mayor a 1, significa que se han gastado menos recursos para realizar los SP que lo planeado inicialmente.
- Rendimiento de la Planificación (SPI). La razón entre la velocidad actual y la velocidad histórica. Es un indicador de lo rápido que recibe el cliente el valor esperado. Si el índice es mayor a 1, significa que lo recibe a mayor velocidad de lo planeado al inicio del proyecto.

Los indicadores CPI y SPI pueden servir para calcular retrasos, reaccionar y tomar decisiones (B. D. Rawsthorne, 2008). Por ejemplo, si se han planificado 6 iteraciones para la siguiente versión del producto y actualmente se posee un CPI de 0,75; la cantidad de iteraciones necesarios será: $6/0,75 = 8$.

7.4.1.1 El índice Earned Business Value (EBV)

Asimismo se puede trabajar con otro indicador, el Earned Business Value (EBV) o Valor de Negocio Ganado que representa el porcentaje del valor del producto que está actualmente construido. Para ello se tienen en cuenta dos métricas:

7. Métricas y seguimiento de proyectos ágiles

- IVNG. El índice del valor ganado de una historia de usuario. Es el resultado de dividir el valor de negocio de la historia respecto de la suma de los valores de todas las historias de usuario de la release o versión del producto.
- VNG o EBV. El total de todos los IVNG para las historias de usuario completadas hasta el momento. Esta métrica es la que se grafica habitualmente.

Para poder obtener estas métricas, el equipo debe disponer de una estimación del valor de negocio para cada historia de usuario. La Figura 38 refleja la suma de puntos de valor de negocio entregados sobre el total de puntos estimados expresado en porcentaje, mostrando así el grado de avance del proyecto en términos del objetivo del negocio. El total de puntos de valor de negocio entregado es validado en cada reunión tras la iteración.

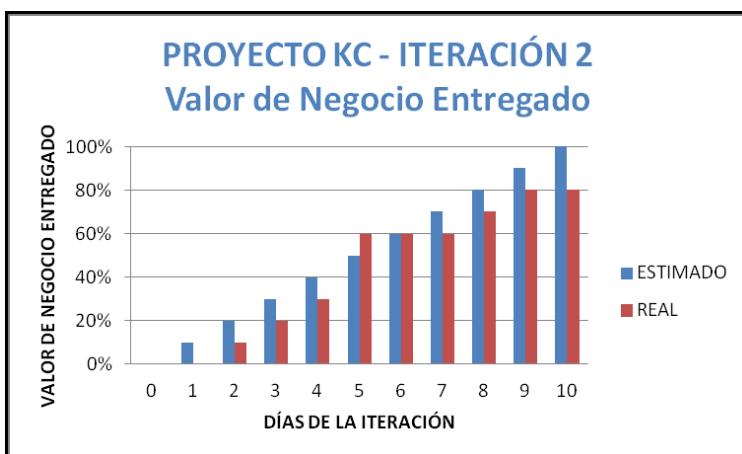


Figura 38. Gráfico de Valor de Negocio Entregado

Por lo general el valor entregado en un proyecto se comporta de acuerdo a una curva S (D. Rawsthorne, 2006) , de tal manera que al inicio del proyecto se suelen realizar las historias que tienen menos valor para el negocio. Una vez

7. Métricas y seguimiento de proyectos ágiles

terminada la construcción de la infraestructura del producto, se procede a realizar las historias de usuario que aportan un valor de negocio directo; y por último, se realizan tareas adicionales para completar el valor final de la versión del producto.

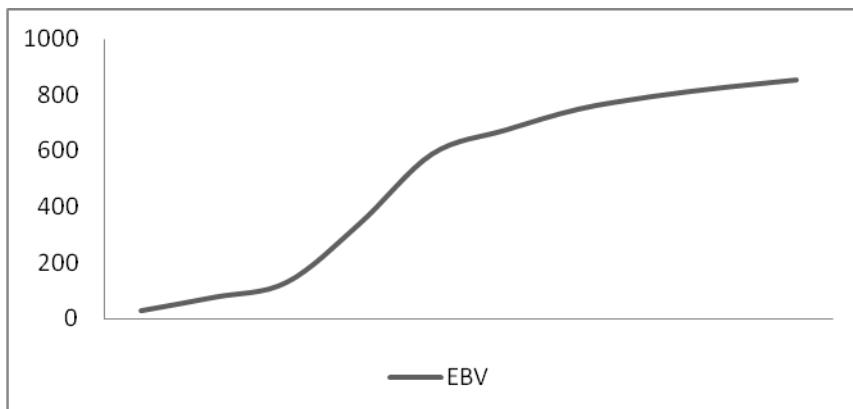


Figura 39. Curva S esperada de un proyecto ágil

7.5 Gestión de riesgos: Impediment Backlog

Al igual que en cualquier otro proyecto de desarrollo software, la gestión de los riesgos es una actividad esencial en la gestión y seguimiento de proyectos ágiles. En este tipo de proyectos, la identificación de riesgos es un ejercicio que casi todos los miembros del equipo deberían tratar de realizar continuamente, o por lo menos, colaborar en la identificación de posibles problemas.

La reunión de planificación de iteración (ver apartado 4.4) es un buen momento para expresar preocupaciones o para notificar e identificar posibles alertas. La definición de riesgos es una actividad que puede ser fácilmente realizada en colaboración entre el equipo y el cliente, y que perfectamente podría complementar la reunión de planificación.

7. Métricas y seguimiento de proyectos ágiles

Una de las herramientas usadas en este ámbito es el Impediment Backlog (Figura 40). Esta herramienta es realmente una lista priorizada de incidencias que se gestiona (en Scrum suele ser responsabilidad del Scrum Master) con el fin de solucionar los obstáculos que impiden el correcto progreso del desarrollo del equipo en la iteración.

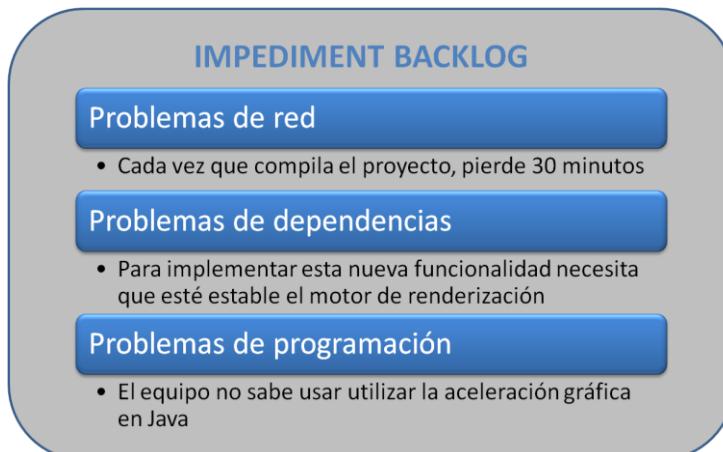


Figura 40. Ejemplo de Impediment Backlog

7.6 Cuadro de mando ágil

Todos los indicadores, métricas, gráficos y diagramas que se han mostrado con anterioridad en este capítulo se pueden agrupar en un mismo muro o tablero en lo que se denomina un cuadro de mando “ágil” (Figura 41), con el fin de centralizar la información que proveen.

El primer cuadro de mando integral fue realizado por Robert Kaplan y David Norton, en febrero del año 1992 (Kaplan & Norton, 1992). Entre sus principales características es que los cuadros de mando “ágiles” deberán ser fácilmente mantenibles y actualizables por el equipo de trabajo.

7. Métricas y seguimiento de proyectos ágiles

El principal objetivo de estos cuadros de mando es mejorar la comunicación, aumentando su claridad y brindando un estado actualizado del proyecto al cliente/usuario, y a los integrantes del equipo. Los indicadores englobados en un cuadro de mando ágil deberán estar asociados siempre a un objetivo concreto, y, es recomendable que se revisen en todo momento. Por ejemplo, en Scrum se podría revisar durante las reuniones de retrospectiva (ver apartado 4.4).

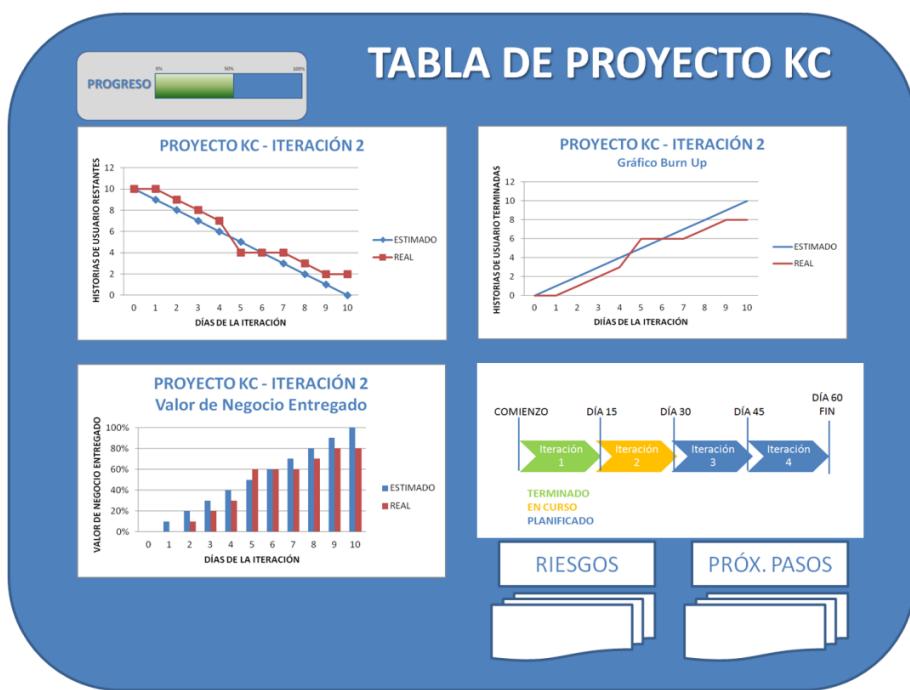


Figura 41. Ejemplo de cuadro de mando ágil

7. Métricas y seguimiento de proyectos ágiles

7.7 Diagramas de Gantt en proyectos ágiles

A la hora de implantar una metodología ágil en un proyecto, un tema que genera bastante debate y que no queríamos dejar sin comentar en este libro, es si las metodologías ágiles pueden convivir con los tradicionales diagramas de Gantt.

Como todo el mundo que ha trabajado en el desarrollo software (o en cualquier otro tipo de desarrollo) conoce, un diagrama de Gantt es una herramienta gráfica (tal vez la más popular) para realizar la planificación y seguimiento de un proyecto, cuyo objetivo es mostrar el tiempo de dedicación previsto para diferentes tareas o actividades a lo largo de un tiempo total determinado.

Sin embargo siempre que se recomienda no abandonar el uso de diagramas de Gantt en el mundo ágil, esta idea no es bien recibida ya que un diagrama de Gantt puro con sus fases, actividades, dependencias, etc., es considerado de todo, menos “ágil”. Este rechazo hacia este tipo de diagramas viene de una serie de contradicciones que presentan frente a los valores y principios ágiles descritos en el capítulo 2, y de las que se destacan las siguientes:

- Los diagramas de Gantt son predictivos y no adaptativos. Estos diagramas son eficientes para mostrar un plan de proyecto a largo plazo definido en las primeras fases del desarrollo, sin embargo los proyectos ágiles tratan de adaptarse al cambio en lugar de seguir un plan a toda costa.
- Los diagramas de Gantt suelen ser gestionados por los jefes de proyecto. En los diagramas de Gantt el jefe de proyecto es el que asigna las tareas lo que choca con la filosofía de equipos autogestionados de los proyectos ágiles.
- Los diagramas de Gantt tratan a las personas como recursos, lo que no cuadra demasiado con el valor ágil que habla de valorar a las personas sobre el proceso y las herramientas.

7. Métricas y seguimiento de proyectos ágiles

Si se quiere destacar una característica a utilizar de los diagramas de Gantt en los proyectos ágiles, es que permiten representar una gran cantidad de información en un espacio reducido. Por tanto, se podrían aprovechar estos beneficios visuales, sobre todo a la hora de representar las iteraciones de los proyectos ágiles, basándose en sus fechas, plazos y en el progreso de las funcionalidades desarrolladas en la iteración respecto a lo que se estimó al inicio de la misma. Esto puede ser útil a la hora de hablar con el cliente/usuario que estará más familiarizado con este tipo de diagramas que con los propios de las metodologías ágiles.

No obstante, Jeff Sutherland en (J. Sutherland, 2006) menciona esta posibilidad de usar diagramas de Gantt a la hora de tratar con clientes/usuarios acostumbrados a proyectos “tradicionales”. Sin embargo se reafirma en su decisión de rechazar el uso de estos diagramas cuando realizó la implementación inicial de Scrum, ya que en todos sus años de consultoría desde aquel 1993 no ha visto ninguna razón para cambiar esta idea. Sutherland concluye que si los equipos actualizan de manera diaria sus gráficos BurnDown, cualquier distracción a la hora de actualizar diagramas de Gantt es un esfuerzo inútil y que no aportará un valor relevante al proyecto.

7.8 Métricas asociadas al tablero Kanban

Por último se quiere presentar un conjunto de las métricas más usadas que permiten conocer el funcionamiento de los tableros Kanban. En (H. Kniberg & Skarin, 2010) se identifican las métricas de la Tabla 8 y cuáles son sus ventajas y desventajas:

7. Métricas y seguimiento de proyectos ágiles

Métrica candidata	Ventajas	Desventajas
Cycle time (ver apartado 6.2.3, pág. 83)	Fácil de medir. No necesita estimación. Métrica desde el punto de vista de la duración del proyecto para el cliente	No tiene en cuenta consideraciones de tamaño.
Velocidad total (teniendo en cuenta todas las tareas)	Indicador fácilmente medible que puede ser utilizado para mejorar la implantación de Kanban.	No es posible utilizarlo para prever fechas de entrega para tareas específicas.
Velocidad por tipo de tarea	Más preciso que la métrica de velocidad total.	Para mejorar su aporte debe tener en cuenta desde una necesidad del cliente hasta un entregable. Es necesario recolectar más información para tomar decisiones.
Longitud de las colas de espera	Un indicador de las fluctuaciones de la demanda. Fácil de visualizar.	No se puede distinguir si el problema es de la demanda de trabajo o de la capacidad.

Tabla 8. Métricas de Kanban.



8. **eXtreme Programming**

“No estar en producción es gastar dinero sin hacer dinero” – Kent Beck

La programación extrema o eXtreme Programming (XP) es una metodología ágil de desarrollo. En este capítulo se presentarán las características esenciales de la metodología y los puntos principales del ciclo de vida. Al igual que en otras metodologías ágiles, XP es una metodología adaptativa y centrada en las personas. A su vez, agrega buenas prácticas que abarcan más allá de la gestión de proyectos (como se ha podido ver en Scrum). Por ejemplo, incluye aspectos de pruebas (unitarias y funcionales), la construcción del producto software resultante, el mantenimiento y el cierre del proyecto.

Uno de los principios fundamentales en cualquier proyecto XP es la definición de cuatro variables: tiempo, alcance, coste y calidad. De estas cuatro, como máximo tres variables serán fijadas por el cliente o el jefe de proyectos mientras que la cuarta será fijada por el programador. Esto se encuentra estrechamente relacionado con la definición de contratos ágiles, como puede verse en el capítulo 9.

Las buenas prácticas de eXtreme Programming, explicadas con gran nivel de detalle en diferentes libros, como por ejemplo "eXtreme Programming Explained" de Kent Beck (K. Beck & Andres, 2004), son diez. A continuación se indicarán las características más relevantes de cada una de estas prácticas.

8. Extreme Programming

Asimismo, es muy recomendable repasar las explicaciones y especialmente sus fundamentos. Para ello, se pueden consultar libros como (Auer & Miller, 2002; Paulk, 2001; Stephens & Rosenberg, 2003) o (K. Beck & Andres, 2004).

8.1 Las buenas prácticas de XP

Así como otras metodologías o modelos de procesos, XP enumera un conjunto de buenas prácticas que son la base para un correcto funcionamiento de esta metodología. Hay que tener en cuenta que este tipo de metodologías ágiles, a diferencia de otras metodologías más tradicionales, se componen de buenas prácticas y no ofrecen un compendio de pasos rigurosos a seguir. A continuación se enumera cada una de ellas, manteniendo la terminología en inglés de aquellos términos sin una traducción directa al castellano.

- **Planning Game¹¹:** determinar rápidamente el alcance de la siguiente iteración. Para ello se utilizarán tanto las prioridades del negocio como las estimaciones técnicas. En este sentido, los desarrolladores tendrán mayor peso en los aspectos técnicos: las estimaciones ágiles (ver el capítulo 5, pág. 57), riesgos técnicos, orden en las actividades, etc. Y los jefes de proyecto o, en general, los encargados del negocio tendrán una perspectiva más global. Aportarán su experiencia en aspectos tales como el establecimiento del alcance o la priorización de las entregas.
- **Entregas pequeñas:** que los ciclos de construcción y despliegue de nuevas funcionalidades sean rápidos. Es necesario poner en producción rápidamente una nueva versión del sistema para planificar la siguiente iteración. Las divisiones no deben ser forzadas, una nueva entrega no puede estar conformada por "media funcionalidad".
- **Metáfora:** es una historia simple, conocida y entendida por todos, para guiar a los desarrolladores acerca de cómo funciona el sistema. Cada proyecto software estará guiado por una metáfora. En la Tabla 9 se re-

¹¹ Se mantiene la expresión en inglés por ser de uso más habitual en este entorno.

sumen las características de una metáfora.

Características	Pero...
Ayuda a entender los elementos básicos y sus relaciones	No es una explicación detallada y profunda
Se pueden utilizar ejemplos para aclarar la metáfora	Un ejemplo no debe ser tomado de manera literal.
La metáfora es una forma de reemplazar la arquitectura	La metáfora no es sinónimo de arquitectura
Fácil de elaborar y de comunicar	No es sólo para los programadores, clientes o jefes de proyecto

Tabla 9. Características de la metáfora de acuerdo con XP

Uno de los aspectos a tener en cuenta es que el cliente y el grupo de desarrolladores deben compartir esta “metáfora” para que puedan dialogar. Una buena metáfora debe ser fácil de comprender por el cliente y contener suficiente información para guiar la arquitectura del proyecto. La utilidad de la misma muchas veces es cuestionada, en ocasiones por la dificultad de implementarla.

- **Diseño simple:** construir el diseño lo más simple posible para dar respuesta a las necesidades/requisitos actuales del sistema. Las complejidades extras deberán ser eliminadas tan pronto como sean descubiertas. En la Tabla 10 se indican las características de un diseño simple, de acuerdo con (K. Beck & Andres, 2004).

8. EXtreme Programming

Características de un diseño simple
Permite la ejecución de todas las pruebas
No contiene lógica duplicada
Expone todas las intenciones importantes del desarrollador
Contiene la menor cantidad de clases y métodos posibles

Tabla 10. Características de un diseño simple de acuerdo con
(K. Beck & Andres, 2004)

En este sentido, los lenguajes de modelado no están prohibidos. Por ejemplo, XP y UML no son del todo incompatibles. Aunque XP no ponga todo el énfasis en los diagramas, la recomendación es clara: “utilízalo sólo si es útil”.

Muchos consideran que las prácticas ágiles deberían evitar cualquier tipo de documentación, sin embargo una de las premisas de cualquier metodología ágil es potenciar la comunicación entre los miembros del equipo. Antes de juzgar la utilización de diagramas como los de UML (que no es más que un lenguaje de modelado) es indispensable conocer el motivo por el cual es necesario el diagrama, el cual será siempre debería ser útil para mejorar la comunicación.

XP permite explorar diseños mediante diagramas antes de realizar el desarrollo. Esto puede resultar muy provechoso para mantener la dirección única en el proyecto. Entre las premisas descritas por Martin Fowler en el artículo “Is Design dead?” (M. Fowler, 2008), se resaltan tres aspectos a tener en cuenta:

- Diagramas pequeños: la cantidad de elementos capaces ser retenidos y analizados por las personas no suelen ser mayor que diez. Si se quiere favorecer la comunicación es necesario mantener esta premisa de simplicidad.
- Centrados en los detalles importantes: el código contiene la información detallada, los diagramas sólo contendrán información a alto nivel y recalcarán los aspectos más importantes.
- Los diagramas son simplificaciones, guías, no un diseño final completamente detallado. Debido a ese nivel de abstracción de los modelos y diagramas, si se realizan cambios en el diseño durante la construcción, no debiera ser necesario modificar los diagramas.
- **Pruebas:** los desarrolladores deben escribir pruebas unitarias de sus desarrollos. Es importante mantener un criterio de cobertura de pruebas y priorizar los métodos que deben ser probados. No es necesario realizar pruebas de cada método, pero las pruebas sobre métodos y clases claves en el desarrollo deben ser amplias. Los clientes también deben escribir pruebas para demostrar que la nueva funcionalidad se encuentra terminada.
- **Refactorización:** los desarrolladores se encargarán de reestructurar el sistema, sin cambiar su funcionalidad. De esta manera eliminarán código duplicado, disminuirán el acoplamiento, aumentarán la cohesión y agregarán flexibilidad. Ahora bien, ¿cuándo es el momento adecuado para realizar una refactorización? Algunas razones que nos indican que podemos estar en el momento adecuado para realizar refactorizaciones son: la necesidad de simplificar el diseño para hacerlo más mantenible, la imposibilidad de realizar una prueba unitaria de manera sencilla o la necesidad de introducir código duplicado porque el diseño actual así lo obliga. Existen síntomas o "malos hábitos de programación" (conocidos como "malos olores", "bad smells" en inglés) a tener en cuenta,

8. Extreme Programming

como por ejemplo clases con demasiada funcionalidad, métodos con un número elevado de parámetros, etc. Pueden encontrarse una lista de síntomas en (M. Fowler et al., 2000; Lanza & Marinescu, 2006).

- **Programación por pares:** se desarrolla por pares, con dos desarrolladores por cada ordenador. Existen dos roles en cada par de desarrolladores. Uno es el encargado de escribir el código fuente y las pruebas; mientras que el otro tiene que revisar dicho código y dichas pruebas, aportando una visión crítica/constructiva. De esta manera, el segundo desarrollador inspeccionará el código desde otra perspectiva, propondrá nuevas pruebas unitarias y revisará la simplicidad del diseño.
- **Propiedad colectiva del código:** el código es propiedad de todo el equipo. Cualquier desarrollador puede modificar cualquier parte del código fuente en cualquier momento. Esto significa que todos los desarrolladores tienen responsabilidad por todo el sistema, aunque no todos conozcan todas las partes con la misma profundidad.
- **Integración continua:** la integración de las diferentes partes del sistema, así como la ejecución de las pruebas unitarias que lo acompañan, deben ser realizadas al finalizar cada tarea de desarrollo. Es necesario que estas actividades sean realizadas varias veces al día (como mínimo 1 vez al día).
- **Semana de 40 horas:** una de las premisas más importantes para ayudar a la planificación es que la semana tiene solamente 40 horas útiles de trabajo. Por lo tanto, no debe trabajarse más allá de esas 40 horas. No terminar el trabajo planificado en el tiempo asignado es un síntoma de serios problemas en el proyecto. Una semana de trabajo extraordinario es lo máximo tolerable.
- **Clientes en la misma ubicación que el equipo:** incluir una persona del cliente en el equipo de desarrollo que esté disponible para resolver cuestiones. En este contexto, cliente es alguien que utilizará el sistema en producción.

- **Programación estándar:** los desarrolladores escribirán el código fuente de acuerdo con estándares y buenas prácticas de codificación que favorezcan la comunicación entre los miembros del equipo. En este sentido, esto no debe implicar una gran cantidad de documentación sobre cómo escribir el código fuente. Puede ser recomendable disponer de una herramienta automatizada de análisis del código fuente y de su estilo de codificación, con el fin de comprobar que las convenciones y buenas prácticas definidas se están cumpliendo.

Para finalizar esta sección queríamos expresar que cualquiera de las buenas prácticas indicadas anteriormente no son efectivas por sí solas. La fortaleza de estas buenas prácticas se encuentra en utilizarlas en conjunto.

8.2 El ciclo de vida de XP

El ciclo de vida de XP se basa en el modelo iterativo e incremental, de tal manera que en cada iteración se construyen un conjunto de historias de usuario (de manera similar a lo visto en Scrum, en el capítulo 3 pág. 25). De acuerdo con una de las buenas prácticas comentadas anteriormente, se realizan iteraciones cortas y se obtiene la respuesta y/o visión del cliente que utilizará las funcionalidades en un futuro. De esta manera los errores se detectan y se corrigen cuanto antes. En XP existe una fase de análisis inicial donde se planifican las iteraciones de desarrollo. Cada iteración incluye tareas de diseño, codificación y pruebas (no realizadas de manera secuencial). En la Figura 42 se puede ver un esquema del ciclo de vida.

- **Fase de exploración:** en esta fase, los clientes plantean las necesidades, usando las historias de usuario que serán implementadas durante la primera iteración. A su vez el equipo de desarrollo revisará las tecnologías, prácticas y herramientas a ser utilizadas durante el proyecto. Puede ser necesario explorar diferentes opciones de arquitectura, usando los denominados "architectural spikes". Un "spike" (traducido como "punta" o "clavo") consiste en el desarrollo de una aplicación pro-

8. Extreme Programming

totipo para probar conceptos y explorar diferentes soluciones, que por lo general será desechada una vez terminada la fase de exploración. Como resultado de esta fase se obtendrá la metáfora del sistema, el hilo conductor de las historias de usuario, el alcance y las priorizaciones.

- **Fase de planificación:** los desarrolladores y clientes se ponen de acuerdo para priorizar las historias de usuario y el alcance de la primera versión del sistema. La planificación de esta primera versión tendrá una duración de entre dos y seis meses. Se limita esta duración ya que los riesgos aumentan conforme va aumentando la duración de la fase de construcción de la primera versión. Aunque no existe un tiempo fijo para la planificación, esta debe tardar como máximo un par de días. La primera versión estará conformada por un número determinado de iteraciones de entre una y cuatro semanas. Durante la primera iteración se crea la estructura del sistema (por ello es importante elegir correctamente las historias de usuario), en la planificación, al igual que en la fase de exploración, también se pueden utilizar "spikes" para mejorar la estimación. De esta manera se tendrán estimaciones de mayor confianza.
- **Fase de desarrollo o iteración hacia la primera versión:** en cada iteración el cliente decide las historias de usuario que se realizarán, las cuales serán los requisitos de la siguiente versión. E idealmente al finalizar cada iteración el cliente habrá realizado las pruebas funcionales para asegurarse de que todo funciona correctamente. En caso de que existan errores, estos se solucionarán en las siguientes iteraciones. Por lo tanto, al finalizar la última iteración, el sistema estará en producción.
- **Fase de producción:** el pase de un sistema a producción requiere de pruebas de aceptación y comprobaciones adicionales. Los errores, las nuevas funcionalidades o las modificaciones que surjan de estas comprobaciones todavía pueden ser tenidas en cuenta y debe tomarse la decisión de si se incluyen o no en la versión actual. Durante esta fase,

las iteraciones pueden disminuir su duración, pasando por ejemplo de tres a una semana para aumentar la velocidad y el ritmo con el que se converge al final de la versión. A partir de aquí, las evoluciones no tendrán la velocidad con la que se ha realizado la primera versión.

- **Fase de mantenimiento:** es la fase en la que están la mayoría de los proyectos de desarrollo software actualmente, incluidos los que trabajan con XP. La evolución constante, refactorizaciones y mejoras del código fuente requieren de un mayor esfuerzo para satisfacer las tareas del cliente. Así, la velocidad del desarrollo puede disminuir una vez el sistema esté en producción y requerir la incorporación de nuevas personas o cambiar la estructura del equipo. El mantenimiento, por lo tanto, consiste en volver a realizar la planificación, el desarrollo en iteraciones y la puesta en producción.
- **Fase de muerte del proyecto:** un proyecto entrará en esta fase cuando el cliente no tenga más historias de usuario para ser incluidas en el sistema. Las necesidades del cliente pasan a ser aspectos no funcionales como el rendimiento del sistema. Este será el disparador para generar la documentación final del sistema y dejar de realizar cambios en la arquitectura. Por otro lado, un proyecto puede entrar en esta fase cuando los beneficios generados no sean los esperados por el cliente o los costes de mantenimiento sean excesivos.

8. EXtreme Programming

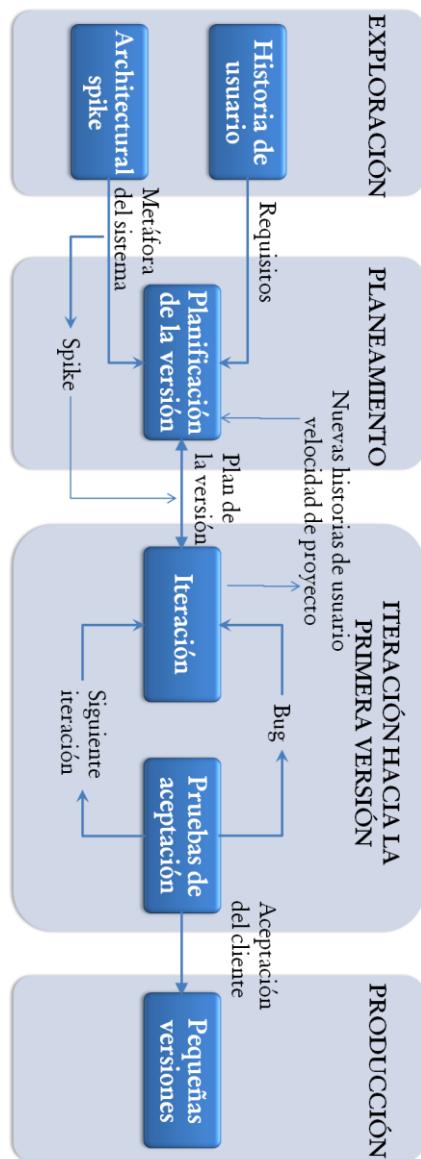


Figura 42. Esquema de un ciclo de vida XP

8.3 Algunos datos sobre el uso de XP

Entre los trabajos orientados a conocer la utilización de metodologías ágiles en el mundo empresarial existen algunos resultados muy interesantes. En el artículo de (Rumpe & Schröder, 2002) se presenta el resultado de una encuesta realizada a 45 proyectos de diferentes empresas. Según el citado estudio:

- El 100% de los desarrolladores utilizarían nuevamente la metodología.
- El riesgo más frecuente está relacionado con la falta de implicación del cliente.
- Las buenas prácticas relacionadas con los desarrolladores, como las pruebas unitarias o la integración continua, fueron las más utilizadas.
- Las buenas prácticas relacionadas con las interacciones entre personas, como las metáforas, la implicación del cliente y la programación por pares fueron las más conflictivas.
- Los factores claves de éxito fueron las pruebas, la programación por pares y las metáforas.

En lo que refiere a los motivos para utilizar XP (ver Figura 43), en gran parte, muchas de las organizaciones han elegido esta metodología debido a malas experiencias con otras metodologías, normalmente, de carácter más tradicional.

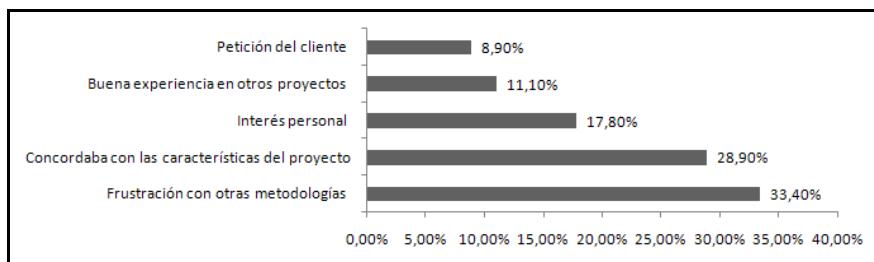


Figura 43. Motivos por los que se ha aplicado XP

8. Extreme Programming

Entre los factores claves para el éxito de un proyecto que utiliza XP (Figura 44), destaca el uso de pruebas, la programación por pares y un buen control de calidad.

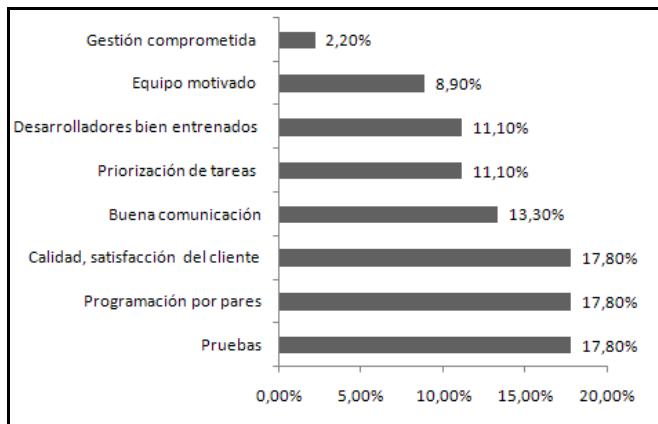


Figura 44. Factores claves para el éxito de un proyecto XP

Por último, de manera similar, se evaluó el uso de las diferentes buenas prácticas de XP. Valorando con 10 las más utilizadas y con 0 cuando no se utilizaban, la Figura 45 muestra una estadística de estas prácticas.

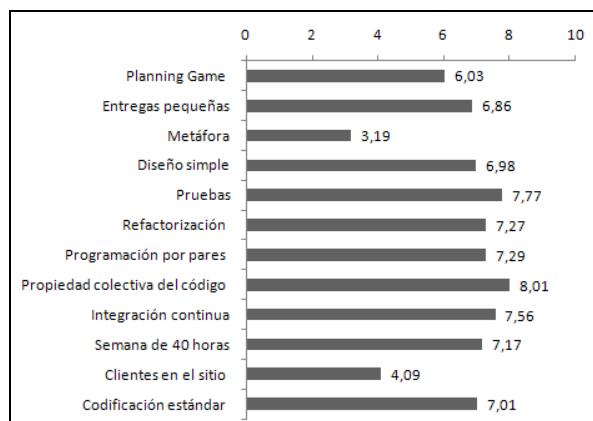


Figura 45. Uso de las buenas prácticas de XP

9. Buenas prácticas de desarrollo para proyectos ágiles



9. *Buenas prácticas de desarrollo para proyectos ágiles*

"Hay dos formas de escribir programas sin errores; sólo la tercera funciona" – Alan J. Perlis

Existen múltiples prácticas que facilitan el desarrollo software. Aunque estas prácticas pueden ayudar indistintamente a un proyecto de tipo ágil o tradicional, hay prácticas que se adaptan en mejor medida a un tipo de desarrollo u otro.

En este capítulo se van a describir algunas técnicas que se adaptan perfectamente a proyectos ágiles, facilitando en gran medida sus actividades de gestión. Se ha observado que implantando estas prácticas se puede obtener un alto beneficio en términos de fiabilidad, estabilidad y mantenimiento del producto, así como en la detección temprana de errores y problemas.

Aunque existen muchas prácticas consideradas claves en el desarrollo, se han querido destacar las siguientes: la importancia de poseer un entorno de integración continua, la refactorización del código como actividad esencial en el proceso de desarrollo y la generación y automatización de pruebas unitarias y pruebas “de humo” para garantizar que la realización de cambios no afecta a la estabilidad ni a la funcionalidad del producto software.

9. Buenas prácticas de desarrollo para proyectos ágiles

La recomendación de incluir estas prácticas en un proyecto de desarrollo software es tal que consideramos que el hecho de no introducirlas, pone en riesgo el éxito del proyecto.

9.1 Integración continua y “builds” automáticos

La **integración continua** (también conocida por su nombre en inglés, continuous integration) es una práctica cada vez más usada en desarrollos software. La primera publicación sobre integración continua fue realizada por Kent Beck (K. Beck & Andres, 2004). Esta práctica consiste en realizar construcciones e integraciones automáticas (conocidas como “builds” automáticos por su término en inglés) del código de un proyecto o de partes del mismo lo más a menudo posible. Esto se realiza principalmente con el fin de poder detectar los problemas en las primeras fases del desarrollo.

A la hora de implantar un entorno de integración continua, es necesario definir un ciclo de integración que se adapte a la naturaleza del proyecto. El ciclo que se muestra a continuación podría ser un ejemplo bastante completo (Figura 46):

1. El desarrollador compila el código y ejecuta las pruebas en su máquina de desarrollo y si no obtiene ningún error, sube los cambios al repositorio central de control de versiones (gestionado con herramientas como Subversion¹², Plastic SCM¹³, Git¹⁴ etc.).
2. El servidor de integración continua (gestionado con herramientas como Jenkins¹⁵, Continuum¹⁶, CruiseControl¹⁷, etc.) comprueba si ha habido cambios en el código fuente del repositorio.

¹² Subversion: <http://subversion.tigris.org/>

¹³ Plastic SCM: <http://www.plasticscm.com/>

¹⁴ Git: <http://git-scm.com/>

¹⁵ Jenkins: <http://jenkins-ci.org/>

¹⁶ Continuum: <http://continuum.apache.org/>

¹⁷ CruiseControl: <http://cruisecontrol.sourceforge.net/>

9. Buenas prácticas de desarrollo para proyectos ágiles

Si ha habido cambios, ejecuta los siguientes pasos del ciclo de construcción.

- a. Compila todo el código del proyecto. Generalmente mediante la ejecución de “builds” automáticos con herramientas como Maven¹⁸ o Ant¹⁹ ya que posibilitan que el proceso de compilación, dependencias con otras librerías, etc. se especifique en un fichero de configuración.
 - b. Ejecuta las pruebas unitarias, generando una alerta si alguna de las pruebas unitarias no se ejecuta satisfactoriamente.
 - c. Instala y despliega los productos generados (librerías, instalables, etc.) para que estén disponibles para todos los desarrolladores u otros equipos del proyecto.
 - d. Se ejecutan las pruebas de humo (que se conocen como “smoke test”): conjunto de pruebas que verifican las funcionalidades básicas del producto software y que se explican con detalle más adelante en el apartado 9.2.2. Cuantas más pruebas de humo automatizadas existan, mayor será la seguridad y fiabilidad de que el sistema está funcionando correctamente (Larman & Vodde, 2010). Aún así, estas pruebas tampoco deben ser excesivas para que el ciclo de integración continua no se demore demasiado en el tiempo (S. McConnell, 1996).
3. Si alguno de estos pasos falla, se detecta cual ha sido el desarrollador que ha provocado el fallo y se le notifica para que lo solucione rápidamente. Si todo ha sido correcto, se sigue con el resto de pasos.
 4. Se obtienen métricas (y/o indicadores) acerca de la calidad del producto (con herramientas como Sonar²⁰, Kemis²¹, PMD²², etc.). Este paso

¹⁸ Maven: <http://maven.apache.org/>

¹⁹ Ant: <http://ant.apache.org/>

²⁰ Sonar: <http://www.sonarsource.org/>

9. Buenas prácticas de desarrollo para proyectos ágiles

sería opcional, sin embargo, es muy aconsejable tener un control de la calidad del producto mediante unas métricas definidas de antemano con el fin de poder detectar si un cambio perjudica o mejora cualquier aspecto de la calidad del producto.

5. Se publican los resultados del proceso con el fin de que estén disponibles para todos los miembros del equipo de desarrollo. Normalmente estos resultados se publican en el sitio web del proyecto (se puede crear este sitio web también con herramientas como Maven¹⁸).

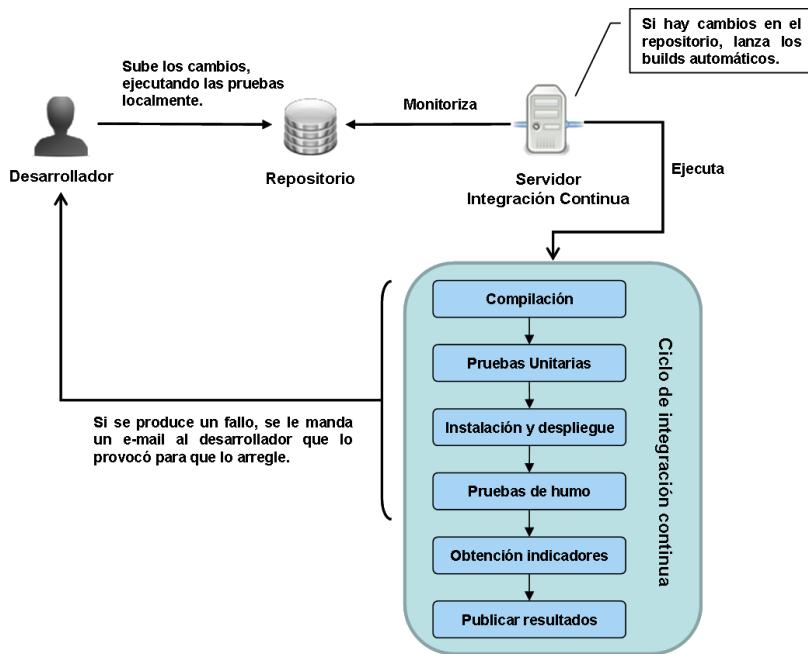


Figura 46. Estructura de la integración continua (Larman & Vodde, 2010)

²¹ Kemis: <http://iso25000.com/index.php/project-kemis.html>

²² PMD: <http://pmd.sourceforge.net>

9. Buenas prácticas de desarrollo para proyectos ágiles

De esta manera, poseer un entorno de integración continua similar al que se muestra en la Figura 46 tiene como ventajas:

- Se detectan los errores en las primeras etapas del desarrollo, minimizando los costes de resolución y evitando el caos a la hora de realizar la entrega definitiva.
- Se minimizan los costes de integración entre las funcionalidades que se van añadiendo al producto.
- Las pruebas automáticas se ejecutan continuamente con los nuevos cambios, pudiendo obtener un informe con el resultado de las mismas.
- Se incrementa considerablemente la visibilidad del avance del desarrollo y de la calidad del producto.
- Ofrece la posibilidad de controlar la estabilidad del producto y obtener métricas sobre su calidad de manera periódica.

A la hora de implantar un entorno de integración continua, se deben tener en cuenta los siguientes aspectos:

- La repercusión inmediata del resultado de subir código erróneo, al “romper” el ciclo de integración continua, tiene el riesgo de provocar que los desarrolladores no hagan tantas subidas a los repositorios como serían convenientes. Esto puede hacer que existan menos “copias de seguridad” del código fuente y que se retrase la integración de los cambios pudiendo provocar más conflictos.
- La integración continua puede ser un problema a nivel de organización ya que involucra a muchos equipos del proyecto: equipo de desarrollo, de gestión de configuración, de pruebas, de sistemas, jefes de proyecto, etc. Si no quedan claras las responsabilidades de cada uno de estos involucrados, puede dar lugar a discusiones interminables que eviten el avance del trabajo.

9. Buenas prácticas de desarrollo para proyectos ágiles

- Se necesita un esfuerzo inicial para montar el entorno y la generación de los diferentes ciclos de integración continua.

Respecto a la posibilidad de utilizar esta práctica en proyectos ágiles, como ya se ha comentado, la estrategia de la integración continua y los ciclos iterativos e incrementales que caracterizan a las metodologías ágiles son muy similares. A su vez, las subidas continuas promovidas por esta práctica evitan que se bloquee el código por parte de los desarrolladores, lo que está directamente relacionado con la buena práctica propuesta por XP (apartado 8.1) que expresa que “el código es de todos” (propiedad colectiva del código).

El principal problema de incluir la integración continua en un proyecto es que generalmente requiere un cambio en la mentalidad de los miembros del equipo. Los desarrolladores deben acostumbrarse a subir sus cambios de manera frecuente y a comprobar que no “rompen” la construcción (“build” automático) del producto en el ciclo de integración continua (Larman & Vodde, 2010). La presión que se puede ejercer (de manera directa o indirecta) sobre el desarrollador que “rompe” el build es perjudicial, ya que puede provocar que en futuras ocasiones se demore al realizar la integración.

Por último, conviene siempre recordar que la integración continua es una práctica que hace referencia a la actividad y no está directamente relacionada con una herramienta. Por lo tanto, es posible pensar que, conforme un proyecto va avanzando, el equipo de desarrollo está practicando integración continua solamente porque el entorno está implantado. Sin embargo, si los desarrolladores no siguen la disciplina de integrar sus cambios de manera frecuente y de asegurar que el ciclo de integración se ejecuta correctamente, no estarán siguiendo esta práctica de manera adecuada (Larman & Vodde, 2010).

9. *Buenas prácticas de desarrollo para proyectos ágiles*

9.1.1 *Herramientas de integración continua*

En este apartado vamos a realizar una comparativa de las que actualmente son las cuatro principales herramientas de integración continua disponibles para entornos Web. Cabe destacar que Hudson²³ y Jenkins son similares ya que Jenkins nace a partir de Hudson, cuando parte de sus creadores se desvinculan de ese proyecto al no estar de acuerdo con la compra de la herramienta por parte de la compañía Oracle®. En la primera columna de la Tabla 11 se listan las características a partir de las cuales se han comparado las herramientas.

Como diferencia más importante destacaríamos la manera de administrar y/o configurar las herramientas. Hudson y Jenkins posibilitan toda la administración y configuración de la herramienta a través de una potente interfaz Web, mientras que esta administración en CruiseControl se realiza editando archivos XML o a través de una consola no demasiado intuitiva. Continuum también provee una interfaz Web pero todavía existen determinados aspectos que tienen que ser configurados desde los archivos XML.

Hudson y Jenkins permiten establecer relaciones entre diferentes proyectos, de tal manera, que la ejecución de uno conlleva la ejecución de los que dependen de él y se integran (mediante plugins y mostrando sus informes) con gran cantidad de herramientas de ejecución de pruebas como JUnit²⁴, Emma²⁵, CppUnit²⁶, etc. (Continuum sólo se integra con algunas de estas herramientas).

Una limitación bastante destacable que tiene CruiseControl es que no posee un asistente de creación de proyectos tipo Maven, Ant, etc. Como principal desventaja de Hudson y Jenkins podemos especificar que es menos estable que las otras dos herramientas, libera versiones con más frecuencia y éstas a veces presentan errores, aunque son solucionados rápidamente.

²³ Hudson: <http://hudson-ci.org/>

²⁴ JUnit: <http://www.junit.org/>

²⁵ Emma: <http://emma.sourceforge.net/>

²⁶ CppUnit: <http://sourceforge.net/apps/mediawiki/cppunit/index.php>

9. Buenas prácticas de desarrollo para proyectos ágiles

Herramienta / Caracterís.	Hudson	Jenkins	CruiseControl	Continuum
Múltiples ejecutores	✓	✓	✓	✓
Administración / Configuración vía web	✓	✓	✗	✗
Autenticación / Autorización	✓	✓	✗	✗
Asistencia en creación de proyectos	✓	✓	✗	✗
Vistas (o organización) de proyectos	✓	✓	✗	✗
Relaciones entre proyectos	✓	✓	✗	✗
Soporte múltiples JDK	✓	✓	✗	✗
Soporte Maven	✓	✓	✓	✓
Soporte Ant	✓	✓	✓	✓
Configuración / Notificación vía e-mail	✓	✓	✓	✗
Bugs en versiones	✗	✗	✓	✗
Integración con herramientas de pruebas	✓	✓	✗	⚠
Precio	Gratis	Gratis	Gratis	Gratis

Tabla 11. Comparativa de las principales herramientas de integración continua

9. Buenas prácticas de desarrollo para proyectos ágiles

9.2 Pruebas en entornos ágiles

La realización de diferentes fases de pruebas durante el ciclo de vida de desarrollo evalúa la calidad del producto con el objetivo de poder mejorarlo identificando errores y problemas. Las pruebas son una verificación dinámica del comportamiento de un programa a partir de un conjunto finito de casos de pruebas, adecuadamente seleccionado.

A lo largo del ciclo de vida de un desarrollo software se deben realizar diferentes fases de pruebas como se representa en el conocido modelo en V (Figura 47). Este modelo muestra las relaciones entre cada una de las fases del ciclo de vida de desarrollo software con su fase de pruebas correspondiente.

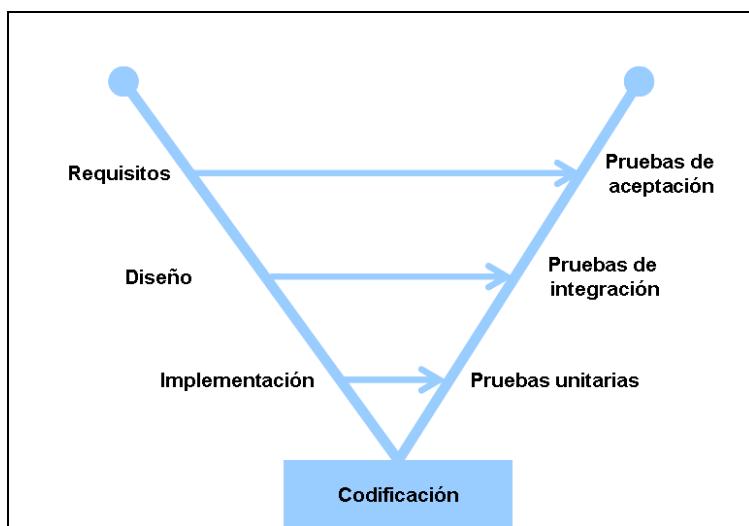


Figura 47. Modelo en V

No obstante, aunque todas estas fases de pruebas son importantes en el desarrollo software, en este capítulo se van a describir, como práctica clave para el desarrollo software ágil, las siguientes:

9. Buenas prácticas de desarrollo para proyectos ágiles

- La existencia de un conjunto de pruebas unitarias ejecutadas de manera frecuente por los desarrolladores en el entorno local y periódicamente en el entorno de integración continua.
- La existencia de un conjunto de pruebas “de humo” que verifican las funcionalidades básicas del producto software y que también se deberían ejecutar de manera periódica dentro del ciclo de construcción del entorno de integración continua. Estas pruebas “de humo” son una selección del conjunto de pruebas de integración y/o aceptación.

En los siguientes epígrafes se hace una descripción de las características y beneficios de generar pruebas unitarias y pruebas de humo en un proyecto de desarrollo software y de su aplicación en proyectos ágiles.

9.2.1 Pruebas unitarias

En un desarrollo software, las pruebas unitarias son las encargadas de aislar cada parte del programa y mostrar que las partes funcionan correctamente de manera individual. Estas partes aisladas de código se corresponden con los componentes más pequeños de una aplicación software que se pueden probar, generalmente, métodos o funciones.

Las pruebas unitarias principalmente son responsabilidad de los programadores para garantizar que su código fuente se comporta de acuerdo a lo esperado. Estas pruebas unitarias, aunque pueden ser manuales, idealmente deben estar automatizadas e integrada su ejecución en construcciones automáticas (“builds” automáticos lanzados en el entorno de integración continua).

El desarrollo de buenas pruebas unitarias puede ser una tarea compleja, de hecho Feathers en (Feathers, 2004) definió un conjunto de reglas que las pruebas unitarias deberían cumplir, de tal manera que una prueba no es una buena prueba unitaria si:

9. Buenas prácticas de desarrollo para proyectos ágiles

- Accede a base de datos o a ficheros del sistema.
- Necesita comunicarse a través de redes.
- No se puede ejecutar al mismo tiempo que el resto de las pruebas unitarias.
- Es necesaria la configuración del entorno para su ejecución.

Los problemas más comunes encontrados a la hora de implantar un desarrollo orientado a pruebas unitarias suelen ser los siguientes:

- La imposibilidad o la alta dificultad a la hora de realizar pruebas unitarias puede reflejar deficiencias serias en el diseño, por ejemplo un diseño demasiado acoplado. Esto generalmente se refleja en la dificultad de emular los componentes con los que interactúa la parte del software que se está intentando probar.
- Se tiende a realizar pruebas de integración en vez de pruebas unitarias: no se emulan correctamente los componentes con los que se interactúa, se accede a base de datos, etc. De esta manera, no se prueba cada parte del software de manera individual.
- Problemas para automatizar las pruebas complejas.

Así mismo existen beneficios. Generar pruebas unitarias como actividad simultanea a la implementación del código fuente, proporciona cinco ventajas básicas:

1. Favorecen el cambio: facilitan la refactorización del código (apartado 9.3). De hecho, para empezar un proceso de refactorización es obliga-

9. Buenas prácticas de desarrollo para proyectos ágiles

torio tener un completo conjunto de pruebas de regresión para garantizar que los cambios introducidos en el código no producen errores.

2. Aseguran el diseño: como se ha comentado, la imposibilidad o una gran dificultad a la hora de realizar pruebas unitarias suele reflejar deficiencias en el diseño. Generalmente la inclusión de patrones de diseño suele facilitar esta tarea. Por ejemplo, la dificultad de emular una clase en una prueba unitaria suele solucionarse con la inclusión del patrón inversión de control (Inversion of Control en inglés).
3. Simplifican la integración: permiten llegar a la fase de integración con un alto grado de seguridad de que el código está funcionando como se prevé.
4. Documentan el código: las pruebas unitarias son ejemplos de utilización del código.
5. Separan la interfaz y la implementación: dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro.
6. Los errores están más acotados y son más fáciles de localizar: las propias pruebas unitarias descubren estos errores.

Por último, como principales inconvenientes de las pruebas unitarias se podrían destacar:

1. Se genera más código fuente que se deberá mantener ya que en el fondo las pruebas unitarias son también código.
2. Los desarrolladores necesitarán una formación previa para poder empezar a desarrollar pruebas unitarias efectivas.

A pesar de poseer un completo conjunto de pruebas unitarias en un proyecto, es importante reconocer que no se podrán descubrir algunos tipos de errores existentes en el código: problemas derivados de la integración de los diferentes

9. Buenas prácticas de desarrollo para proyectos ágiles

módulos, de rendimiento o cualquier otro problema que afecte al sistema en su conjunto, serán responsabilidad de las fases de pruebas de integración, sistema y aceptación. Por tanto las pruebas unitarias no son efectivas si no van acompañadas de otros tipos de pruebas software.

Por último, no queríamos dejar sin mencionar una práctica actualmente muy popular en el desarrollo software y que está totalmente relacionada con las pruebas unitarias, el desarrollo orientado a pruebas (TDD – Test-driven development). TDD se basa en que el diseño se realiza a través de pruebas desarrolladas en pequeños ciclos (K. Beck, 2002):

1. Se implementa y automatiza un caso de prueba que define una nueva funcionalidad o una mejora a añadir en el código fuente.
2. Se implementa el código fuente necesario para pasar de manera satisfactoria dicho caso de prueba.
3. Se refactoriza el código adaptándolo a estándares y patrones.

El principal problema que supone implantar la práctica TDD en un proyecto de desarrollo es que implica “desaprender” la manera tradicional de programar, ya que en un primer lugar se desarrollan las pruebas unitarias para posteriormente implementar el código que cubre dichas pruebas. TDD es una de las prácticas más complejas de adaptar a los proyectos de desarrollo, pero sin embargo, es una de las mayores oportunidades para mejorar la calidad del diseño y del código (Larman & Vodde, 2010).

9.2.2 Pruebas de humo

Las pruebas de humo (“smoke test”) son una serie de pruebas de integración y/o aceptación funcionales que verifican las operaciones y funcionalidades básicas del producto software. Normalmente se ejecutan en la construcción automática del producto de manera periódica dentro de un entorno de integración continua (S. McConnell, 1996).

9. Buenas prácticas de desarrollo para proyectos ágiles

Este tipo de pruebas deberían ser desarrolladas por personas que no hayan estado involucradas en el desarrollo de la funcionalidad que se quiere probar. Para estos miembros del equipo, el mantenimiento de las pruebas de humo debe ser una de sus máximas prioridades.

Un buen conjunto de pruebas de humo debe probar el sistema ampliamente. No tiene porque ser una prueba exhaustiva, pero mediante ella se deben poder detectar los problemas más graves. El conjunto de pruebas de humo debe evolucionar conforme el sistema evoluciona, de tal manera que un conjunto de pruebas que en sus inicios duraba unos segundos, puede llegar a durar minutos o incluso horas (S. McConnell, 1996).

La existencia de un conjunto bien seleccionado de pruebas de humo ejecutado de manera automática y periódica lleva consigo los siguientes beneficios (S. McConnell, 1996):

- Se reduce el riesgo de baja calidad del producto: integraciones problemáticas acarrean el riesgo de generar un producto software de baja calidad. Con la ejecución de las pruebas de humo de manera periódica, los problemas en la calidad están más controlados, manteniendo el producto estable.
- Se facilita la detección temprana de errores funcionales.
- Sirven de motivación para los desarrolladores: mediante las pruebas de humo se puede mostrar la ejecución de alguna funcionalidad del producto resultante, motivando de esta manera a los miembros del equipo.

9. Buenas prácticas de desarrollo para proyectos ágiles

9.2.3 Pruebas unitarias y pruebas de humo en proyectos ágiles

Aunque el modelo en V visto con anterioridad se asocia más a modelos en cascada, todas estas fases de pruebas se pueden (y es muy aconsejable) realizar a su vez en los ciclos iterativos e incrementales característicos de los proyectos ágiles como se muestra en la Figura 48.

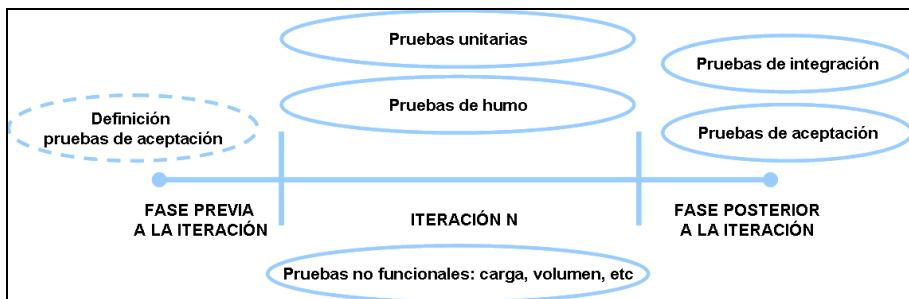


Figura 48. Estructura de pruebas en desarrollo iterativo

En un proyecto ágil, se desarrollan y ejecutan las pruebas unitarias asociadas al código fuente que cubre las funcionalidades planificadas para cada iteración. Mediante construcciones automáticas (idealmente en un entorno de integración continua) también se deberían ejecutar de manera periódica las pruebas de humo seleccionadas del conjunto de pruebas de integración y aceptación (Larman & Vodde, 2010). También se recomienda que se ejecuten las pruebas (de aceptación) no funcionales: pruebas de carga, de volumen, etc. durante la iteración, con el fin de detectar lo antes posible si algún requisito no funcional (de rendimiento, de disponibilidad, etc.) no está siendo cubierto.

Como se muestra en la Figura 48, se pueden realizar actividades previas al comienzo de la iteración (durante la planificación de la iteración), como definir, acordar con el cliente/usuario (en el caso de Scrum, con el Product Owner) y planificar qué pruebas de aceptación se deben cubrir en la iteración (este

9. Buenas prácticas de desarrollo para proyectos ágiles

concepto va asociado al desarrollo orientado a pruebas de aceptación o Acceptance-Test-driven development que se explica en el epígrafe 9.2.3.1).

Una vez finalizada la iteración, en la demo del producto resultante que se realiza al cliente/usuario se ejecutan de nuevo las pruebas de integración y aceptación, con el objetivo de demostrar que el producto resultante cubre todas las funcionalidades acordadas.

A la hora de ejecutar pruebas en proyectos ágiles es conveniente replantearse y adaptar algunos de los conceptos anteriormente vistos (Larman & Vodde, 2010):

- Se debe evitar la existencia de un departamento de pruebas independiente. Como en los proyectos ágiles se recomiendan los equipos pequeños (no más de 10 personas), autoorganizados y multidisciplinares, los roles tradicionales de analistas de pruebas y testers deben ser miembros del equipo. Estos miembros no deben estar especializados únicamente en el tema de las pruebas, pudiendo por ejemplo, realizar programación por pares con otros programadores del equipo (programar junto a un especialista en pruebas, muy probablemente mejorará la calidad del código).
- Los errores que se encuentran mediante la ejecución de las pruebas durante una iteración se deben corregir en el momento. Por otro lado, los defectos encontrados fuera de la iteración (generalmente por los usuarios finales del producto) si que se deben registrar y planificar su corrección en una iteración posterior.

9.2.3.1 Desarrollo orientado por pruebas de aceptación (Acceptance Test-driven development, A-TDD)

Al igual que para el desarrollo de pruebas unitarias era apropiada la práctica TDD, la práctica que se adapta al desarrollo en ciclos iterativos e incrementales

9. Buenas prácticas de desarrollo para proyectos ágiles

es el orientado por pruebas de aceptación (Acceptance Test-driven development, A-TDD). A-TDD, en los proyectos ágiles, se basa en la utilización de casos de prueba de aceptación para especificar las historias de usuario. Estos casos de prueba se crean mediante reuniones del equipo, del cliente/usuario y del resto de interesados.

Siguiendo el enfoque propuesto por A-TDD en proyectos ágiles, se pueden llevar a cabo los siguientes pasos (Larman & Vodde, 2010):

1. Antes de la iteración, se definen y acuerdan los casos de prueba de aceptación que cubren las historias de usuario que se desean implementar en la iteración (se suelen describir en el campo “criterios de aceptación” definido en el apartado 3.1). Estos acuerdos se realizan entre el equipo, el cliente/usuario y el resto de interesados.
2. En la reunión de planificación de la iteración, se definen como tareas las implementaciones del código que cubren cada uno de estos casos de prueba.
3. Durante el tiempo que dura la iteración, se implementa el código de las tareas definidas anteriormente, ejecutando los casos de prueba para comprobar que sus resultados son satisfactorios. Se puede tomar la decisión incluir alguno de estos casos de prueba en el conjunto de pruebas de humo con el fin de que se ejecuten de manera automática en el entorno de integración continua en ésta y en futuras iteraciones.
4. Una vez terminada la iteración, en la reunión de revisión de la iteración se vuelven a ejecutar las pruebas de aceptación. El objetivo es mostrar al cliente/usuario (y al resto de interesados) que las funcionalidades que fueron acordadas, efectivamente han sido cubiertas.

Como se ha podido observar, A-TDD se adapta adecuadamente a los ciclos iterativos e incrementales de los proyectos ágiles, guiando la planificación de las tareas de una iteración mediante casos de prueba de aceptación definidos entre

9. Buenas prácticas de desarrollo para proyectos ágiles

el equipo y el cliente/usuario. La ejecución de estos casos de prueba al finalizar cada iteración dan al cliente/usuario la visibilidad necesaria para comprobar que las funcionalidades se van implementando tal y como se acordaron.

9.3 Refactorización

La práctica de la refactorización (o “refactoring”) consiste en realizar una transformación al software preservando su comportamiento, modificando sólo su estructura interna con el objetivo de mejorarlo.

El término refactorización es de Opdyke, quien lo introdujo por primera vez en 1992, en su tesis doctoral (Opdyke, 1992). Por otro lado, en 2001 Tokuda y Batory (Tokuda & Batory, 2001) definieron las refactorizaciones como una transformación parametrizada a un programa preservando su comportamiento, modificando automáticamente el diseño de la aplicación y el código fuente subyacente. Fowler (M. Fowler et al., 1999), en cambio, dice que son cambios realizados en el software para hacerlo más fácil de modificar y comprender, por lo que no son una optimización del código, ya que esto en ocasiones lo hace menos comprensible, sin solucionar errores o mejorar algoritmos.

Importante: las refactorizaciones pueden verse como un tipo de mantenimiento preventivo, cuyo objetivo es disminuir la complejidad del software en anticipación a los incrementos de complejidad que los cambios pudieran traer.

Una de las razones para refactorizar es ayudar al código a mantenerse en “buena forma”, ya que con el tiempo los cambios en el software hacen que éste pierda su estructura, y esto hace difícil ver y preservar el diseño. Refactorizar ayuda a evitar los problemas típicos que aparecen con el tiempo, como, por ejemplo, un mayor número de líneas para hacer las mismas cosas o la aparición de código duplicado.

9. Buenas prácticas de desarrollo para proyectos ágiles

Existen incluso posturas, como la de la metodología ágil XP (apartado 8.1), que afirman que la refactorización puede ser una alternativa a diseñar, codificando y refactorizando directamente. Sin llegar a extremos tan radicales y poco recomendables, sí que es cierto que una buena refactorización cambia el rol del tradicional “diseño planificado”, pudiendo relajar la presión por conseguir un diseño totalmente correcto en fases tempranas, buscando sólo una solución razonable.

Otro aspecto importante de la refactorización es que ayuda a aumentar la simplicidad en el diseño. Sin el uso de refactorizaciones se busca en un primer momento la solución más flexible para el futuro, siendo estas soluciones, por lo general, más complejas y, por tanto, el software resultante más difícil de comprender y por ello de mantener. Además, ocurre que muchas veces toda esa flexibilidad no es necesaria, ya que es imposible predecir qué cambiará y dónde se necesitará la flexibilidad, forzando a incluir mucha más flexibilidad de la necesaria (síntoma de esto es la sobrecarga de patrones). Con la refactorización en lugar de implantar soluciones flexibles antes de codificar, se hace en fases posteriores.

En lo que refiere al proceso de refactorización, existen algunos consejos y buenas prácticas a tener en cuenta:

- No se debe añadir funcionalidad mientras se refactoriza. La regla básica de la refactorización es no cambiar la funcionalidad del código o su comportamiento observable externamente. El programa debería comportarse exactamente de la misma forma antes y después de la refactorización. Si el comportamiento cambia, entonces será imposible asegurar que la refactorización no ha estropeado algo que antes ya funcionaba.
- Un uso riguroso de las pruebas. Para comenzar un proceso de refactorización se debe disponer de un buen conjunto de pruebas, ya que permiten comprobar que una vez refactorizado, el software mantiene su funcionalidad.

9. Buenas prácticas de desarrollo para proyectos ágiles

- Refactorizar en diferentes momentos del ciclo de vida. Se aconseja refactorizar antes de añadir nueva funcionalidad (haciendo más fácil añadirla) o después (simplificando el código una vez introducida), cuando se necesita reparar un error o al revisar el código.
- Usar los bad smells (“malos olores” o “malos hábitos de programación”). Los “bad smells” pueden ser una buena ayuda a la hora de detectar dónde aplicar refactorizaciones (se puede consultar un catálogo de malos olores en (M. Fowler et al., 2000; Lanza & Marinescu, 2006)).

Por otro lado, hay que considerar que llevar a cabo la refactorización manual supondrá una tarea larga y tediosa. Por ello, desde hace años se ha estado trabajando en herramientas de refactorización, siendo “Smalltalk Refactoring Browser” de 1999, desarrollada bajo del marco de la Tesis de Roberts (Donald, 1999), la que se considera primera herramienta de refactorización. Pero, sin embargo, aún hoy, el problema de automatizar la refactorización sigue siendo complejo, ya que aunque hay refactorizaciones que apenas necesitan analizar la estructura del software, como por ejemplo, aquellas que se encargan de renombrar, la mayoría debe analizar y manipular el árbol del programa, y en ocasiones esto es complejo, por lo que aún queda camino por recorrer en la automatización de la refactorización.

Aunque hay varios catálogos de refactorizaciones, el más famoso es el de Fowler, que se mantiene en la página www.refactoring.com. Algunos ejemplos de las refactorizaciones que podéis encontrar en este catálogo son: *Add Parameter*, *Change Bidirectional Association to Unidirectional*, *Consolidate Conditional Expression*, *Extract Class*, *Introduce Null Object*, *Move Method*, etc. A continuación se muestra un ejemplo en UML de una refactorización de tipo *Extract Class*. Esta refactorización se realiza cuando una clase tiene un comportamiento que debería ser realizado por dos clases.

9. Buenas prácticas de desarrollo para proyectos ágiles

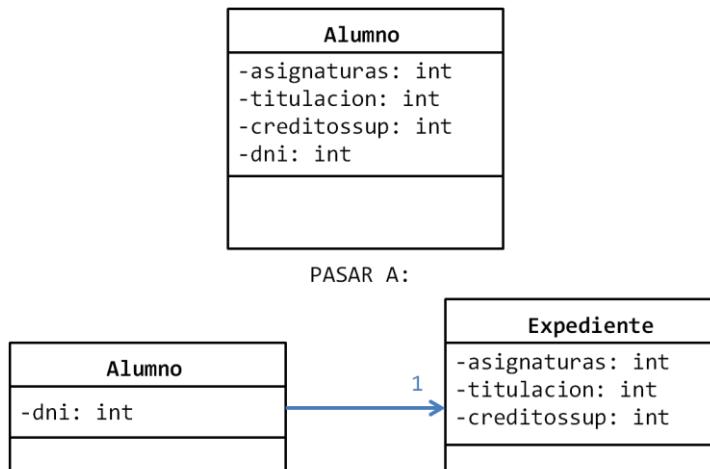


Figura 49 Refactorización *Extract Class*

9.4 Gestión de la configuración software

La gestión de la configuración software (GCS) define la manera en que se deben realizar diferentes procesos a lo largo del desarrollo del producto con el fin de controlar su evolución e integridad mediante la identificación de sus elementos, la gestión y el control del cambio, y la verificación, registro y presentación de informes (IEEE, 2004). Entre las actividades principales que se encuadran dentro de la gestión de configuración de software se destacan las siguientes:

- Identificación de líneas base del producto. Identificar y especificar donde se encontrarán los diferentes elementos que se asocian al producto: código fuente, ejecutables, manuales de usuario, etc.
- Definición de patrones de nombrado de versiones. Especificar cómo se nombrarán las versiones en las diferentes entregas del proyecto: ya sean entregas a pruebas de integración, a pruebas de aceptación, etc.

9. Buenas prácticas de desarrollo para proyectos ágiles

- Identificación, gestión y mantenimiento de librerías, sistemas y herramientas externas. Definir cómo se van a administrar todos los elementos de terceros que usa el proyecto, de quién es decisión del cambio de una versión en uno de estos elementos, etc.
- Definición de estrategias y políticas de GCS. Definir la manera de trabajar con los repositorios de control de versiones (gestionados con herramientas como Subversion²⁷, Plastic SCM²⁸, Git²⁹, etc.).

Este último punto es el que suele ser más complejo de definir en cada proyecto, según la naturaleza del mismo existen múltiples patrones y políticas de GCS que se adaptan en mayor o menor medida a cada tipo de proyecto. Y aunque existe un completo catálogo de estrategias, patrones y políticas de GCS que se puede consultar en (Appleton, B. Berczuk, S. et al., 1998), en el siguiente epígrafe, se explica una forma de trabajar descrita por Kniberg en (H. Kniberg, 2008) que ha resultado exitosa y que ha tenido una buena acogida en los diferentes proyectos ágiles en los que se ha implantado.

9.4.1 *Modelo de GCS aplicado con éxito en proyectos ágiles*

Antes de comenzar a describir el modelo, se quieren resaltar los objetivos que debe cumplir un modelo de control de versiones en un proyecto ágil:

- Debe posibilitar la detección rápida de errores. Los conflictos en el código y los problemas de integración deben descubrirse lo antes posible.
- Debe ser siempre versionable. Incluso después de una iteración problemática debe existir “algo” que se pueda versionar.

²⁷ Subversion: <http://subversion.tigris.org/>

²⁸ Plastic SCM: <http://www.plasticscm.com/>

²⁹ Git: <http://git-scm.com/>

9. Buenas prácticas de desarrollo para proyectos ágiles

- Debe ser sencillo. Todo el equipo usará el modelo a diario, por lo que las reglas, políticas y estrategias deben ser claras y sencillas.

El modelo propuesto por Kniberg (H. Kniberg, 2008) intenta cubrir estos objetivos a partir de los puntos descritos a continuación. Kniberg inicia la presentación de su modelo explicando el concepto de “realizada”, refiriéndose a cuando se considera que una historia de usuario ha sido realizada.

9.4.1.1 Concepto de “realizada”

El modelo se basa en el concepto de historias de usuario “realizadas” (Figura 50), aunque puede servir igualmente para las tareas en las que se descompone una historia de usuario. No obstante, para presentar el modelo, nos centraremos en las historias de usuario y no en sus tareas.

En este modelo, una historia de usuario se considera realizada y por tanto se pasa al estado “Realizadas”, cuando todos los miembros del equipo están de acuerdo en que esa historia de usuario está terminada y se podrá incluir en la siguiente versión sin que exista duda.



Figura 50. Concepto historias de usuario realizadas

9. Buenas prácticas de desarrollo para proyectos ágiles

9.4.1.2 Rama de historias de usuario realizadas

Debido al anterior concepto, surge la necesidad de poseer en el sistema de control de versiones, una rama que suponga la ubicación de las historias de usuario realizadas, que sea estable y a su vez versionable en cualquier momento del proyecto para pasar a producción. Esta rama se aconseja que se ubique en el “trunk”³⁰(tronco) o línea principal del repositorio de control de versiones (gestionado con herramientas como Subversion, Plastic SCM, Git, etc.).

La rama de historias de usuario realizadas (en este ejemplo, el “trunk” de la herramienta Subversion) va a tener las siguientes políticas:

- El “trunk” se puede versionar en cualquier momento.
- No se pueden subir (publicar) código de historias de usuario que no se quieren incluir en la siguiente versión.

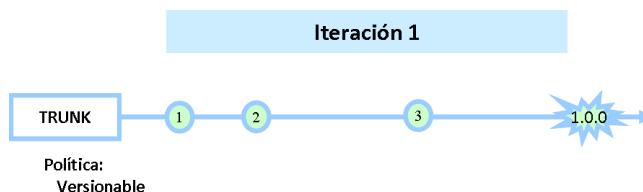


Figura 51. Rama de historias de usuario realizadas (en este caso, el “trunk”)

En la Figura 51, la línea representa el trunk. Cada “círculo” representa una “publicación” al trunk (integración de cambios procedentes de las ramas que veremos a continuación) y al final de la iteración, se genera la versión 1.0.0. Aunque generalmente la liberación de la versión se suele realizar al final de cada

³⁰ Usaremos la palabra en inglés trunk, por ser usada frecuentemente en este contexto. En el campo del desarrollo software, trunk se refiere a una rama del árbol de ficheros que está bajo el sistema de control de versiones y que suele ser la línea base sobre la que se realiza el desarrollo del proyecto.

9. Buenas prácticas de desarrollo para proyectos ágiles

iteración, si surgiese la necesidad de generar una versión durante la iteración, se podría llevar a cabo gracias a la política de que esta rama sea siempre estable y versionable.

9.4.1.3 Ramas de trabajo

Una vez que se tiene la rama estable y versionable (en este caso, el “trunk”), surge el problema de que en los desarrollos software es necesario guardar código fuente que no se ha podido verificar de manera adecuada, por ejemplo, porque finaliza la jornada de trabajo de un desarrollador sin que haya podido probar su código apropiadamente. Al no ser código completamente verificado, no se puede subir al “trunk” ya que se corre el riesgo de desestabilizarlo y por lo tanto se incumpliría su política. Por ello, surge la necesidad de tener una nueva rama, que se llamará rama de trabajo y que será compartida por todos los miembros del equipo.

En la Figura 52 se observan las dos ramas creadas hasta el momento, representando la nueva rama de trabajo (Equipo A), rama menos estable que el “trunk”. Esto es debido a que no se han pasado sobre ella pruebas de integración y puede darse el caso de que no sea lo suficientemente estable para ser versionada

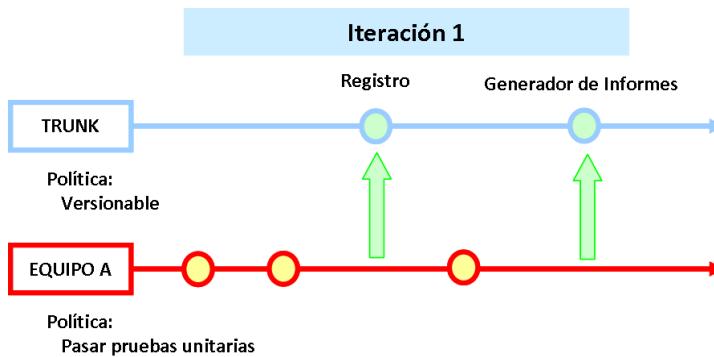


Figura 52. Rama de trabajo

9. Buenas prácticas de desarrollo para proyectos ágiles

Esta nueva rama de trabajo seguirá la siguiente política:

- El código compila y construye correctamente.
- Se pasan todas las pruebas unitarias de manera satisfactoria.

9.4.1.4 Publicar de la rama de trabajo al “trunk”

En algún momento de una iteración, el equipo de desarrollo da por realizada una historia de usuario. En este punto se debe “publicar” (copiar/subir) el código de la rama al “trunk”. Estas publicaciones son representadas en la Figura 52 con flechas, pudiendo observar que se publica la implementación del código de dos historias de usuario (registro y generador de informes).

Antes de realizar estas publicaciones, el equipo debe asegurarse de que se pasan todas las pruebas de integración y de que se corrigen todos los defectos que se encuentren en las mismas.

En la publicación de la rama al “trunk” pueden surgir problemas en el caso de que un mismo equipo esté desarrollando varias historias de usuario en paralelo y/o en el caso de que existan varios equipos de desarrollo en el proyecto. Ambos problemas se tratan a continuación.

9.4.1.5 Implementando historias de usuario en paralelo

Como se ha comentado, cada vez que se finaliza una historia de usuario, ésta se publica en el “trunk”, es decir, se realiza una copia completa de la rama de trabajo al “trunk”.

Cuando se están desarrollando varias historias en paralelo sobre una misma rama de trabajo, puede darse el caso que aunque una se haya dado por realizada, el resto aún no lo estén. Esto incumple la política de que el “trunk” debe ser siempre versionable ya que a la hora de publicar una historia, se estarían

9. Buenas prácticas de desarrollo para proyectos ágiles

copiando otras parcialmente completas. Para evitar esta problemática, Kniberg propone varias estrategias:

- No realizar demasiado desarrollo en paralelo. Intentar centrar al equipo en la misma historia de usuario.
- En el caso de que algún desarrollador comience a desarrollar una nueva historia de usuario antes de que se finalice la historia en curso, demorar la subida de este nuevo código hasta que se complete la actual o comenzar por aquellas partes del código que no la van a afectar.

Por lo general, los equipos tienden a sobrevalorar el hecho de desarrollar varias historias de usuario de manera simultánea. Es verdad que puede generar un sentimiento de rapidez en el trabajo. Sin embargo, puede ser solamente una ilusión ya que el tiempo ganado se tiende a perder al final de la iteración en el momento de integrar y probar todo el código.

9.4.1.6 Desarrollando con varios equipos de trabajo

Por el contrario, cuando se elige poseer varios equipos de trabajo suele ser con el objetivo de poder desarrollar diferentes historias de usuario de manera concurrente. En este caso, cada equipo posee una rama de trabajo como se muestra en la Figura 53:

9. Buenas prácticas de desarrollo para proyectos ágiles

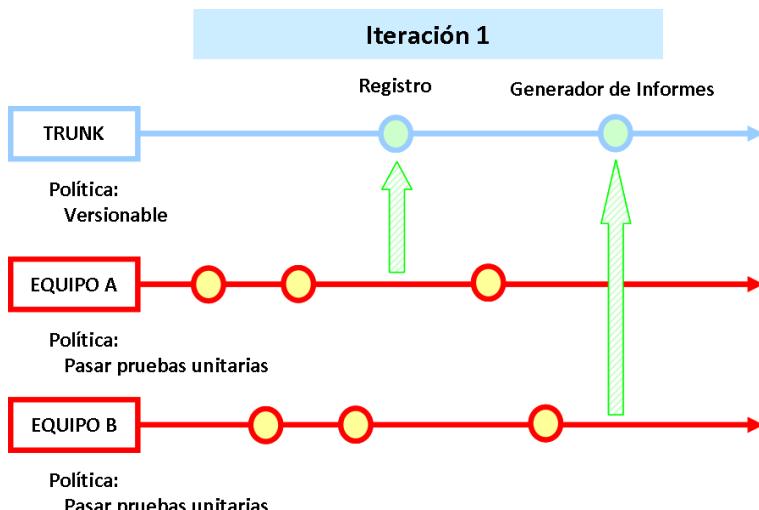


Figura 53. Varios equipos de desarrollo cada uno con su rama de trabajo

En este escenario, la problemática surge porque un equipo de desarrollo no controla lo que hace el otro. Esto puede provocar que uno de los equipos publique una nueva historia al “trunk” y de esta manera, la rama de trabajo del otro equipo (el que no ha realizado la publicación) se quede desincronizada y a la hora de intentar publicar su siguiente historia de usuario, obtengan conflictos en el código.

La solución que propone Kniberg a la problemática descrita en el párrafo anterior es descargar e integrar el código del “trunk” a cada rama de trabajo diariamente (Figura 54)³¹. Es aconsejable fijar la responsabilidad de esta actividad en un mismo miembro del equipo para que la realice diariamente y a primera hora, antes de que el resto del equipo comience a trabajar:

³¹ La acción de descargar e integrar el código de una rama a otra se conoce como “realizar un merge” o “mergear”, por su término en inglés.

9. Buenas prácticas de desarrollo para proyectos ágiles

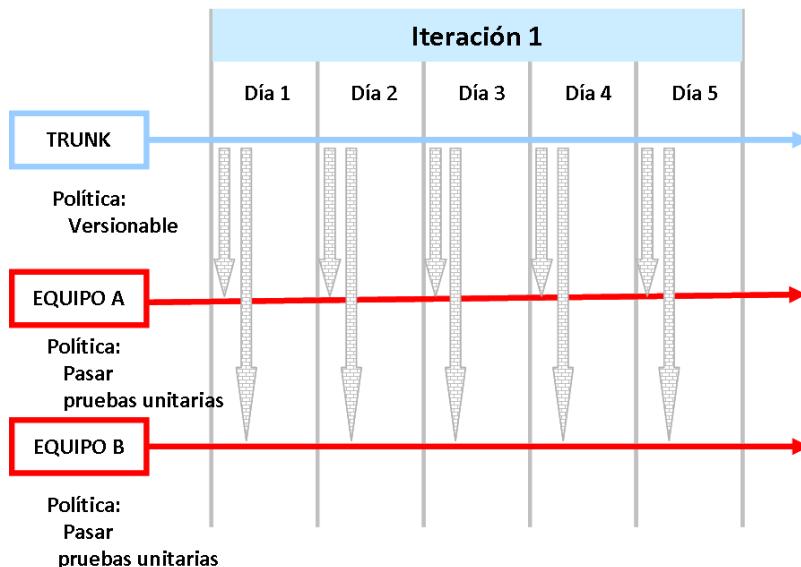


Figura 54. Integración diaria del “trunk” con cada rama de trabajo

Si en estas integraciones diarias se encuentra algún conflicto u otro problema, la resolución del mismo pasa a ser la principal prioridad del equipo, pudiendo pedir ayuda al otro equipo si fuese necesario.

A continuación se muestra una figura donde se describe una iteración completa: las flechas rayadas ascendentes muestran las publicaciones al “trunk”, las flechas rayadas descendentes las integraciones diarias sin que haya habido cambios y las flechas sólidas descendentes las integraciones diarias que sí descargan cambios (publicados por el otro equipo) en alguna de las ramas de trabajo (estos cambios son los que pueden provocar que aparezcan conflictos):

9. Buenas prácticas de desarrollo para proyectos ágiles

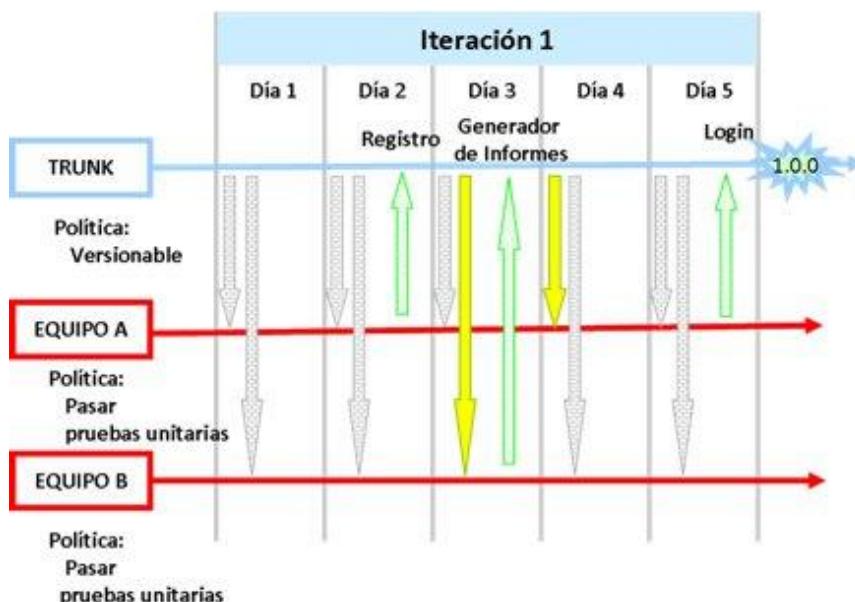


Figura 55. Iteración completa

9.4.1.7 Ramas de mantenimiento de versión

Por último, queda por explicar las ramas de mantenimiento de versión. Como se ha visto en la Figura 55, generalmente al finalizar una iteración se genera una nueva versión (en este caso, la 1.0.0).

En este momento se comenzaría la segunda iteración y puede darse el caso que mientras esté en curso, se abra un defecto grave en la versión generada al finalizar la iteración anterior. Lo normal sería solucionar el defecto sobre el "trunk" y generar una nueva versión (en este caso, la 1.0.1). El problema es que si entre las 2 versiones se publica alguna historia de usuario de la segunda iteración, irá incluida en la nueva versión y esto puede no ser lo deseado.

9. Buenas prácticas de desarrollo para proyectos ágiles

Si estamos en el caso de que no se desea etiquetar las nuevas versiones sobre el “trunk” con el riesgo de incluir funcionalidad de historias de usuario de otras iteraciones posteriores, se deben seguir los siguientes pasos:

- Se debe crear una nueva rama (rama de mantenimiento de versión, en este caso la rama VERSIÓN 1-0-X) a partir de la versión en la que se ha detectado el defecto grave (en este caso, la 1.0.0).
- Se debe corregir el error sobre esa rama y generar la nueva versión (en este caso, la 1.0.1).
- Una vez generada la nueva versión, se debe unir el contenido de esa rama con el “trunk”.

Este proceso se describe de manera gráfica en la siguiente figura:

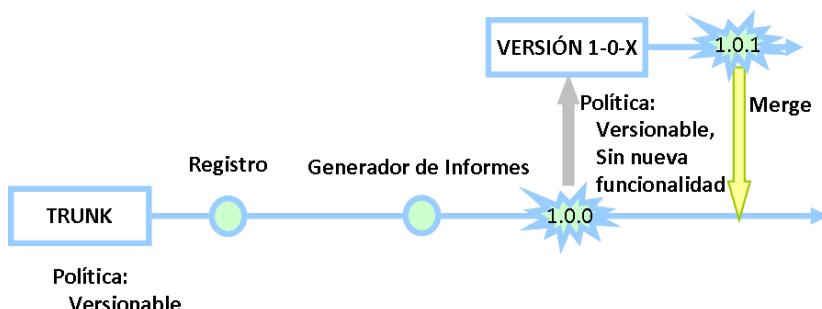


Figura 56. Rama de mantenimiento de versión

Con la definición de las ramas de mantenimiento de versión se pone fin a la explicación del modelo sugerido por Kniberg para proyectos ágiles. Aunque este modelo ha resultado exitoso en los proyectos en los que se ha probado y ha tenido generalmente una buena acogida por parte de los equipos de desarrollado hay que tener en cuenta que cualquier otro modelo usando otros patrones o políticas de GCS de las descritas en (Appleton, B. Berczuk, S. et al., 1998) también podría ser perfectamente válido.

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)



10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

"Cada década más o menos las metodologías de desarrollo entran en conflicto. La 1^a Década del siglo 21 no ha sido diferente, con los partidarios de los métodos ágiles en batalla con el mundo de CMMI"— Robert Austin

En este capítulo se tratará por un lado, la diferencia, y por otro, como se complementan las metodologías ágiles y los principales modelos de procesos, como CMMI-DEV (Capability Maturity Model Integration for Development) (CMMI Product Team, 2010) o ISO/IEC 12207 (ISO, 2008).

De hecho son cada vez más las empresas que han combinado ambos enfoques y muchas las que encuentran la necesidad de hacerlo, aprovechando las ventajas de las metodologías ágiles y de modelos como CMMI-DEV o ISO 15504.

10.1 Modelos de procesos (CMMI e ISO/IEC 12207) y las prácticas ágiles

Pueden ser muchas las razones de por qué se perciben los métodos ágiles y CMMI-DEV como elementos opuestos. Puede que las primeras implantaciones de CMMI-DEV fueran en organizaciones de gran tamaño, y las primeras implantaciones ágiles fueran realizadas en empresas pequeñas, pequeños equipos o con requisitos volátiles.

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

Tampoco se debe olvidar la mala aplicación, interpretación y evaluación de CMMI-DEV en empresas, donde los implantadores y/o evaluadores olvidan el negocio de la empresa, y la ingeniería del software que más se adapta a la organización, generando cantidades inútiles de sobredocumentación para justificar la implantación del modelo. Quizás también por la terminología, “institucionalizar”, “repetible”, etc., frente a “integración continua”, “refactoring”, “iterativo e incremental”, etc. O quizás porque CMMI-DEV o ISO/IEC 12207 se vean antiguos y los métodos ágiles algo moderno; aunque prácticas clave en los métodos ágiles, como el ciclo de vida iterativo e incremental (de los años 60), o la refactorización (de los año 90), son anteriores a la aparición de CMMI-DEV o ISO/IEC 12207.

Como indica Austin (Austin, 2007), entre las metodologías de desarrollo aparecen conflictos cada cierto tiempo. La batalla entre las metodologías ágiles y los modelos de proceso como CMMI-DEV o ISO/IEC 12207 persiste desde hace ya unos años. Aún así, el uso de prácticas ágiles se utiliza cada vez más junto con los modelos de procesos. Incluso destacados autores del mundo ágil, como los padres de Scrum, sugieren que existe una gran sinergia al utilizarlas (J. Sutherland & Johnson, 2010).

CMMI-DEV (CMMI Product Team, 2010) es actualmente estándar de facto para la mejora de procesos y la determinación de los niveles de madurez en las organizaciones. CMMI-DEV y las metodologías ágiles comparten características, aunque se encuentren en diferentes niveles de abstracción. No están necesariamente en competición, y, de hecho, pueden ser complementarias. A continuación se listan las características más relevantes de los modelos de proceso (como CMMI-DEV o ISO/IEC 12207) respecto de las buenas prácticas ágiles:

- Un modelo de procesos se centra en el qué se espera encontrar en una organización, mientras que metodologías y métodos ágiles se centran en el cómo elaborar productos del ciclo de vida del software. En algu-

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

nos puntos el modelo de procesos puede mostrar productos de trabajo típicos, pero son recomendaciones o ejemplos, no obligaciones.

- Un modelo de procesos no establece orden en la ejecución de los procesos, ni determina un ciclo de vida, son las metodologías quienes determinan este punto, recomendando por ejemplo el ciclo de vida iterativo e incremental.
- Un modelo de procesos muestra áreas de proceso, no procesos en sí. Muestra tipologías de proceso, que luego en cada organización pueden instanciarse de diferente manera, y pueden existir numerosas correspondencias entre “áreas de proceso” y “procesos” de la organización, etc. De hecho las evaluaciones de un modelo de procesos tienen (o deberían tener) el objetivo de interpretar el modelo en la organización concreta, y no el buscar una relación uno a uno entre áreas de proceso y procesos de la organización.

Llevar los procesos al extremo, no aceptando prácticas ágiles, es tan perjudicial como llevar prácticas ágiles al extremo opuesto, sin incorporar características consideradas “no ágiles”. Y esto produce resultados insatisfactorios en los proyectos, en las metodologías y especialmente en las personas.

Esta es una de las barreras más arraigadas actualmente en la mejora de procesos y el desarrollo software en el mundo ágil. Ya sea con la frase “los procesos implican ciclos de vida en cascada” o sustituyendo la palabra “procesos” por algún modelo de procesos concreto, principalmente CMMI-DEV o ISO 12207: “No implantamos CMMI-DEV porque implica ciclo de vida en cascada”, “ISO 12207 no funciona porque trabajamos de manera ágil, iterativa”.

En las especificaciones de los modelos de mejora y evaluación de procesos actualmente más conocidos no existe ninguna mención explícita que indique secuencia, fases o comportamientos estancos entre actividades. El ciclo de vida en cascada es mencionado solamente una vez, junto el ciclo de vida iterativo e

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

incremental en CMMI-DEV. Asimismo, el modelo de procesos de referencia ISO/IEC 12207 (utilizado junto con la norma ISO/IEC 15504 en las evaluaciones de la calidad del proceso de desarrollo software) tampoco hace mención a ningún aspecto relacionado con un ciclo de vida en cascada. De hecho, indica explícitamente que el orden de los procesos indicados en la norma no establece ningún tipo de secuencia.

10.2 Relación entre Scrum y las áreas de proceso CMMI-DEV

Como se ha visto en capítulos anteriores, Scrum es fundamentalmente una metodología para la gestión de proyectos. En este sentido, CMMI-DEV y las prácticas ágiles están en diferentes niveles de abstracción. CMMI se enfoca en lo que debe hacer la organización (incluyendo los proyectos software) y Scrum en cómo tiene que hacerse una gestión de proyectos ágiles. Como indicaba Glazer y otros autores, (Glazer et al., 2008) Scrum brinda un “how-to” de la gestión del proyecto software. Los métodos ágiles, como Scrum, contienen los pasos de cómo trabajar en entornos particulares (Paultk, 2001)(Glazer et al., 2008).

En ese sentido, pueden encontrarse diferentes estudios que intentan unir los modelos de desarrollo de procesos y las prácticas ágiles, como por ejemplo (Glazer et al., 2008) o (J. Sutherland & Johnson, 2010). Estos autores comentan que tanto CMMI como los métodos ágiles pueden trabajar juntos. En el caso de Scrum, (J. Sutherland et al., 2008) y (Jakobsen & Johnson, 2008) discuten sobre la relación entre el nivel 5 de madurez de CMMI-DEV y Scrum. Estudios similares, como (Marcal et al., 2007)(Jakobsen & Sutherland, 2009)(Potter & Sakry, 2011) identifican equivalencias entre ambos elementos.

A continuación se analizarán las buenas prácticas de las áreas de proceso correspondientes a la gestión de proyectos en CMMI-DEV y como pueden ser cumplidas con las prácticas ágiles de Scrum. Las áreas de procesos de CMMI-

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

DEV implicadas son las siguientes³²:

- Integrated Project Management (IPM)
- Project Monitoring and Control (PMC)
- Project Planning (PP)
- Quantitative Project Management (QPM)
- Requirements Management (REQM)
- Risk Management (RSKM)
- Supplier Agreement Management (SAM)

Para que un área de proceso sea correctamente implementada, deben alcanzarse las metas específicas definidas para esa área de proceso, que a su vez se consiguen mediante la implementación de las prácticas específicas de cada meta. Cuando se va a evaluar la implementación de las prácticas se comienza desde abajo hasta arriba, esto es, evaluando en primer lugar el cumplimiento de las prácticas específicas de cada meta específica, posteriormente las metas específicas y luego las áreas de proceso.

Las relaciones entre las metas específicas de estas áreas de proceso y las prácticas de Scrum se ven en la Tabla 12. La primera columna contiene las metas específicas (o SG, del término inglés *specific goal*) de CMMI-DEV y la segunda contiene el grado de apoyo por parte de Scrum. Estos resultados provienen de un estudio realizado en el año 2012 y publicado por Javier Garzás y Mark Paulk en (Garzás & Paulk, 2012). Así mismo pueden encontrarse trabajos similares relacionados con la norma ISO/IEC 12207 (Garzás et al., 2011).

³² Los nombres de las áreas de proceso fueron tomados de la versión original, en inglés, del modelo CMMI-DEV v1.3.

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

Área de proceso	Apoyo por parte de Scrum
Requirements Management (REQM)	
SG 1 Manage Requirements	++
Project Planning (PP)	
SG 1 Establish Estimates	++
SG 2 Develop a Project Plan	+
SG 3 Obtain Commitment to the Plan	++
Project Monitoring and Control (PMC)	
SG 1 Monitor the Project Against the Plan	+
SG 2 Manage Corrective Action to Closure	++
Integrated Project Management (IPM)	
SG 1 Use the Project's Defined Process	O
SG 2 Coordinate and Collaborate with Relevant Stakeholders	++
Quantitative Project Management (QPM)	
SG 1 Prepare for Quantitative Management	O
SG 2 Quantitatively Manage the Project	O
Risk Management (RSKM)	

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

SG 1 Prepare for Risk Management	<input type="radio"/>
SG 2 Identify and Analyze Risks	<input type="radio"/>
SG 3 Mitigate Risks	<input type="radio"/>
Supplier Agreement Management (SAM)	
SG 1 Establish Supplier Agreements	<input type="radio"/>
SG 2 Satisfy Supplier Agreements	<input type="radio"/>

Tabla 12. Metas específicas cubiertas con Scrum (Garzás & Paulk, 2012).
+ Parcialmente; + + Totalmente; O No cubiertas

Scrum ayuda a la gestión de requisitos mediante el uso de las historias de usuario, el Product Backlog y el Sprint Backlog. Para entender los requerimientos, el Product Owner junto con el equipo revisa el Product Backlog. La reunión de planificación del Sprint, las reuniones diarias y la revisión final del Sprint son utilizadas para encontrar impedimentos e inconsistencias. Para administrar los requerimientos de forma efectiva, las buenas prácticas como los “cambios de historia” o el “impacto” son muy recomendadas. De acuerdo con (K. Schwaber & Sutherland, 2010), el Product Backlog es el origen de los requerimientos con los que desarrollar y modificar el producto software, incluyendo los errores que serán corregidos en futuras versiones. En CMMI-DEV se especifica de manera explícita la verificación y validación en lugar de las pruebas de software, pero las pruebas no son una práctica explícita. Aún así, las pruebas de regresión y la integración continua son partes implícitas de las implementaciones ágiles (por ejemplo, en eXtreme Programming, una de las buenas prácticas es la integración continua y las pruebas). Este tipo de pruebas ayudan a mantener la trazabilidad bidireccional de los requerimientos y ayuda a la alineación entre los productos de trabajo y los requerimientos.

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

Scrum también ayuda a la planificación de proyectos mediante el uso de “puntos de historia” para estimar las diferentes iteraciones. Las reuniones diarias, la reunión de planificación y la revisión del Sprint son utilizadas para revisar el plan del proyecto, entender los avances y mejorar estimando los recursos actuales y futuros. Pero para realizar una estimación efectiva, elementos como la gestión de riesgos o la gestión de datos son recomendados. Estos aspectos no están explícitamente cubiertos por Scrum.

La monitorización y control del proyecto se realiza en Scrum a partir de los gráficos

BurnDown y a partir de las reuniones (en este sentido, también XP y otras prácticas ágiles hacen un amplio uso de las reuniones). Este gráfico indica el esfuerzo necesario para terminar con la iteración, así como los puntos de historia ya completados. Las reuniones de revisión del Sprint y las reuniones diarias permiten la monitorización del proyecto y el registro de acciones, pero al igual que con la planificación de proyectos, no se realiza correctamente la gestión de los riesgos.

Respecto a la gestión integrada de proyectos, Scrum utiliza roles y reuniones para conseguirlo. Aunque en Scrum sólo existen tres roles principales (“Product Owner”, el equipo de desarrollo y el “Scrum Master”) muchos proyectos terminan necesitando algunos roles más. A mayor nivel de detalle las relaciones potenciales entre las metas específicas (SG), prácticas específicas (o SP, del término inglés *specific practices*) y Scrum se listan en la Tabla 13.

Área de proceso	Apoyo por parte de Scrum
Requirements Management (REQM)	
SG 1 Manage Requirements	+
SP 1.1 Understand Requirements	++
SP 1.2 Obtain Commitment to Requirements	++

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

SP 1.3 Manage Requirements Changes	++
SP 1.4 Maintain Bidirectional Traceability of Requirements	+
SP 1.5 Ensure Alignment Between Project Work and Requirements	+
Project Planning (PP)	
SG 1 Establish Estimates	++
SP 1.1 Estimate the Scope of the Project	++
SP 1.2 Establish Estimates of Work Product and Task Attributes	++
SP 1.3 Define Project Lifecycle Phases	++
SP 1.4 Estimate Effort and Cost	++
SG 2 Develop a Project Plan	+
SP 2.1 Establish the Budget and Schedule	++
SP 2.2 Identify Project Risks	O
SP 2.3 Plan Data Management	O
SP 2.4 Plan the Project's Resources	++
SP 2.5 Plan Needed Knowledge and Skills	O
SP 2.6 Plan Stakeholder Involvement	++
SP 2.7 Establish the Project Plan	+
SG 3 Obtain Commitment to the Plan	++
SP 3.1 Review Plans That Affect the Project	++
SP 3.2 Reconcile Work and Resource Levels	++
SP 3.3 Obtain Plan Commitment	++
Project Monitoring and Control (PMC)	
SG 1 Monitor the Project Against the Plan	+
SP 1.1 Monitor Project Planning Param-	++

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

ters	
SP 1.2 Monitor Commitments	++
SP 1.3 Monitor Project Risks	O
SP 1.4 Monitor Data Management	O
SP 1.5 Monitor Stakeholder Involvement	+
SP 1.6 Conduct Progress Reviews	++
SP 1.7 Conduct Milestone Reviews	+
SG 2 Manage Corrective Action to Closure	++
SP 2.1 Analyze Issues	++
SP 2.2 Take Corrective Action	++
SP 2.3 Manage Corrective Actions	++
Integrated Project Management (IPM)	
SG 1 Use the Project's Defined Process	O
SG 2 Coordinate and Collaborate with Relevant Stakeholders	+
SP 2.1 Manage Stakeholder Involvement	++
SP 2.2 Manage Dependencies	+
SP 2.3 Resolve Coordination Issues	+
Quantitative Project Management (QPM)	
SG 1 Prepare for Quantitative Management	O
SG 2 Quantitatively Manage the Project	O
Risk Management (RSKM)	
SG 1 Prepare for Risk Management	O
SG 2 Identify and Analyze Risks	O
SG 3 Mitigate Risks	O
Supplier Agreement Management (SAM)	
SG 1 Establish Supplier Agreements	O
SG 2 Satisfy Supplier Agreements	O

Tabla 13. Prácticas y metas específicas cubiertas con Scrum (Garzás & Paulk, 2012). + + Totalmente; + Parcialmente; O No cubiertas

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

En la Figura 57 se puede apreciar de manera gráfica el grado de cobertura de cada meta específica con las prácticas de Scrum. Se han tenido en cuenta las áreas de proceso que se encuentran amplia o parcialmente cubiertas, dejando fuera las áreas de proceso Quantitative Project Management (QPM), Risk Management (RSKM) y Supplier Agreement Management (SAM) que no se cubren.

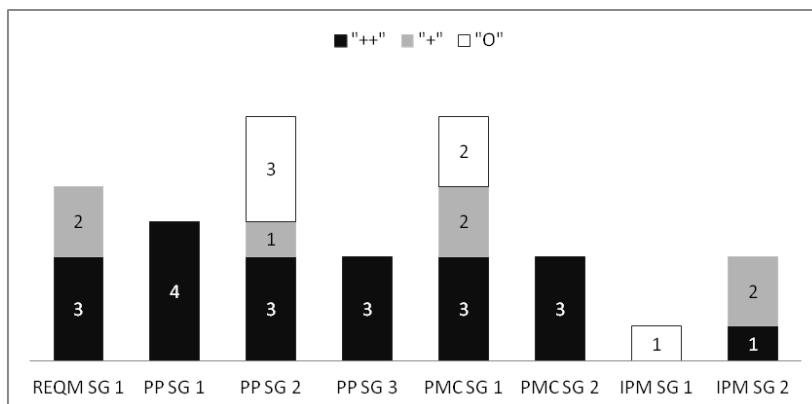


Figura 57. Grado de cobertura de las metas específicas cubiertas con Scrum (Garzás & Paulk, 2012). + + Totalmente; + Parcialmente; O No cubiertas

10.3 Conclusiones

Hay que tener en cuenta que cubrir las buenas prácticas de los modelos de proceso no es una tarea simple y sí que hay aspectos no cubiertos o partes más complejas de abordar. Teniendo en cuenta estas comparaciones se resumen los aspectos principales respecto a la relación entre Scrum y las áreas de proceso de gestión de proyectos de CMMI-DEV:

- **Ausencia de evidencia documental.** Algunas de las prácticas de Scrum no dejan una evidencia documental que demuestre la implementación de las buenas prácticas de CMMI-DEV. Por ejemplo, un tablero en el cual las tareas o las historias de usuario son capturadas en

10. Metodologías ágiles y modelos de procesos software (CMMI e ISO)

notas es una evidencia de planificación y registro de progresos, pero este tipo de evidencias no son fáciles de demostrar en una evaluación formal de CMMI-DEV. Algunas empresas de software seguramente deberán agregar algún producto de trabajo para ser evaluadas correctamente. Pero esto puede ser considerado como una “evidencia artificial” y el equipo puede invertir demasiado tiempo en recolectarla.

- **La gestión de riesgos.** De acuerdo con el glosario de CMMI, la planificación del proyecto incluye la estimación de los productos de trabajo y tareas, la determinación de recursos, la planificación y la gestión de riesgos. Muchas de estas actividades se encuentran cubiertas por Scrum, con la excepción de la gestión de riesgos. Aún así, en las reuniones diarias se tienen en cuenta los impedimentos del equipo de desarrollo.
- **Los “stakeholders” y el “Product Owner”.** De acuerdo con el glosario de CMMI, un “stakeholder” es un grupo o individuo que es afectado por el resultado del proyecto. Por lo tanto incluye a proveedores, clientes y usuarios finales entre otros. En la guía 2011 de Scrum el “Product Owner” es el responsable de maximizar el valor del producto y el trabajo del equipo de desarrollo. El “Product Owner” es una persona, no un grupo aunque lo represente. Por lo tanto, el “Product Owner” debe recolectar y representar las necesidades de todos los implicados.

Las prácticas de Scrum deben ser consideradas como buenas prácticas para la gestión de proyectos en la mayoría de las organizaciones, pudiendo ser un soporte excelente para las prácticas de CMMI-DEV. Incluso si las prácticas de Scrum no cumplen todas las buenas prácticas de CMMI-DEV, existen beneficios comprobados.



11. Contratos ágiles

"Los practicantes ágiles muchas veces tenían problemas para cumplir los principios ágiles" – Rashina Hoda

De manera general, existen dos paradigmas para establecer un contrato de desarrollo software: los contratos fijos y los contratos de pago por tiempo de proyecto. Y sobre estos paradigmas, encontraremos diferentes maneras de manejar el tiempo, alcance, precio y la calidad, y diferentes repartos en el riesgo que cada parte (cliente y proveedor) asumen en el proyecto.

En el año 2009 Rashina Hoda comentaba el problema de cumplir las prácticas ágiles (Hoda et al., 2009), especialmente por los contratos y su negociación. Entre los principales comentarios enumerados por Rashina Hoda e indicados por los profesionales que intentaban negociar contratos ágiles, se encontraban algunos como los siguientes:

- Muchas veces los clientes imponen limitaciones. Buscan satisfacer sus expectativas y requerimientos, esperando fecha fija de cierre, precio fijo y alcance fijo.
- La mayoría de los contratos que hacen los clientes desde hace años han sido contratos de precio fijo, también llamados contratos cerrados o llave en mano. En este tipo de contratos la empresa de desarrollo asume la mayoría de los riesgos.

11. Contratos ágiles

11.1 Los principales tipos de contratos en entornos ágiles

Para solucionar los anteriores problemas, en los entornos ágiles podemos encontrar diferentes tipos de contratos. Alistair Cockburn recopila un gran número de ellos en el “Agile Contracts” (A. Cockburn, 2006). De esta y otras fuentes como (Beamount, 2008) o (J. Sutherland, 2008), se pueden concluir tres grandes grupos de contratos: pago por tiempo o por iteración, pago por punto función o punto historia y contrato fijo.

Estos tres grupos son divisiones teóricas. A la hora de aplicar estos conceptos en un proyecto, la mayoría de los contratos tienen tanto componentes ágiles como componentes de contrato fijo. A continuación se comentan las principales características de cada grupo.

11.1.1 Pago por tiempo o por iteración

La manera más natural de formalizar un contrato para un proyecto ágil es la de pago por tiempo de trabajo (o por número de Sprint), también conocido como “por servicio” o “por bolsa de horas”. En su forma más básica, el proyecto duraría hasta que la versión final del producto software esté terminada. Y los pagos se harían normalmente por el tiempo que ha durado el proyecto, por iteración o por número de Sprint. Si el tiempo estimado de proyecto es muy largo también se puede facturar al finalizar cada Sprint.

El problema de esta modalidad es que la mayoría del riesgo lo tiene el cliente. El proveedor podría relajarse, dando lugar a más iteraciones de las necesarias y mayores costes.

Para ello, en este tipo de contratos se suelen introducir cláusulas que reducen el riesgo del cliente. Lo normal es cerrar de antemano un tiempo máximo de proyecto, dando la opción al cliente de cancelar el proyecto o renegociar una continuación. O de manera similar, introducir hitos de chequeo cada cierto

tiempo. Por ejemplo, en un proyecto se podría determinar que cada cuatro meses el cliente podrá decidir si continúa o cancela el proyecto.

11.1.2 *Pago por punto función o punto historia*

Este es uno de los esquemas más recomendados. En algunos sectores del desarrollo software, como por ejemplo la banca, se ha introducido con fuerza un modelo en el que el proveedor factura por la cantidad de funcionalidad desarrollada. Y para cuantificar la funcionalidad entregada se mide con diferentes técnicas, como por ejemplo, la técnica de puntos función para medir el tamaño de la funcionalidad. Cabe destacar que el cliente paga por puntos función entregados, no por los que se estimaron, por lo que suele ser un esquema bastante eficiente para ambas partes. Asimismo se puede replicar este modelo cambiando puntos función por puntos de historia de usuario o similar.

11.1.3 *El contrato cerrado o fijo en proyectos ágiles*

En este tipo de contratos el cliente fija el alcance, requisitos, precio y el tiempo del proyecto. Este tipo de contrato debiera utilizarse sólo cuando los requisitos son altamente estables (lo cual disminuye las ventajas que tienen los desarrollos ágiles respecto a requisitos volátiles). Normalmente en este tipo de contrato los mayores riesgos corren por parte del proveedor.

Uno de los problemas cuando se establece un contrato cerrado es que de las variables que se pueden manejar (tiempo, alcance, precio y calidad) normalmente sólo se hace mención a tiempo, alcance y precio, olvidando la calidad. Esto suele derivar en que la calidad es mínima, en pro de entregar en tiempo algo que funcione.

No hay que olvidar el manifiesto ágil en el que uno de sus cuatro valores indica que la colaboración con el cliente está por encima de la negociación contractual. Y este tipo de contrato se suele dar cuando las partes, cliente y proveedor,

11. Contratos ágiles

confían poco la una en la otra (haciendo más evidentes los intereses enfrentados enumerados en la Tabla 14), lo que vulnera el valor ágil de la interacción con el cliente.

Cliente	Proveedor
Interpreta los requisitos de la manera más amplia posible para aumentar la cantidad de funcionalidad obtenida con respecto al dinero invertido.	Interpreta los requisitos de una manera simplista, para reducir el gasto en recursos.
Quiere que el trabajo se realice lo más rápido posible.	Quiere terminar el trabajo en la fecha prevista.
Quiere una calidad superlativa.	Quiere invertir en la calidad justa, de acuerdo con lo que ha pagado el cliente.
El agotamiento de los desarrolladores no es problema del cliente, a menos que amenace la entrega del producto software.	Quiere que los miembros del equipo tengan éxito en el proyecto actual y conservarlos frescos para el siguiente proyecto.

Tabla 14. Intereses enfrentados que aparecen en un contrato cerrado (K. Beck & Cleal, 1999)

Una alternativa para adaptar este tipo de contrato a entornos ágiles es fijar el precio pero no el alcance. O fijar el alcance y dejar que el precio sea variable. Jeff Sutherland describía estas dos aproximaciones en una ponencia realizada en

2008 titulada “Money for nothing and Change for Free” (J. Sutherland, 2008). El “Money for nothing” describía un contrato de precio fijo - alcance variable que permitía al cliente darlo por terminado en cualquier momento (más concretamente al final de cualquier Sprint). Esto normalmente puede suceder cuando el cliente observa que el coste de continuar es mayor que el valor que va a recibir. Cuando ocurre esta situación, cuando se para el proyecto antes del tiempo previsto, el cliente debe pagar a la empresa de desarrollo un 20% de lo que se facturaría por los futuros Sprint que ya no se van a desarrollar. El “Change for free” es similar pero describe la situación de precio variable – alcance fijo.

La realidad dice que la mayoría de los contratos son de precio fijo y alcance fijo, y este es un tema cultural que aunque sería deseable, es complejo de que cambie.

11.2 Los riesgos que asume cada parte en un contrato

Un punto clave a la hora de realizar un contrato es entender los riesgos que cada parte quiere minimizar: el cliente necesita minimizar el riesgo de satisfacción de sus necesidades, ya que el software suele contratarse para satisfacer una necesidad relativa al negocio u objetivo de mercado. Y el proveedor debe minimizar el riesgo de la entrega del software (y de su facturación). Como se ve en la Figura 58, en el contrato cerrado el riesgo es asumido generalmente por el proveedor mientras que en el pago por tiempo e iteración, el riesgo pasará a estar del lado del cliente. En ningún caso podrá existir la ausencia del riesgo, por lo tanto una de las opciones es compartir los riesgos.

El desafío es encontrar una manera de trabajar que permita el cambio de requisitos durante el ciclo de vida sin largos procesos de negociación, gestión del cambio y nuevas estimaciones de costes.

11. Contratos ágiles

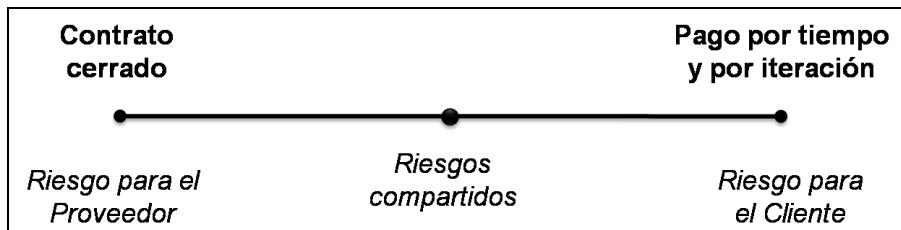


Figura 58. ¿Quién asume el riesgo en un proyecto software? (Beamount, 2008)

En relación con los riesgos toman un papel fundamental las estimaciones. El nivel de precisión de la estimación es distinto a medida que avanza el proyecto. Al principio del proyecto, cuando sólo se tienen requisitos, el error con el que se trabaja es mayor. Por tanto, el formato de contrato de “pago por tiempo e iteración” disminuirá en mayor medida la incertidumbre al realizar las estimaciones que el “contrato cerrado”.

11.3 Otros aspectos a tener en cuenta en un contrato

Aunque en este capítulo hemos visto sólo las líneas generales, a la hora de fijar un contrato hay otros factores a considerar. Cabe resaltar que los contratos son un conjunto de reglas para llevar a cabo durante el proyecto de la manera más óptima y satisfactoria para ambas partes. De acuerdo con Peter Stevens, los principales puntos que siempre deberían aparecer en todo contrato son (Stevens, 2009):

1. Objetivos del proyecto y de la cooperación entre las organizaciones.
2. Un resumen de la estructura del proyecto. Por ejemplo, metodología que se va a usar, roles clave, etc.
3. Personal clave: quién es responsable a nivel operacional y jerárquico, y qué se necesita de estas personas.
4. Pagos y facturación, incluyendo las cláusulas de bonificaciones y penalizaciones.

11. Contratos ágiles

5. Terminación normal y temprana del contrato.
6. "Detalles legales". Dependiendo de las leyes locales y costumbres legales, es posible que tenga que limitarse la responsabilidad civil, especificar la ganancia, o incluir otros textos que se necesiten legalmente. Aquí es necesario conocer las prácticas legales del país donde se firma el contrato.
7. Garantía y tiempo de duración.
8. Propiedad intelectual.

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

12. Los proyectos ágiles y el desarrollo global del software



12. Los proyectos ágiles y el desarrollo global del software

“El desarrollo global de software se está convirtiendo rápidamente en la norma de las empresas de base tecnológica” – James Herbsleb

El desarrollo global de software (global software development o GSD) es una práctica que consiste en desarrollar un producto software a través de interacciones entre personas, organizaciones y tecnologías a lo largo de diferentes naciones con diferentes ambientes, lenguajes y estilos de trabajo (Herbsleb & Mockus, 2003). Esta definición es quizás demasiado estricta, y últimamente incluso se aplica el término GSD cuando el desarrollo es realizado en varias provincias del mismo país. Sea como fuere, la utilización de esta práctica implica la necesidad de que exista una comunicación, una coordinación y un control de los diferentes grupos de trabajo, de manera que puedan evitar los problemas que puede acarrear esta forma de desarrollar software.

El GSD es una **manera novedosa de llevar a cabo los desarrollos software**. Frente al modelo tradicional en el que los diferentes implicados en el desarrollo se reúnen con cierta frecuencia en un lugar físico, en un GSD los distintos implicados se encuentran en diversos sitios remotos, conformando un “equipo virtual” que utiliza las diferentes tecnologías existentes para interactuar y llevar a cabo las diferentes tareas.

12. Los proyectos ágiles y el desarrollo global del software

Existen diferentes tipos de GSD en función de la tipología de la organización:

- **GSD Intra-organizacional:** los desarrollos son realizados por la misma compañía, pero en sedes alejadas geográficamente.
- **GSD Inter-organizacional:** en este caso los desarrollos son realizados por diferentes organizaciones alejadas geográficamente.
- **GSD no organizacional:** se corresponde con desarrollos de software libre, en los que una comunidad de usuarios desarrolla un software desde diferentes posiciones geográficas.

Típicamente se han asociado una serie de beneficios al GSD, similares a la externalización del tipo “offshoring”³³:

- Minimizar los costes de desarrollo.
- Formar un equipo más cualificado.
- Aprovechar la diferencia horaria entre los diferentes lugares de desarrollo.

Sin embargo, existen otras ventajas que no se observan a simple vista, como mejorar la innovación gracias a la presencia de personal de diferentes países y culturas, compartir las mejores prácticas de diferentes organizaciones, mejorar la modularización de las tareas, incrementar la autonomía de los equipos o mejorar la documentación y la claridad de los procesos.

³³ Los términos externalización y offshoring tienden a confundirse. El offshoring es un tipo de externalización, en el cual el proveedor está en un país diferente al cliente, teniendo como uno de los principales objetivos el ahorro en costes.

12. Los proyectos ágiles y el desarrollo global del software

BENEFICIOS ORGANIZACIONALES	Ahorro de costes	Puede accederse a personal con un coste más bajo.
	Acceder a personal con diferentes habilidades	Permite acceder a personal de diferentes países y culturas, con diferentes habilidades.
	Reducir el tiempo de mercado	Posibilita realizar la estrategia de 24 horas de desarrollo, lo que permite mejorar la productividad, pudiendo llegar así antes al mercado.
	Proximidad al mercado y al cliente	Tener una sede en otro país permite conocer mejor el mercado local y a los potenciales clientes.
	Innovación y compartir mejores prácticas	Al tener personal de diferentes países y culturas, se incrementa la innovación y se comparten las mejores prácticas.
	Distribución de recursos	Permite la reasignación de recursos y los traspasos de actividades entre organizaciones para maximizar la productividad.
BENEFICIOS DEL EQUIPO	Mejorar la modularización de tareas	Al tener que separar las tareas de cada organización, se produce una mejora en la modularización de las mismas.
	Incrementar la autonomía del equipo	Permite que se mantenga un pequeño grado de autonomía en las culturas de trabajo de los equipos.
BENEFICIOS DE LOS PROCESOS	Registro formal de las comunicaciones	Al utilizarse comunicaciones como email, VoIP, etc, permite obtener trazabilidad en las comunicaciones.
	Mejora de la documentación	La necesidad de documentos claros hace que la documentación de la organización mejore.
	Procesos claramente definidos	Los procesos deben ser claros y entendidos por todas las clases, por lo que han de estar claramente definidos.

Tabla 15. Beneficios del GSD

12.1 Metodologías ágiles y GSD

Aparentemente, muchas de las prácticas y principios propuestos por GSD parecen chocar directamente con las prácticas y principios propuestos por las metodologías ágiles. Las largas distancias físicas, culturales y horarias impactan principalmente en la comunicación entre los miembros del equipo, y en las diferentes reuniones necesarias que son pieza fundamental en las metodologías ágiles. Muchas metodologías ágiles propugnan la interacción cara a cara, cosa compleja cuando hay distancia geográfica y husos horarios.

Sin embargo, estos problemas pueden minimizarse sobre todo con el uso de herramientas de trabajo colaborativo, herramientas de conferencia, herramientas

12. Los proyectos ágiles y el desarrollo global del software

tas de comunicación electrónica, etc. De hecho, como se observa en la Figura 59, según un estudio de Samireh Jalaji y Claes Wohlin (Jalali & Wohlin, 2010) generalmente la implantación de GSD en proyectos ágiles acaba de manera exitosa.

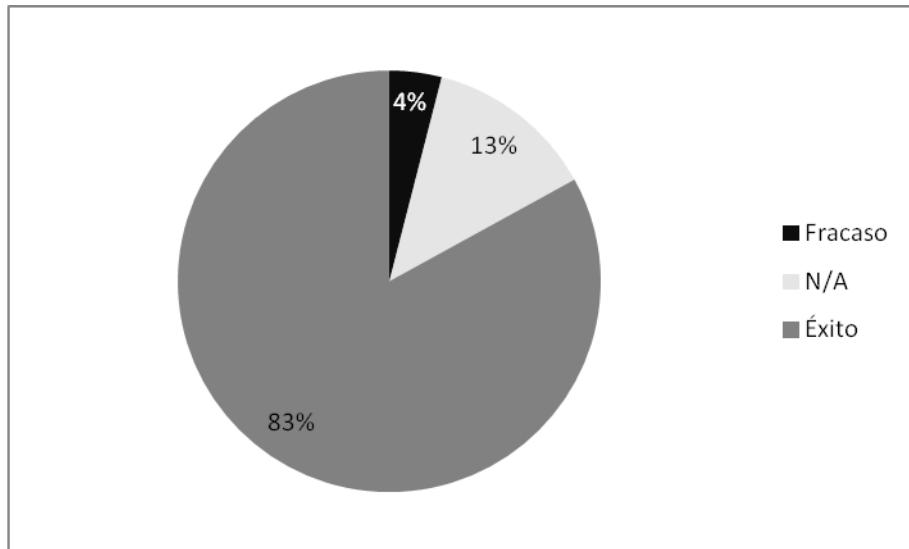


Figura 59. Datos de la implantación de GSD y prácticas ágiles, extraídos de (Jalali & Wohlin, 2010)

12.2 Problemas y soluciones al implementar el desarrollo global de software con prácticas ágiles

Los principales retos y soluciones de implantar prácticas ágiles y GSD se encuentran en los siguientes puntos:

- **El reto de la realización de comunicaciones síncronas:** Se proponen estrategias como sincronizar las horas de trabajo para las comunicaciones entre equipos y la creación de equipos locales que se reúnen físicamente.

12. Los proyectos ágiles y el desarrollo global del software

- **Las dificultades en la colaboración:** Para solucionar este problema, se propone la realización de visitas entre los principales miembros de los equipos o la realización de reuniones no oficiales entre los equipos.
- **El problema del soporte de herramientas:** Los equipos distribuidos utilizan herramientas colaborativas como blogs, wikis, pizarras compartidas, marcadores sociales, bases de conocimiento, bases de datos de experiencia, repositorios, etc. Por ejemplo, para facilitar el pair programming (programación por pares) en la que se requiere que dos desarrolladores participen juntos durante el desarrollo en un sitio de trabajo, una posibilidad podría ser recurrir a herramientas como Terminal Server de Microsoft (o cualquier otra similar) que proporcionan servicios de escritorio remoto, de manera que podrían compartir un mismo equipo físico donde realizarían el desarrollo, trabajando uno o los dos desarrolladores de manera remota.
- **El desafío de la gestión de los equipos:** en las prácticas ágiles es común contar con equipos pequeños que tienden a autogestionarse. En GSD es necesario coordinar estos equipos para que puedan trabajar en conjunto. Este problema puede resolverse utilizando diferentes soluciones como:
 - **Equipos aislados:** los equipos son independientes unos de otros, apenas existiendo comunicación.
 - **Equipos locales-globales:** en este caso se forman equipos locales con los miembros de cada equipo de trabajo, y los responsables de cada equipo forman un equipo global (en Scrum, una reunión de este equipo se conoce como “Scrum de Scrums”).

12. Los proyectos ágiles y el desarrollo global del software

- **Equipos completamente distribuidos:** los equipos están formados por miembros de diferentes localizaciones de **manera distribuida**. Por ello, es común que se realicen adaptaciones y que se incluyan solamente algunas prácticas ágiles. Una de las claves para implantar una metodología ágil en una organización que realiza GSD se encuentra en tener la capacidad de adaptar las prácticas que defina la metodología a las necesidades de la organización y las implicaciones que tiene GSD. En la Figura 60 pueden verse algunas de las prácticas ágiles que se han implantado con mayor éxito en las organizaciones que implementan prácticas ágiles y GSD (Jalali & Wohlin, 2010).

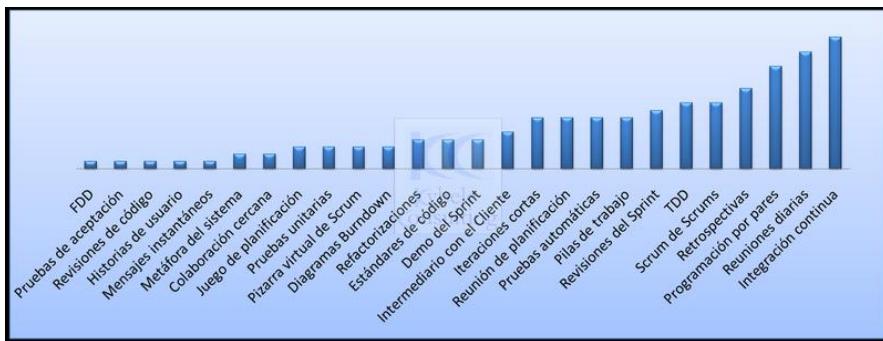


Figura 60. Prácticas ágiles y GSD

13. La implantación de prácticas ágiles en la empresa



13. La implantación de prácticas ágiles en la empresa

“Nada es veneno, todo es veneno: la diferencia está en la dosis” - Paracelso

La adopción de las metodologías ágiles por la organización no es tarea fácil. **La implantación de las prácticas ágiles** conlleva un proceso y como tal debe estar constituido por etapas y **debe realizarse en el marco de un proyecto de mejora**.

Las etapas principalmente son tres (Hu Guang-yong, 2011), las cuales vamos a ir desarrollando a continuación:

1. Conocimiento.
2. Aplicación.
3. Evaluación.

Estas etapas están estrechamente relacionadas entre sí a partir de **factores que influyen directamente sobre ellas y sus interrelaciones**. Estos factores son las personas, la organización y la tecnología (Figura 61).

13. La implantación de prácticas ágiles en la empresa



Figura 61. Etapas y factores de la implantación

13.1 Primera etapa: conocimiento

Esta primera etapa está orientada a **difundir, conocer y motivar al equipo de desarrollo**, formándolo en metodologías ágiles, su manifiesto, alcances y resultados.

Antes de decidirse a implantar cualquier metodología ágil dentro de una empresa u organización, se debe **plantear** un debate sobre **si interesa** hacer el esfuerzo de **implantar las prácticas ágiles** (M. Fowler, 2005), ya que conlleva realizar un cambio de mentalidad, y modificar los conceptos del pasado.

Es muy importante que la mayoría de las personas, departamentos o grupos que trabajan en una empresa **estén de acuerdo en querer implementar la metodología ágil** ya que existirán cambios en los hábitos y en la gestión.

13. La implantación de prácticas ágiles en la empresa

Importante: será necesario realizar una formación inicial de los conceptos relacionados con la metodología. Se pueden realizar grupos de trabajo y talleres para lograr la implicación de las personas en la organización y lograr el entendimiento teórico así como las aportaciones prácticas iniciales. Ninguna metodología se adapta 100% a una empresa.

El cambio de mentalidad comienza por **romper los esquemas tradicionales**: cadenas de mandos tradicionales, organización de trabajo, control de la productividad, confianza en las personas, delegación de responsabilidades, etc.

13.2 ¿Qué debemos saber antes de aplicar metodologías ágiles?

El cambio hacia una metodología ágil requiere también un alto esfuerzo por parte de la organización., **su aplicación debe estar justificada** ya que puede acarrear altos costes, baja funcionalidad, etc. En la adopción de metodologías ágiles debemos tener en cuenta que en la organización se producirán una serie de cambios y desafíos:

- **El equipo es responsable de la calidad.** Debido a que ellos serán los encargados de realizar la planificación de las iteraciones y el tiempo dedicado a las tareas. En ocasiones es interesante incluir mínimos de calidad o formas de medirla para que el mismo equipo vea su deuda técnica³⁴ (Irrazabal, 2011).
- **Dejan de existir los héroes y apagafuegos.** Una de las ventajas de las prácticas ágiles está en el equipo y en mejorar su comunicación. Es menos probable escuchar la frase: "Esto sólo lo sabe Juan".

³⁴ La deuda técnica es un concepto introducido por Ward Cunningham en el año 1992, y se define como las consecuencias y sobrecargos que se producen por aplicar malas prácticas durante el desarrollo software.

13. La implantación de prácticas ágiles en la empresa

- Una gestión de productos software más compleja. Es la consecuencia natural de tener entregables en cada iteración. Sin embargo este tema puede verse como una ventaja a la hora de la entrega final, la cuál será menos "traumática".
- Aumenta la comunicación y puede aumentar el conflicto. Surge la necesidad de que una persona asuma el rol de moderador en los equipos para evitar este tipo de inconvenientes.
- Los problemas y deficiencias de funcionalidad serán más visibles. Las entregas parciales y la mayor participación del cliente/usuario favorecen la detección temprana de errores y por lo tanto, disminuyen los costes posteriores y las "sorpresa" en iteraciones finales del proyecto.

13.3 Factores de éxito

Una vez que la organización y las personas están de acuerdo y mentalizadas para el cambio en la forma de trabajar, hay que tener en cuenta unos **factores críticos de éxito**. Los factores críticos de éxito establecerán la estrategia y modo de trabajo óptimo para minimizar el riesgo y maximizar el éxito en la implantación de las metodologías ágiles.

A continuación se muestran los principales factores críticos de éxito basados en diferentes casos de estudio sobre proyectos de implantación de metodologías ágiles. Se organizan en 5 categorías: organización, personal, proceso, tecnología y proyecto (Chow & Cao, 2008).

13. La implantación de prácticas ágiles en la empresa

Dimensión	Factor crítico de éxito
Organización	Apoyo de la directiva.
	Ayuda de un sponsor o manager.
	Cultura cooperativa.
	Alto valor al "cara a cara".
Personal	Equipos competentes y experimentados.
	Equipos motivados.
	Jefes de equipo con altos conocimientos en metodologías ágiles.
	Buena relación con el cliente.
Proceso	Seguimiento de gestión ágil de requisitos.
	Seguimiento de gestión ágil de proyectos.
	Seguimiento de gestión ágil de configuración.
	Fuerte comunicación focalizada en el cara a cara.
	Presencia regular del cliente.
Tecnología	Establecimiento de estándares de programación.
	Actividades rigurosas de refactorización.
	Preparación técnica del equipo.
	Correcta documentación.
Proyecto	No crítico.
	Dinámico.
	Equipos pequeños.
	Requisitos cambiantes.

Tabla 16. Factores críticos de éxito

Los desarrolladores o jefes de proyecto muchas veces buscan **implantar prácticas ágiles sin el apoyo total de la organización**. El problema está en que las prácticas se transforman en "pseudoágiles", realizando **retrabajo para mantener los criterios metodológicos de la organización**.

13. La implantación de prácticas ágiles en la empresa

La formación será de mucha utilidad para lograr la motivación del personal y conseguir un conocimiento homogéneo en todos los niveles. El papel de las personas que conocen el trabajo actual es fundamental en cualquier proceso de mejora.

13.4 Segunda etapa: aplicación

Esta segunda etapa del proceso tiene por objetivo acercar el Manifiesto Ágil al trabajo de la organización. Para ello se recomienda definir un proyecto de implantación.

El objetivo es avanzar y aplicar las buenas prácticas, empezando por preparar el entorno de desarrollo en temas como:

- **Mayor interrelación con el cliente.** El Product Owner deberá aumentar la comunicación con el equipo de desarrollo. Ello favorecerá la rapidez para la obtención de feedback por parte del Product Owner, el uso de entregables favorece este aspecto.
- **Reuniones de equipos.** La productividad en las reuniones es básica ya que son uno de los pilares de la gestión ágil de proyectos y se realizarán gran cantidad de ellas... Todas las reuniones tienen aquí un objetivo y un tiempo que deben ser respetados.
- **Roles del equipo.** Los roles en las prácticas ágiles no son jerárquicos. No existe un jefe que indica tareas y los demás obedecen. En la estimación, por ejemplo, es necesario que cada opinión valga lo mismo para evitar sesgar los resultados.
- **Administración de cambios.** Con las prácticas ágiles favoreceremos los cambios, lo cual no es síntoma de ausencia de toma de requisitos o no implicación del cliente/usuario.
- **Comprometer entregas.** La idea de compromiso no significa quedarse trabajando horas extra todos los días. De hecho eso sería un mal

13. La implantación de prácticas ágiles en la empresa

síntoma. Esta problemática se puede evitar mediante una buena gestión que favorecerá una mejor estimación y disminuirá las **expectativas no realistas** que pueda tener el cliente.

- **Seguimiento del proyecto.** Materializado en las reuniones y en los gráficos que hemos visto en el capítulo 7. Los equipos se autogestionan por lo que el seguimiento no es tanto un mecanismo de control, sino más bien una forma de comunicación.

Importante: se recomienda trabajar en esta fase con un proyecto de tamaño pequeño o mediano, con clientes con los que se pueda mantener comunicaciones periódicas y con equipos cohesionados.

13.5 Tácticas de implementación

Para superar las barreras del cambio se pueden seguir tres tácticas: trabajar con un coach (externo de la organización), con un sponsor (forma parte de la organización) o actuar como un “caballo de Troya”.

- **Un coach (externo a la organización):** su objetivo es dar el conocimiento necesario para aplicar las metodologías ágiles. Debe lograr que la organización vea como algo interesante la realización de cambios en el proceso de desarrollo y obtener su respaldo a la hora de realizarlos.
- **Un sponsor (interno a la organización):** tiene la misma función que el anterior, pero desde un punto interno de la empresa. Lo cual da un aspecto más continuo y de retroalimentación al adoptar la nueva metodología. Conoce más a la empresa aunque esto puede disminuir su objetividad y la forma en la que la organización responde ante las mejoras.

Recomendación: Encontrar un sponsor/coach es la forma más óptima de implantar una metodología de manera íntegra.

13. La implantación de prácticas ágiles en la empresa

- Existe un tercer camino más ágil y sencillo, consiste en actuar como un “caballo de Troya”. De esta manera, podemos ir **introduciendo prácticas ágiles paulatinamente**, descartando aquellas que no resulten evidentemente útiles, sin necesidad de realizar imposiciones y logrando el efecto buscado, es decir, entrega rápida de un producto maximizando el valor y la calidad percibida por el cliente. Esta vía del caballo de Troya también tiene sus desventajas ya que no se puede llevar a cabo a gran escala, sólo a nivel de equipos u organizaciones pequeñas. La ventaja principal es el **bajo coste**.

13.6 Tercera etapa: evaluación

Esta etapa está orientada a que el equipo de desarrollo decide si en base a lo conocido, aprendido y aplicado, es viable poder adoptar o ir mejorando la utilización de las metodologías ágiles. Para ello es necesario centrarse en dos cuestiones: ¿Ha ido todo según lo planificado? ¿Cuáles son los resultados?

Los **resultados** se orientan a poder identificar si se cumplieron los siguientes objetivos:

- **Mejoras en el proceso de desarrollo de software.** Con la aplicación de la técnica ágil se mejoraron los procesos de desarrollo software en sus diferentes etapas.
- **Mejoras en la valorización que hace el cliente o usuario** sobre el software. El cliente puede mencionar que el producto desarrollado es el esperado y cumple con los objetivos por los cuales fue diseñado.
- **Mejora en la productividad** de los miembros del equipo. Trabajar con objetivos y metas claras durante todo el proyecto incrementa los resultados esperados tanto a nivel de proyecto como de producto elaborado.

13. La implantación de prácticas ágiles en la empresa

- **Percepción global del equipo.** El equipo tiene la plena convicción de que mejorar sus procesos de desarrollo software es lo fundamental para poder obtener mejores resultados y ganar también en imagen como profesionales del software.
- **Valoración de la organización.** Aumento de la valoración del trabajo realizado y de la percepción que se tenga del mismo por parte de la organización.

13.7 Conclusiones y consideraciones sobre la implantación

En la práctica ambos enfoques, el ágil y el tradicional, conviven (Vinekar & Huntley, 2010), y lo seguirán haciendo durante muchos años. Cuando las organizaciones se plantean implantar prácticas ágiles raramente lo hacen al 100%. En lugar de ello, lo que hacen son adaptaciones al mismo, muchas veces incorporando prácticas tradicionales como por ejemplo, el robustecer y hacer menos iterativa la fase de diseño de la arquitectura software.

Consejo: Es muy importante no olvidar que la teoría ágil se puede adaptar a cada caso en particular con el fin de obtener los máximos beneficios, muchas veces relajando la agilidad y obteniendo la verdadera riqueza y productividad de las múltiples soluciones que ofrece la ingeniería del software.

Desarrollar software no es una tarea fácil, y nunca lo será. Por ello existen numerosas propuestas metodológicas que intentan minimizar los problemas del desarrollo. Pero nunca debemos olvidar que siempre será necesario adaptar estas propuestas metodológicas a nuestra empresa, equipo y línea de negocio.

En cualquier caso las metodologías se pueden agrupar en dos grandes categorías: **las tradicionales y las ágiles.**

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

Anexo

A

A. Herramientas para la gestión ágil

Este anexo enumera una serie de herramientas que facilitan las tareas de gestión y seguimiento de los proyectos ágiles

Se debe tener en cuenta que cualquier herramienta de gestión de proyectos, de gestión de incidencias, herramientas colaborativas, etc., se puede llegar a adaptar a las prácticas de los proyectos ágiles. No obstante, existen herramientas que se han desarrollado específicamente para trabajar con metodologías ágiles. También conviene recordar que estas herramientas de gestión deben complementar la gestión ágil con otras más cercanas al desarrollo o la programación, como las de integración continua, métricas, control de versiones, etc., las cuales están fuera del alcance de este anexo

En este capítulo se expone una breve descripción (con las características más relevantes) de cada herramienta. Van acompañadas de su tipo de licencia y de su página oficial, donde se podrá acceder a toda información asociada a la herramienta. Cabe destacar que aunque muchas herramientas presentan licencias comerciales, suelen poseer a su vez licencias libres para un número de usuarios y/o un número de proyectos determinado (para un mejor estudio de la licencia de cada una de las herramientas se recomienda visitar las páginas oficiales).

Herramientas para la gestión ágil

A.1 Herramientas más usadas

Antes de describir las herramientas comentaremos un estudio del año 2011 realizado por VersionOne (VersionOne Inc., 2011) que intenta reflejar el estado de las prácticas ágiles en la industria del software. En este estudio se muestra un gráfico incluido que responde a qué tipos de herramientas son las más utilizadas en proyectos que siguen metodologías ágiles.

Del estudio (Figura 62) se obtiene que las herramientas actualmente más utilizadas son los gestores de incidencias, las hojas de cálculo, los tableros de tareas y las wikis. Se puede observar a su vez que las herramientas que cubren dos de las prácticas ágiles descritas en el capítulo 9: herramientas de integración continua /builds automáticos y de pruebas unitarias, se encuentran entre las 8 primeras posiciones.

De este estudio también resulta interesante observar que las herramientas de gestión de ideas son las herramientas menos utilizadas en los proyectos ágiles pero a su vez suelen ser las más deseadas por los encuestados.

Se podría concluir que son las herramientas no propias de las metodologías ágiles (pero totalmente adaptables a este tipo de proyectos) las que se implantan en un primer momento, muy probablemente por el hecho de que los equipos ya están familiarizados con ellas. Como se ha comentado en el resto de libro, la introducción tanto de las prácticas ágiles como de las herramientas que las cubren se recomienda que se realice de manera gradual para causar el mínimo impacto posible en los equipos de trabajo.

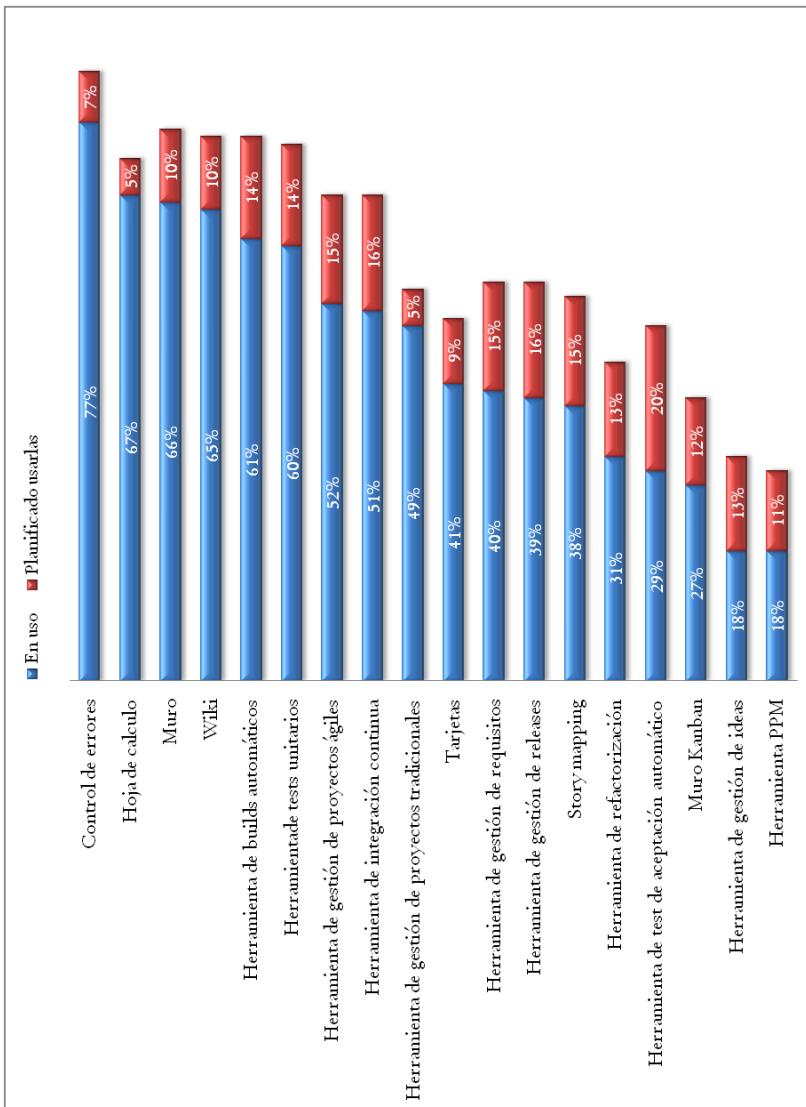


Figura 62. Tipos de herramientas más usadas en proyectos ágiles

Herramientas para la gestión ágil

A.2 Herramientas generales de gestión y seguimiento de proyectos

Finalmente, en este apartado se listarán una serie de herramientas que pueden ayudar a la gestión ágil de proyectos. Aunque en los pequeños equipos la visibilidad del proyecto, su gestión y seguimiento puedan ser realizados de forma “manual” mediante una pizarra, notas, etc., es posible utilizar herramientas que ayuden a la gestión ágil. Así mismo, si se trabajan con equipos o personas separadas físicamente las herramientas cobran un papel principal en la mejora de la comunicación (para ver este tema con mayor profundidad ir al capítulo 12 de este libro).

Las descripciones de cada herramienta se han basado en las revisiones realizadas por las comunidades: (*Fábricas de software*.2011), (*Agile software development news*.2012), (*Know scrum*.2012), (*User stories*.2012) y por la información obtenida de las páginas oficiales de cada herramienta. En la Tabla 17 se resumen herramientas generales de gestión y seguimiento de proyectos ágiles. A continuación se describen algunas de estas herramientas, relacionadas con las buenas prácticas ágiles:

Nombre	Descripción	Licencia
Agile Bench	Herramienta de interfaz intuitiva y sencillo flujo de trabajo. Permite la gestión efectiva de equipos, el mantenimiento del backlog (o lista de tareas), planificación de iteraciones y estimación del trabajo así como estar al tanto en todo momento de los errores y problemas surgidos en el proyecto. http://agilebench.com/	De pago

Nombre	Descripción	Licencia
Agile Express	Agile Express es una herramienta libre que proporciona módulos para gestionar y registrar historias de usuario y tareas, proyectos e iteraciones. A su vez contiene un tablero Kanban virtual que muestra el progreso de las historias de usuario y tareas. Ofrece también la posibilidad de generar gráficos BurnDown y de comparación de velocidad entre iteraciones. http://agileexpress.sourceforge.net/	Libre
Agile Fant	Herramienta Web que permite la gestión y el seguimiento de las iteraciones con gráficos de avance y gestión de backlog a nivel de producto, proyecto y Sprint. Permite la realización automatizada de informes. http://www.agilefant.org/	Libre
Agile Tracking Tool	Herramienta libre que facilita la manera de trabajar con Scrum y Kanban. Permite ordenar los elementos de la pila de producto y de la pila de Sprint por categorías, permitiendo añadir a cada elemento los criterios de aceptación y los comentarios oportunos. Posee un mecanismo para medir la velocidad de iteración y estimar tareas de acuerdo a la misma. http://sites.google.com/site/agiletrackingtool/home	Libre

Herramientas para la gestión ágil

Nombre	Descripción	Licencia
Agilito	Agilo es una herramienta Web libre que se encuentra orientada a la gestión ágil con Scrum. Es altamente configurable para adaptarse a cualquier flujo de trabajo. Está diseñada y desarrollada para cada rol involucrado en un proyecto ágil. Sus características destacadas son que permite organizar de manera eficiente la pila de producto, planificar, estimar y comprobar el avance de los Sprint, así como gestionar las historias de usuario. http://www.agiloforscrum.com/	De pago
Airgile	Airgile permite la planificación y control de la evolución del proyecto y sus tareas, simplificando la colaboración entre equipos y la gestión de tareas del proyecto. Cabe destacar que Airgile está disponible como herramienta en línea. http://en.airgile.com/	De pago
Banana Scrum	Servicio Web que proporciona soporte de múltiples proyectos y funcionalidades orientadas a Scrum tradicional. http://www.bananascrum.com/	De pago

Nombre	Descripción	Licencia
Bright Green Projects	<p>Herramienta de gestión de proyectos ágiles pensada para Scrum y Kanban que se caracteriza por su simplicidad y transparencia. Permite realizar estimaciones gracias al cálculo de la velocidad de trabajo en cada iteración. Contiene pantallas interactivas simulando las pizarras Kanban y la posibilidad de generar gráficos de avance BurnDown. También ofrece la posibilidad de almacenar imágenes o videos asociados a las historias de usuario.</p> <p>http://brightgreenprojects.com/</p>	De pago
Daily-Scrum	<p>Daily-Scrum es una herramienta Web desarrollada por Scrum Masters y jefes de proyectos ágiles. Su desarrollo se ha apoyado en la experiencia de miembros de equipos ágiles. Su principal peculiaridad es que se adapta a cualquier metodología ágil: Scrum, DSDM, eXtreme Programming, Lean, etc. Solamente es necesario que el proyecto siga ciclos cortos, iterativos e incrementales.</p> <p>http://daily-scrum.com/</p>	De pago
Easy Backlog	<p>Herramienta muy intuitiva para la gestión de tareas (backlog) únicamente. Cabe destacar la facilidad de uso de su interfaz, los paneles de métricas y estadísticas de los que dispone, así como la facilidad de asignar tareas a cada uno de los Sprint.</p> <p>http://easybacklog.com</p>	Libre

Herramientas para la gestión ágil

Nombre	Descripción	Licencia
Explain PMT	Herramienta libre para la gestión de proyectos ágiles especialmente si la metodología es eXtreme Programming. Es fácilmente adaptable a las necesidades de un proyecto concreto. https://github.com/explainpmt/explainpmt	Libre
FireScrum	FireScrum es una herramienta libre desarrollada para apoyar la gestión de proyectos bajo la metodología ágil Scrum. Es una aplicación Web intuitiva y fácil de usar. Contiene la técnica de estimación de Planning Poker con audio y video, así como un módulo de gestión de incidencias. http://sourceforge.net/projects/firescrum/	Libre
Flow	Flow facilita la gestión y el seguimiento de proyectos basados en Kanban. Contiene pizarras Kanban virtuales, permite realizar gráficos y comentarios de las tareas y obtiene métricas y estadísticas del progreso de los proyectos. También posibilita realizar un filtrado de la información que se desea mostrar (por usuario, por proyecto, etc.). http://flow.io/	De pago
Fulcrum	Herramienta libre para la gestión de listas de tareas (backlogs) basadas en historias de usuario. http://wholemeal.co.nz/projects/fulcrum.htm	Libre

Nombre	Descripción	Licencia
Gravity-Dev	<p>GravityDev está diseñada para gestionar proyectos ágiles. Como en la mayoría de estas herramientas, se pueden definir y gestionar historias de usuario, crear casos de prueba de aceptación, asignar costes y planificar el trabajo en iteraciones. Lo que tiene de especial esta herramienta es una aplicación de Google Chrome totalmente integrada con Google Apps.</p> <p>https://www.gravitydev.com</p>	De pago
Green-Hopper	<p>GreenHopper es una completa herramienta de gestión de proyectos ágiles usada en más de 6.000 empresas, facilitando las tareas de gestión y seguimiento con Scrum, Kanban o XP. Posee más de 400 plugins para integrarse con otras herramientas.</p> <p>http://www.atlassian.com/software/greenhopper/overview</p>	De pago
IceScrum	<p>Herramienta libre para Scrum y Kanban. Ofrece las opciones de operación, consulta y estimación de historias de usuario. Permite añadir historias de usuario a la pila de producto, dividir el tiempo en Sprints y mover estas historias de la pila de producto a cada uno de los Sprint. Posee la técnica de Planning Poker para la estimación y paneles virtuales.</p> <p>http://www.icescrum.org</p>	Libre

Herramientas para la gestión ágil

Nombre	Descripción	Licencia
Kanbanize	Kanbanize es una herramienta Web para la gestión de proyectos basados en Kanban. Su principal fortaleza es la monitorización que permite realizar sobre el proyecto apoyado en un completo conjunto de métricas y gráficos. http://kanbanize.com/	Libre
Kunagi	Herramienta Web gratuita que proporciona un sistema integral de gestión de proyectos, complementando Scrum con un conjunto de buenas prácticas para cubrir todas las necesidades en las actividades de gestión de proyecto. Ofrece herramientas colaborativas y otras facilidades como cuadro de mando del proyecto, panel interactivo para el Sprint actual o la técnica de Planning Poker de estimación. http://kunagi.org/	Libre
Leankit Kanban	Leankit Kanban es una completa herramienta para la gestión de proyectos basados en Kanban. Aparte de todas las actividades asociadas a la gestión de este tipo de proyectos, posee un sistema de alertas de tareas bloqueadas para notificar el porqué del bloqueo al resto de miembros del equipo. http://leankitkanban.com	De pago
Mingle	Mingle contiene todas las funcionalidades de este tipo de herramientas de gestión y seguimiento de proyectos ágiles con la peculiaridad de que lleva integrada una plataforma de gestión de pruebas y de incidencias. http://www.thoughtworks-studios.com/mingle-agile-project-management .	De pago

Nombre	Descripción	Licencia
OnTime	<p>La peculiaridad de OnTime es que aparte de facilitar todas las actividades propias de la gestión de proyectos ágiles, incluye un módulo de gestión de incidencias y se integra con facilidad con sistemas de control de versiones (Git, Subversion, etc.) y con IDEs como Visual Studio o Eclipse.</p> <p>http://www.axosoft.com/ontime</p>	De pago
Pango Scrum	<p>Herramienta online para la gestión de proyectos basada en Scrum con una interfaz sencilla y amigable que permite escribir, estimar y priorizar la pila de producto. Facilita en gran medida la planificación de Sprints y sus reuniones asociadas.</p> <p>http://pangoscrum.com</p>	Libre
PlanBox	<p>PlanBox contiene todas las funcionalidades deseadas en una herramienta de gestión y seguimiento de proyectos ágiles. Es una herramienta recomendada si se quiere que todos los roles clave de las metodologías ágiles estén involucradas en el proceso. Como peculiaridades respecto al resto, posee una aplicación muy intuitiva y se integra con sistemas de control de versiones (Git, Subversion, etc.). Además está disponible como una aplicación para móviles y como una aplicación de Google Chrome.</p> <p>http://www.axosoft.com/ontime</p>	De pago

Herramientas para la gestión ágil

Nombre	Descripción	Licencia
Pivotal Tracker	<p>Herramienta para la gestión ágil de proyectos con una interfaz intuitiva y muy amigable. Entre sus características más peculiares es que se integra con herramientas como JIRA, Lighthouse, Bugzilla, Satisfaction, Zendesk e incluye un interfaz muy flexible para poder integrarse con cualquier otra herramienta. Su sección de gráficos y generación de informes es muy potente.</p> <p>http://www.pivotaltracker.com</p>	De pago
Rally	<p>Gestor de proyectos online (tipo SaaS) orientado a los proyectos que usan metodologías ágiles como Scrum o Kanban. Integra entre sus herramientas principales: módulo de métricas y gráficos que apoyan el proceso de aseguramiento de la calidad, módulo de generación de informes y módulos de gestión de casos de prueba y de defectos o incidencias.</p> <p>http://www.rallydev.com</p>	De pago
RedCritter Tracker	<p>Herramienta completa de gestión de proyectos ágiles que está enfocada como un juego (la técnica es conocida como Gamification). La herramienta premia con insignias virtuales al equipo conforme van consiguiendo una serie de objetivos. Por lo demás, contiene todas las necesidades deseadas en este tipo de herramientas.</p> <p>http://redcrittertracker.com/</p>	De pago
Scrum 2Go	<p>Scrum2Go es una aplicación para iPhone o iPod Touch que cubre todas las actividades del día a día en proyectos ágiles.</p> <p>http://www.bitcoders.com/</p>	De pago

Nombre	Descripción	Licencia
Scrum-Desk	Gestor de proyectos en equipos ágiles usando Scrum. Entre sus funcionalidades permite administrar la pila de producto, los Sprints, las distribuciones, las historias y tareas como cualquier otra herramienta de gestión ágil. Como características no tan comunes permite la gestión del tiempo usando la técnica de Pomodoro y posee un módulo de gestión de impedimentos. También posee un cliente para iPhone. http://www.scrumdesk.com/	De pago
ScrumDo	ScrumDo es una herramienta que sigue la filosofía Scrum, centrándose en la simplicidad y en la facilidad de uso. No obstante, permite gestionar las listas de tareas (backlog) e historias de usuario, crear y gestionar iteraciones (a través de una pantalla interactiva), obtener gráficos de avance BurnUp y permite realizar estimaciones a través de la técnica de Planning Poker. http://www.scrumdo.com/	De pago
Scrumpy	Herramienta especialmente orientada a ayudar a Product Owners en la gestión de la pila del producto y sus historias de usuario. http://www.scrumpytool.com/	Libre
Scrum Sprint Planner	Scrum Sprint Planner es una aplicación para iPad que supone un gestor de proyectos ágiles basado en los principios de Scrum. http://www.darsoft.com/index.php/products/agile-project-manager	De pago

Herramientas para la gestión ágil

Nombre	Descripción	Licencia
Scrum-Works	<p>Herramienta para la gestión de proyectos de desarrollo basados en la metodología ágil Scrum. Se puede utilizar desde un cliente Web o una aplicación de escritorio. Como peculiaridad cabe destacar que posee un módulo para el cálculo del retorno de inversión y proporciona una alta visibilidad de los impedimentos encontrados en el proyecto. Contiene un robusto servicio Web para integrarse con otras herramientas como Jira, BugZilla, Eclipse, etc.</p> <p>http://www.open.collab.net/products/scrum-works</p>	De pago
Sprintometer	<p>Herramienta gratuita para gestión, medición y seguimiento de proyectos Scrum y eXtreme Programming. Para simplificar el intercambio de datos exporta sus gráficos e informes a Microsoft Excel. Posee gráficos de avance BurnDown en 3D.</p> <p>http://Sprintometer.com</p>	De pago
Swift Kanban	<p>Swift Kanban es una aplicación de gestión de proyectos Kanban con un potente tablero Kanban, detecta los cuellos de botella y permite configurar el valor WIP (Work-In-Progess). Por otro lado es una herramienta personalizable y contiene un completo módulo de métricas que posibilita detectar oportunidades de mejora.</p> <p>http://www.swift-kanban.com/kanban-tool.html</p>	De pago

Nombre	Descripción	Licencia
Target-Process	<p>Herramienta online para la gestión de proyectos de desarrollo de software que siguen metodologías ágiles. Es una herramienta altamente configurable. Como características menos comunes cabe destacar sus módulos de gestión de defectos y de gestión de casos de prueba. Se integra con múltiples herramientas como Subversion, Bugzilla, JIRA, Visual Studio, NUnit, Selenium, etc.</p> <p>http://www.targetprocess.com</p>	De pago
Team Pulse	<p>Herramienta online de gestión de proyectos ágiles. Contiene todo lo necesario para facilitar dichas actividades de gestión. Como diferencia del resto, posee un analizador de buenas prácticas de gestión, así como una sección específica para el Product Owner donde recibir un feedback de todos los miembros del equipo sobre diferentes aspectos del proyecto.</p> <p>http://www.telerik.com/agile-project-management-tools.aspx</p>	De pago
TinyPM	<p>Herramienta online para colaborar en los procesos ágiles de desarrollo de software. Entre sus funcionalidades se encuentran la gestión de productos, backlog, taskboard, historias de usuario y una wiki. Cabe destacar de esta herramienta su flexibilidad.</p> <p>http://www.tinypm.com</p>	De pago

Tabla 17. Herramientas generales de gestión y seguimiento de proyectos ágiles

Buyer: Luis Lopez (llopez01@TechEmail.com)
Transaction ID: 7NB93637373806051

Referencias

Agile software development news. (2012). Fecha de consulta: 17 mayo 2012.

Disponible en: <http://agilescout.com>

Ambler, S. (2008). *Acceleration: An agile productivity measure.* Fecha de consulta: 17 mayo 2012. Disponible en:

https://www.ibm.com/developerworks/mydeveloperworks/blogs/ambler/entry/metric_acceleration?lang=en

Anderson, D. (2003). *Agile management for software engineering: Applying the theory of constraints for business results* Prentice Hall PTR.

Anderson, D. (2009). *A kanban system for software engineering.* Fecha de consulta: 17 mayo 2012.

Disponible en: <http://es.scribd.com/doc/13748689/A-Kanban-System-for-Software-Engineering>

Appleton, B. Berczuk, S. et al. (1998). *Streamed lines: Branching patterns for parallel software development.*

Disponible en:

<http://www.cmcrossroads.com/bradapp/acme/branching/>

Auer, K., & Miller, R. (2002). *Extreme programming applied: Playing to win* Addison-Wesley.

Austin, R. D. (2007). *CMM versus agile: Methodology wars in software Development*

(Case study No. 9-607-084). Harvard, USA. Harvard Business School Press.

Referencias

- Beamount, S. (2008). *Agile & contracts. scrum gathering stockholm.* Disponible en: <http://www.scrumalliance.org/resources/442>
- Beck, K., & Andres, C. (2004). *Extreme programming explained: Embrace change* Addison-Wesley Professional.
- Beck, K., & Cleal, D. (1999). *Optional scope contracts.* (White Paper ed.) Three Rivers.
- Beck, K. (2002). *Test driven development: By example.* Boston. Addison-Wesley Professional.
- Beck, K., et al. (2001). *Agile manifesto.* Fecha de consulta: 17 mayo 2012. Disponible en: <http://agilemanifesto.org>
- Bran, S. (2009). Agile documentation, anyone? *IEEE Softw.*, 26(6), 11-12.
- Chow, T., & Cao, D. (2008). A survey study of critical success factors in agile software projects. *J.Syst.Softw.*, 81(6), 961-971.
- Clegg, D., & Barker, R. (1994). *Case method fast-track: A rad approach.* Boston, MA, USA. Addison-Wesley Longman Publishing Co., Inc.
- CMMI Product Team. (2010). In CMMI Product TeamEditors (Ed.), *CMMI for development version 1.3. improving processes for developing better products and services.* Pittsburg, IL, USA. Carnegie Mellon University.
- Cockburn, A. (2006). *Agile contracts.* Fecha de consulta: 17 mayo 2012. Disponible en: <http://alistair.cockburn.us/Agile+contracts/v/multi>
- Cockburn, A. (2002). *Agile software development.* Boston, MA, USA. Addison-Wesley Longman Publishing Co., Inc.

Referencias

Cockburn, A. (2008). *Why I still use use cases*. Fecha de consulta: 17 mayo 2012.

Disponible en:

<http://alistair.cockburn.us/Why+I+still+use+use+cases>

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

Cohn, M. (2005). *Agile estimating and planning*. Upper Saddle River, NJ, USA. Prentice Hall PTR.

Donald, R. (1999). *Practical analysis for refactoring*. Champaign, IL, USA. University of Illinois at Urbana-Champaign.

Doshi, H. (2010). *Template of task breakdown for a user story*. Fecha de consulta: 17 mayo 2012. Disponible en:

<http://www.practiceagile.com/2010/08/template-of-task-breakdown-for-user.html>

Fábricas de software. (2011). Fecha de consulta: 17 mayo 2012.

Disponible en: <http://www.fabricasdesoftware.es/>

Feathers, M. (2004). *Working effectively with legacy code*. Upper Saddle River, NJ, USA. Prentice Hall PTR.

Fowler, M. (2005). *The new methodology*. Fecha de consulta: 17 mayo 2012.

Disponible en:

<http://martinfowler.com/articles/newMethodology.html>

Fowler, M. (2008). *Is design dead?* Fecha de consulta: 17 mayo 2012.

Disponible en: <http://martinfowler.com/articles/designDead.html>

Referencias

Fowler, M. (2005). *Should you go agile?* Fecha de consulta: 17 mayo 2012.

Disponible en:

<http://martinfowler.com/articles/newMethodology.html#ShouldYouGoAgile>

Fowler, M. (2008). *Agile Versus Lean*. Fecha de consulta: 17 mayo 2012.

Disponible en:

<http://martinfowler.com/bliki/AgileVersusLean.html>

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999).

Refactoring: Improving the design of existing code (1st edition ed.) Addison-Wesley Professional.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2000).

Refactoring: Improving the design of existing code (1st ed.) Addison-Wesley Professional.

Garzás, J. (2010). *Veterano ciclo iterativo e incremental..* Fecha de consulta:

17 mayo 2012.

Disponible en: <http://www.javiergarzas.com/2010/01/veterano-ciclo-de-vida-iterativo-incremental.html>, [2010].

Garzás, J. (2010). *Desarrollo ágil o tradicional ¿existe el punto intermedio?*

Fecha de consulta: 17 mayo 2012.

Disponible en: <http://www.javiergarzas.com/2010/08/agiles-formales-e-hibridos.html>

Garzás, J. (2011). *Dos razones por las que fabricar software no es lo mismo que fabricar coches o construir casas.* Fecha de consulta: 17 mayo 2012.

Disponible en: <http://www.javiergarzas.com/2011/02/diferencias-software-fabricacion-tradicional-1.html>

Referencias

- Garzás, J., Irrazábal, E., & Santa Escolástica R. (2011). Guía práctica de supervivencia en una auditoría CMMI. *Boletín De La ETSII, Universidad Rey Juan Carlos*, 002, 1-33.
- Garzás, J., & Paultk, M. (2012). Can scrum help to improve the project management process? A study of the relationships between scrum and project management process areas of CMMI-DEV 1.3. *Software Engineering Process Group*, Madrid. pp. 1-12.
- Glazer, H., Anderson, D., Anderson, D. J., Konrad, M., & Shrum, S. (2008). CMMI® or agile : Why not embrace both ! *IEEE Transactions on Geoscience and Remote Sensing*, 33(November), 48.
- Hartmann, D., & Dymond, R. (2006). Appropriate agile measurement: Using metrics and diagnostics to deliver business value. Artículo presentado en *Proceedings of the Conference on AGILE 2006*, IEEE Computer Society, pp. 126-134.
- Harvey, D. (2004). *Lean, agile*. Fecha de consulta: 17 mayo 2012.
Disponible en: <http://www.davethehat.com>
- Haugen, N. C. (2006). An empirical study of using planning poker for user story estimation. Artículo presentado en *Agile Conference, 2006*, pp. 9.
- Herbsleb, J. D., & Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. *Software Engineering, IEEE Transactions on*, 29(6), 481.
- Highsmith, J. (2001). *History: The agile manifesto*. Fecha de consulta: 17 mayo 2012.
Disponible en: <http://www.agilemanifesto.org/history.html>

Referencias

- Hoda, R., Noble, J., & Marshall, S. (2009). Negotiating contracts for agile projects: A practical perspective. *Agile Processes in Software Engineering and Extreme Programming*, 31, 186-191.
- 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.
- Hu Guang-yong. (2011). Study and practice of import scrum agile software development. Artículo presentado en *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pp. 217-220.
- IEEE. (2004). *SWEBOK: Guide to the software engineering body of knowledge*. Los Alamitos, California, USA.
- Irrazabal, E. (2011). *La deuda técnica*. Fecha de consulta: 17 mayo 2012. Disponible en: <http://www.fabricasdesoftware.es/la-deuda-tecnica-software-calidad.html>
- ISO/IEC 12207:2008 systems and software engineering -- software life cycle processes.*
- Jakobsen, C. R., & Johnson, K. A. (2008). Mature agile with a twist of CMMI. Artículo presentado en *Proceedings of the Agile 2008*, pp. 212-217.
- Jakobsen, C. R., & Sutherland, J. (2009). Scrum and CMMI going from good to great. Artículo presentado en *Proceedings of the 2009 Agile Conference*, pp. 333-337.

Referencias

- Jalali, S., & Wohlin, C. (2010). Agile practices in global software engineering - A systematic map. Artículo presentado en *Global Software Engineering (ICGSE), 2010 5th IEEE International Conference on*, pp. 45.
- Jeffries, R. (2001). *Essential XP: Card, conversation, confirmation*. Fecha de consulta: 17 mayo 2012. Disponible en:
<http://xprogramming.com/articles/expcardconversationconfirmation/>
- Jeffries, R. (2004). *A metric leading to agility*. Fecha de consulta: 17 mayo 2012.
Disponible en: <http://xprogramming.com/articles/jatrtsmetric/>
- Kaplan, R. S., & Norton, D. P. (1992). The balanced scorecard--measures that drive performance. *Harvard Business Review*, 83(7/8), 172-180.
- Kniberg, H. (2008). *Version control for multiple agile teams*. Fecha de consulta: 17 mayo 2012. Disponible en:
<http://www.infoq.com/articles/agile-version-control>
- Kniberg, H., & Skarin, M. (2010). *Kanban and scrum - making the most of both* Lulu Enterprises Inc.
- Know scrum. (2012). Fecha de consulta: 17 mayo 2012.
Disponible en: <http://knowscrum.com/>
- Kuphal, M. (2011). *Agile planning: My top five tips on decomposing user stories into tasks*. Fecha de consulta: 17 mayo 2012. Disponible en:
<http://mkuphal.wordpress.com/2011/05/13/agile-planning-my-top-five-tips-on-decomposing-user-stories-into-tasks/>

Referencias

- Lanza, M., & Marinescu, R. (2006). *Object-oriented metrics in practice. using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer.
- Larman, C. & Basili, V. (2003). Iterative and incremental Development: A brief History. *Computer* 36(6), 47-56.
- Larman, C., & Vodde, B. (2010). *Practices for scaling lean & agile development* Addison-Wesley Professional.
- Marcal, A., Furtado, F., & Belchior, A. (2007). Mapping CMMI project management process areas to SCRUM practices. Artículo presentado en *Proceedings of the 31st IEEE Software Engineering Workshop*, pp. 13-22.
- Maven.* (2012). Fecha de consulta: 17 mayo 2012.
Disponible en: <http://maven.apache.org/>
- McConnell, S. (1996). *Rapid development* Microsoft Press.
- McConnell, S. (2006). *Software estimation: Demystifying the black art.* Redmond, WA, USA. Microsoft Press.
- Opdyke, W. F. (1992). *Refactoring object oriented frameworks.* Champaign, IL, USA. University of Illinois at Urbana-Champaign.
- Parkinson, C. N. (1957). *La ley de parkinson.*
Disponible en:
<http://www.bioinfo.uib.es/~joemiro/teach/.../parkLaw.pdf>
- Paulk, M. C. (2001). Extreme programming from a CMM perspective. *IEEE Software*, 18(6), 19-26.

Referencias

- Piattini, M., Manzano, J., Cervera, J., & Fernández, L. (2003). *Análisis y diseño de aplicaciones informáticas de gestión – una perspectiva de ingeniería del software* (2^a ed.). Madrid. Ra-Ma.
- Pichler, R. (2010). *Agile product management with scrum: Creating products that customers love* Addison-Wesley Professional.
- PMD. (2012). Fecha de consulta: 17 mayo 2012.
Disponible en: <http://pmd.sourceforge.net/>
- Poppendieck, M., & Poppendieck, T. (2003). *Lean software development: An agile toolkit* Addison-Wesley.
- Potter, N., & Sakry, M. (2011). *Implementing scrum (agile) and CMMI® together*. Fecha de consulta: 17 mayo 2012.
Disponible en:
<http://www.agilejournal.com/articles/columns/column-articles/5794-implementing-scrum-agile-and-cmmi-together>
- Rawsthorne, B. D. (2008). Monitoring scrum projects with AgileEVM and earned business value (EBV) metrics. *Agile Journal*, , 1-14.
- Rawsthorne, D. (2006). *Calculating earned business value for an agile project*. Fecha de consulta: 17 mayo 2012. Disponible en:
http://danube.com/system/files/CollabNet_WP_Earned_Business_Value_041910.pdf
- Rumpe, B., & Schröder, A. (2002). Quantitative survey on extreme programming projects. Artículo presentado en *Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002*, pp. 26-30.

Referencias

- Schwaber, K. (2007). *The enterprise and scrum* (1º ed.). Redmond, WA, USA. Microsoft Press.
- Schwaber, C., Leganza, G. & D'Silva, D. (2007). *The truth about agile processes*. Fecha de consulta: 17 mayo 2012.
Disponible en:
http://www.forrester.com/imagesV2/uplmisc/NN_AppDev2.pdf
- Schwaber, K. (2004). *Agile project management with scrum* Microsoft Press.
- Schwaber, K., & Sutherland, J. (2010). *Scrum guide*. Fecha de consulta: 17 mayo 2012. Disponible en:
<http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20-%20ES.pdf#view=fit>
- Stellman, A. (2009). *Requirements 101: User stories vs use cases*. Fecha de consulta: 17 mayo 2012.
Disponible en: <http://www.stellman-greene.com/2009/05/03/requirements-101-user-stories-vs-use-cases/>
- Stephens, M., & Rosenberg, D. (2003). *Extreme programming refactored: The case against XP* APress LP.
- Sulaiman, T., Barton, B., & Blackburn, T. (2006). AgileEVM-earned value management in scrum projects. Artículo presentado en *Agile Conference, 2006*, pp. 10 pp.-16.
- Sutherland, J. (2001). Agile can scale: Inventing and reinventing scrum in five companies. *14(12)*
- Sutherland, J. (2006). *Why gantt charts were banned in the first scrum*. Fecha de consulta: 17 mayo 2012.

Disponible en: <http://scrum.jeffsutherland.com/2006/02/why-gantt-charts-were-banned-in-first.html>

Sutherland, J. (2008). *Money for nothing and your change for free. agile 2008 presentation.* Fecha de consulta: 17 mayo 2012.

Disponible en:

<http://jeffsutherland.com/Agile2008MoneyforNothing.pdf>

Sutherland, J. (2010). *Agile principles and values.* Fecha de consulta: 17 mayo 2012. Disponible en: <http://msdn.microsoft.com/en-us/library/dd997578.aspx>

Sutherland, J., Jakobsen, R., & Johnson, K. (2008). Scrum and CMMI level 5: The magic potion for code warriors. Artículo presentado en *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, pp. 466.

Sutherland, J., & Schwaber, K. (2011). *The scrum papers: Nut, bolts, and origins of an agile framework.* Fecha de consulta: 17 mayo 2012.

Disponible en: <http://jeffsutherland.com/ScrumPapers.pdf>

Sutherland, J., & Johnson, K. (2010). *Using scrum to avoid bad CMMI-DEV®implementation.* Fecha de consulta: 29 octubre 2010.

Disponible en: <http://aplndc.com/eventSlides/JeffSutherlandCMMI-DEV.pdf>

Tokuda, L., & Batory, D. (2001). Evolving object-oriented designs with refactorings. *Automated Software Engg.*, 8(1), 89-120.

User stories. (2012). Fecha de consulta: 17 mayo 2012.

Disponible en: <http://www.userstories.com/>

Referencias

VersionOne Inc. (2011). *State of agile survey 2011*. Fecha de consulta: 17 mayo 2012. Disponible en:

http://www.versionone.com/state_of_agile_development_survey/11/

Vinekar, V., & Huntley, C. L. (2010). Agility versus maturity: Is there really a trade-off? *Computer*, 43(5), 87-89.

Wake, B. (2003). *INVEST in good stories, and SMART tasks*. Fecha de consulta: 17 mayo 2012. Disponible en:

<http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/>