Universidad Mayor de San Andrés
Carrera de Informática
Asociación de Investigadores en Software Inteligente
A I S I

MONOGRAFÍA

MODELADO ÁGIL Y PROGRAMACIÓN EXTREMA

Julio, 2006.
La Paz - Bolivia

CONTENIDO

1. INTRODUCCIÓN

En las dos últimas décadas las notaciones de modelado y posteriormente las herramientas pretendieron ser las "balas de plata" para el éxito en el desarrollo del software, sin embargo, las expectativas no fueron satisfechas. Esto se debe en gran parte a que otro importante elemento, la metodología de desarrollo, había sido postergado. De nada sirven buenas notaciones y herramientas si no se proveen directivas para su aplicación. Así, esta década ha comenzado con un creciente interés en metodologías de desarrollo. Hasta hace poco el proceso de desarrollo llevaba asociada un marcado énfasis en el control del proceso mediante una rigurosa definición de roles, actividades y artefactos, incluyendo modelado y documentación detallada. Este esquema "tradicional" para abordar el desarrollo del software ha demostrado ser efectivo y necesario en proyectos de gran tamaño (respecto a tiempo y recursos), donde por lo general se exige un alto grado de ceremonia en el proceso. Sin embargo, este enfoque no resulta ser el más adecuado para muchos de los proyectos actuales donde el entorno del sistema es muy cambiante, y en donde se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. Ante las dificultades para utilizar metodologías tradicionales con estas restricciones de tiempo y flexibilidad, muchos equipos de desarrollo se resignan a prescindir del “buen hacer” de la ingeniería del software, asumiendo el riesgo que ello conlleva. En este escenario, las metodologías ágiles emergen como una posible respuesta para llenar ese vacío metodológico. Al estar especialmente orientadas para proyectos pequeños, las metodologías ágiles constituyen una solución a medida para ese entorno, aportando una elevada simplificación que a pesar de ello no renuncia a las prácticas esenciales para asegurar la calidad del producto. Las metodologías ágiles son sin duda uno de los temas recientes en ingeniería del software que están acaparando gran interés. Prueba de ello es que se están haciendo un espacio destacado en la mayoría de conferencias y workshops[1] celebrados en los últimos años. En la comunidad de la ingeniería del software, se está viviendo con intensidad un debate abierto entre los partidarios de las metodologías tradicionales (referidas peyorativamente como "metodologías pesadas") y aquellos que apoyan las ideas emanadas del "manifiesto ágil" . La curiosidad que siente la mayor parte de los ingenieros del software, profesores, e incluso alumnos, sobre las metodologías ágiles hace prever una fuerte proyección industrial. Por un lado, para muchos equipos de desarrollo el uso de metodologías tradicionales les resulta muy lejano a su forma de trabajo actual considerando las dificultades de su introducción e inversión asociada en formación y herramientas. Por

[1] Denominados a los Talleres de trabajo.

otro, las características de los proyectos para los cuales las metodologías ágiles han sido especialmente pensadas se ajustan a un amplio rango de proyectos industriales para el desarrollo del software; aquellos en los cuales los equipos de desarrollo son pequeños, con plazos reducidos, requisitos volátiles, o basados en nuevas tecnologías [1].

2. HISTORIA

En las dos últimas décadas las notaciones de modelado y las herramientas han intentado ser la base fundamental para un desarrollo de software exitoso, sin embargo, estas metas no se alcanzaron, debido principalmente a que quedó postergada la metodología de desarrollo propiamente dicha. Las notaciones y herramientas no tienen un uso definido por sí mismas si no se proveen directivas para su aplicación. Así al comenzar esta década se ha prestado gran interés al desarrollo de metodologías que sean diferentes a las que hasta hace poco llevaban asociadas un marcado énfasis en el control del proceso mediante una rigurosa definición de roles, actividades y artefactos, incluyendo modelado y documentación detallada. Dicho esquema que se lo denomina “tradicional” es muy útil, efectivo y necesario para proyectos de desarrollo de software grandes respecto a tiempo y recursos.

Las metodologías ágiles surgieron debido a:

a) Una conciencia particularmente extrema de la crisis del software.

b) La responsabilidad que se les asigna a las grandes metodologías cuando se da la crisis.

c) El propósito de arribar a soluciones.

Diversos estudios revelaron que la práctica metodológica fuerte, con sus exigencias de planeamiento y sus técnicas de control, en muchos casos no brindaba resultados que estuvieran a la altura de sus costos en tiempo, complejidad y dinero. Investigaciones como la de Joe Nandhakumar y David Avison, en un trabajo de campo sobre “la ficción del desarrollo metodológico”, denunciaban que las metodologías clásicas de sistemas de información “se tratan inicialmente como una ficción necesaria para presentar una imagen”, pero dichas metodologías son demasiado ideales, rígidas y mecanicistas para ser aplicadas tal cual se

proponen. Duane Truex, Richard Baskerville y Julie Travis toman una posición aún más extrema y aseguran que es posible que los métodos tradicionales sean “completamente ideales inalcanzables y „hombres de paja“ hipotéticos que llegan a ser una guía normativa en situaciones de desarrollo utópicas”.

Se ha llegado a concluir que las metodologías estándares se basan en objetivos pasados de moda e incorrectos y que la obsesión de los ingenieros con los métodos puede llegar a coartar la iniciativa de una adecuada implementación, tanto a nivel de sistemas como en el plano de los negocios.

“Ágil”, denota la cualidad de ser ágil, facilidad para el movimiento, actividad y destreza en el movimiento. Durante algún tiempo se conocían como las metodologías ligeras, pero el término aceptado ahora es metodologías ágiles.

Estructuralmente, las Metodologías ágiles se asemejan a las metodologías clásicas para el desarrollo rápido de aplicaciones (RAD) y a otros modelos iterativos, pero su énfasis es más distintivo y su combinación de ideas es única.

En una reunión celebrada en febrero de 2001 en Utah-EEUU, nace el término “ágil” aplicado al desarrollo del software. En esta reunión participan un grupo de 17 expertos de la industria del software, incluyendo algunos de los creadores o impulsores de metodologías de software. Su objetivo fue esbozar los valores y principios que deberían permitir 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 del software tradicional, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas. Varias de las denominadas metodologías ágiles ya estaban siendo utilizadas con éxito en proyectos reales, pero les faltaba una mayor difusión y reconocimiento [1].
Tras esta reunión se creó *The Agile Alliance*[2], 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. El punto de partida es fue el Manifiesto Ágil, un documento que resume la filosofía “ágil”.

[2] www.agilealliance.com

Los firmantes del manifiesto ágil fueron: Kent Beck (XP), Mike Beedle, Arie van Bennekum (DSDM), Alistair Cockburn (Crystal), Ward Cunnimgham (XP), Martin Fowler (XP), James Grenning (XP), Jim Highsmith (ASD), Andrew Hunt (Pragmatic rogramming), Ron Jeffries (XP), Jon Kern (FDD), Brian Marick, Robert C. Martin (XP), Steve Mellor, Ken Schwaber (Scrum), Jeff Sutherland (Scrum) y Dave Thomas (Pragmatic Programming) [7].

3. MANIFIESTO ÁGIL

El Manifiesto Ágil, describe los principios y valores sobre los cuales se fundamenta el modelado ágil, este manifiesto fue el resultado de una serie de reuniones en los que participaron distintos expertos y metodologistas.

Martin Fowler, describe de la siguiente manera su experiencia en dichas reuniones:

> *"Con tanta similitud entre estos métodos, sería justo un poco de interés en alguna forma de trabajo colaborativo. Los representantes de cada una de estas metodologías fueron invitados a un taller de dos días en Snowbird, Utah en febrero de 2001. Yo asistí sin muchas expectativas. Después de todo cuando se pone un manojo de metodologistas en la misma habitación, lo mejor que se puede esperar es algo de civismo.*
>
> *Lo que resultó me sorprendió. Todos estábamos conscientes del hecho de que había mucho en común, y este reconocimiento era mucho mayor que las diferencias entre los procesos. Además de un contacto útil entre los líderes de procesos, había también la idea de emitir una declaración conjunta - una llamada a las armas en favor de más procesos de software ágiles. (También estábamos de acuerdo en usar el término "ágil" para referirnos a nuestras ideas comunes.)*
>
> *El resultado es un* Manifiesto para el desarrollo de Software Ágil*, una declaración de los principios y valores comunes de los procesos ágiles. Hay también un deseo de colaborar más en el futuro, para animar más tanto a tecnólogos como a gente de negocios para usar y requerir acercamientos ágiles al desarrollo de software. Hay un*

artículo en una revista de desarrollo de software que es un comentario y una explicación del manifiesto.

El manifiesto fue sólo eso, una publicación que actuó como un punto de partida para aquellos que compartían estas ideas básicas. Uno de los frutos del esfuerzo fue la creación de un cuerpo más longevo, la Alianza Ágil. *La Alianza Ágil es una organización sin fines de lucro que busca promover el conocimiento y la discusión de todos los métodos ágiles. Muchos líderes agilistas que he mencionado aquí son miembros y líderes de la Alianza Ágil" [9].*

Este manifiesto, está definida por cuatro valores simples. Una buena manera de interpretar el manifiesto, es asumir que éste define preferencias, no alternativas.

a. Valores del Manifiesto Ágil

En el Manifiesto Ágil, se valora:

Al individuo y a las interacciones del equipo de desarrollo sobre el proceso y las herramientas. La gente es el principal factor de éxito de un proyecto software. Si se sigue un buen proceso de desarrollo, pero el equipo falla, el éxito no está asegurado; sin embargo, si el equipo funciona, es más fácil conseguir el objetivo final, aunque no se tenga un proceso bien definido. No se necesitan desarrolladores brillantes, sino desarrolladores que se adapten al trabajo de equipo. Así mismo las herramientas (compiladores, depuradores, etc.) son importantes para mejorar el rendimiento del equipo.

Desarrollar software que funciona más que conseguir una buena documentación. Aunque se parte de la base de que el software sin documentación es un desastre, la regla a seguir es *no producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante*. Estos documentos deben ser cortos y centrarse en lo fundamental.

La colaboración con el cliente más que la negociación de un contrato. Las características particulares del desarrollo de software hace que muchos proyectos hayan fracasado por intentar cumplir unos plazos y unos costos preestablecidos al inicio del mismo, según los requisitos que el cliente manifestaba en ese momento. Por ello se propone que exista una interacción constante entre el cliente y el equipo

de desarrollo. Esta colaboración entre ambos será la que marque el avance del proyecto y asegure su éxito.

Responder a los cambios más que seguir estrictamente un plan. La habilidad de responder a los cambios que puedan surgir a lo largo del proyecto (cambios en los requisitos, en la tecnología, en los equipos, etc.) determinan también el éxito o fracaso del mismo. Por lo tanto la planificación no debe ser estricta puesto que hay muchas variables en juego, debe ser flexible para poder adaptarse a los cambios que puedan surgir. Una buena estrategia es hacer planificaciones detalladas para unas pocas semanas y planificaciones mucho más abiertas para unos pocos meses.

b. Principios del Manifiesto Ágil

Los valores anteriormente citados, dan lugar a los principios del *Manifiesto Ágil*; características que diferencian un proceso ágil de uno tradicional. Los dos primeros son generales y resumen gran parte del espíritu ágil [4]:

I. *La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.* Un proceso es ágil si a las pocas semanas de empezar ya entrega software que funcione aunque sea rudimentario.

II. *Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.* Este principio es una actitud que deben adoptar los miembros del equipo de desarrollo. Los cambios en los requisitos permite aprender más y logra mayor satisfacción al cliente. La estructura del software debe ser flexible para poder incorporar los cambios sin demasiado costo añadido. El paradigma orientado a objetos puede ayudar a conseguir esta flexibilidad.

Los siguientes, son principios que tienen que ver directamente con el proceso de desarrollo de software:

III. *Entregar frecuentemente producto software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.* Las

entregas al cliente se insiste en que sean software, no planificaciones, ni documentaciones de análisis o diseño.

IV. *La gente del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.* El proceso de desarrollo necesita ser guiado por el cliente, por lo que la interacción con el equipo es muy frecuente.

V. *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.*

VI. *El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.*

VII. *El software que funciona es la medida principal de progreso.* El estado de un proyecto no viene dado por la documentación generada o por la fase en la que se encuentre, sino por el código generado y el funcionamiento.

VIII. *Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.* No se trata de desarrollar lo más rápido posible, sino de mantener el ritmo de desarrollo durante toda la duración del proyecto, asegurando en todo momento que la calidad de lo producido es máxima.

IX. *La atención continua a la calidad técnica y al buen diseño mejora la agilidad.* Producir código claro y robusto es la clave para avanzar más rápidamente en el proyecto.

X. *La simplicidad es esencial*. Si el código producido es simple y de alta calidad será más sencillo adaptarlo a los cambios que puedan surgir.

XI. *Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.* Todo el equipo es informado de las responsabilidades y éstas recaen sobre todos sus miembros. Es el propio equipo el que decide la mejor forma de organizarse, de acuerdo a los objetivos que se persigan.

XII. *En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más práctico y según esto ajustar su comportamiento.* Puesto que el entorno está cambiando continuamente, el equipo también debe ajustarse al nuevo escenario de forma continua.

4. METODOLOGÍAS ÁGILES REPRESENTATIVAS

Los métodos ágiles nacieron con el espíritu de respuesta revolucionario a la rigidez de los modelos de procesos tradicionales. Varias metodologías encajan bajo el estandarte de ágil, mientras todas ellas comparten muchas características, también hay algunas diferencias significativas. Las siguientes, son las metodologías ágiles más representativas:

a. Scrum

Desarrollada por Ken Schwaber, Jeff Sutherland y Mike Beedle. Define un marco para la gestión de proyectos, que se ha utilizado con éxito durante los últimos 10 años.

Está especialmente indicada para proyectos con un rápido cambio de requisitos. Sus principales características se pueden resumir en dos. El desarrollo de software se realiza mediante iteraciones, denominadas sprints[3], con una duración de 30 días. El resultado de cada sprint es un incremento ejecutable que se muestra al cliente. La segunda característica importante son las reuniones a lo largo del proyecto, entre ellas destaca la reunión diaria de 15 minutos del equipo de desarrollo para coordinación e integración.

La metodología Scrum también se convirtió en la década del año 1980 y la década del año1990, sobre todo con los círculos del desarrollo orientado a objetos, en metodologías altamente iterativas para el desarrollo. Los autores más significativos son Ken Schwaber, Jeff Sutherland, y Mike Beedle. Scrum se concentra en los aspectos de la gerencia para el desarrollo del software, dividiendo el desarrollo en treinta iteraciones y aplicando más cerca la supervisión y el control con reuniones diarias del scrum. Pone mucho menos énfasis en prácticas de la ingeniería y mucha gente combina el acercamiento de la gerencia de proyecto con prácticas de ingeniería de la programación extrema.

[3] Técnicamente referida al tiempo de duración de cada iteración.

Según la Microsoft SCRUM es, después de XP, la metodología ágil mejor conocida y la que muchos otros métodos ágiles recomiendan como complemento. La primera referencia a la palabra “scrum” en la literatura aparece en un artículo de Hirotaka Takeuchi e Ikujiro Nonaka, “The New Product Development Game” en el que se presentaron las mejores prácticas de empresas innovadoras de Japón que siempre resultaban ser adaptativas, rápidas y con capacidad de auto-organización.

La palabra Scrum procede de la terminología del juego de rugby, donde designa al acto de preparar el avance del equipo en unidad pasando la pelota a uno y otro jugador (aunque hay otras acepciones en circulación). Igual que el juego, Scrum es adaptativo, ágil, auto-organizante y con pocos tiempos muertos [7].

b. Metodologías Crystal

La familia de las metodologías Crystal, creada por Alistair Cockburn, fue diseñada como acercamiento de “diseño-humano” al desarrollo del software. Estas metodologías se centran alrededor del equipo y de los individuos implicados en el proceso del software en vez de un acercamiento orientado a procesos.

Entre otras cosas, Crystal proporciona las sugerencias para la organización y el manejo del equipo previsto para reducir la burocracia y el papeleo mientras que aumenta el trabajo en equipo, la satisfacción personal, y la comunicación. Básicamente, los métodos de Crystal intentan encontrar la estructura y el proceso más simples y más compactos del equipo para una organización, haciendo así el proceso de desarrollo más rápido y eficiente. El desarrollo adaptativo del software se centra en la preparación de las organizaciones para un mundo donde los requisitos cambian y la flexibilidad es una necesidad. El proceso y la estructura de la organización deben ser bastante flexibles y adaptarse al cambio de direcciones que la evolución de un producto de software puede exigir.

c. Desarrollo de Software Adaptativo

Su impulsor es J. Highsmith. Sus principales características del Desarrollo de Software Adaptativo (ASD) son: iterativo, orientado a los componentes software más que a las tareas y tolerante a los cambios. El ciclo de vida que propone tiene tres fases esenciales: especulación, colaboración y aprendizaje. En la primera de ellas se inicia el proyecto y se planifican las características del software; en la segunda desarrollan las características y

finalmente en la tercera se revisa su calidad, y se entrega al cliente. La revisión de los componentes sirve para aprender de los errores y volver a iniciar el ciclo de desarrollo.

La estrategia entera se basa en el concepto de emergencia, una propiedad de los sistemas adaptativos complejos que describe la forma en que la interacción de las partes genera una propiedad que no puede ser explicada en función de los componentes individuales.

ASD presupone que las necesidades del cliente son siempre cambiantes. La iniciación de un proyecto involucra definir una misión para él, determinar las características y las fechas y descomponer el proyecto en una serie de pasos individuales, cada uno de los cuales puede abarcar entre cuatro y ocho semanas. Los pasos iniciales deben verificar el alcance del proyecto; los tardíos tienen que ver con el diseño de una arquitectura, la construcción del código, la ejecución de las pruebas finales y el despliegue.

d. Método para el Desarrollo de Sistemas Dinámicos

Define el marco para desarrollar un proceso de producción de software. Nace en 1994 con el objetivo de crear una metodología RAD unificada. Sus principales características son: es un proceso iterativo e incremental y el equipo de desarrollo y el usuario trabajan juntos. Propone cinco fases: estudio viabilidad, estudio del negocio, modelado funcional, diseño y construcción, y finalmente implementación. Las tres últimas son iterativas, además de contar con el proceso de realimentación a todas las fases.

Originado en los trabajos de Jennifer Stapleton, directora del DSDM Consortium, DSDM se ha convertido en la estructura de desarrollo rápido de aplicaciones (RAD) más popular de Gran Bretaña y se ha llegado a promover como el estándar de facto para el desarrollo de soluciones de negocios sujetas a márgenes de tiempo estrechos. Se calcula que uno de cada cinco desarrolladores en Gran Bretaña utiliza DSDM y que más de 500 empresas mayores la han adoptado.

e. Desarrollo Guiado por Rasgos (FDD)

El Desarrollo Guiado por Rasgos, traducción de las palabras inglésas Feature Driven Development (FDD), es un método ágil, iterativo y adaptativo. A diferencia de otros métodos ágiles, no cubre todo el ciclo de vida sino solamente las fases de diseño y construcción y se considera adecuado para proyectos mayores y de misión crítica. FDD es, además, marca registrada de la empresa, Nebulon Pty.

FDD no requiere un modelo específico de proceso y se complementa con otras metodologías. Enfatiza cuestiones de calidad y define claramente entregas tangibles y formas de evaluación del progreso. Se lo reportó por primera vez en un libro de Peter Coad, Eric Lefebvre y Jeff DeLuca, Java Modeling in Color with UML; luego fue desarrollado con amplitud en un proyecto mayor de desarrollo por DeLuca, Coad y Stephen Palmer. Este método consiste en cinco procesos secuenciales durante los cuales se diseña y construye el sistema. La parte iterativa soporta desarrollo ágil con rápidas adaptaciones a cambios en requerimientos y necesidades del negocio. Cada fase del proceso tiene un criterio de entrada, tareas, pruebas y un criterio de salida. Típicamente, la iteración de un rasgo toma entre una a tres semanas.

f. Desarrollo Continuo y Desarrollo Continuo de Software

Los aspectos esenciales del Desarrollo continuo son la relación participativa con el empleado y el trato que le brinda la compañía, así como una especificación de principios, disciplinas y métodos iterativos, adaptativos, auto-organizativos e interdependientes en un patrón de ciclos de corta duración que tiene algo más que un aire de familia con el patrón de procesos de las metodologías ágiles [4]. Existe unanimidad de intereses, consistencia de discurso y complementariedad entre las comunidades de Desarrollo continuo y desarrollo de software.

Mientras que otros métodos ágiles se concentran en el proceso de desarrollo, Charette 2001, sostenía que para ser verdaderamente ágil se debía conocer además el negocio de punta a punta. LD se inspira en doce valores centrados en estrategias de gestión [5]:

1. Satisfacer al cliente es la máxima prioridad.
2. Proporcionar siempre el mejor valor por la inversión.
3. El éxito depende de la activa participación del cliente.
4. Cada proyecto LD es un esfuerzo de equipo.
5. Todo se puede cambiar.
6. Soluciones globales de dominio, no puntos.
7. Completar, no construir.
8. Una solución al 80% hoy, en vez de una al 100% mañana.
9. El minimalismo es esencial.
10. La necesidad determina la tecnología.
11. El crecimiento del producto es el incremento de sus prestaciones, no de su tamaño.
12. Nunca forzar LD más allá de sus límites.

Dado que LD es más una filosofía de gestión que un proceso de desarrollo no hay mucho que decir del tamaño del equipo, la duración de las iteraciones, los roles o la naturaleza de sus etapas. Últimamente LD ha evolucionado como Lean Software Development (LSD); su figura de referencia es Mary Poppendieck [6].

5. PROGRAMACIÓN EXTREMA

A finales de la década del año 1990, dos grandes temas fueron tratados en las prácticas de desarrollo de ingeniería de software y en los métodos para su desarrollo, estos son: el diseño basado en patrones y los métodos ágiles, de los cuales algunos consideran a la Programación Extrema (XP) [4] una innovación extraordinaria, en cambio otros la definen como extremista, falaz o perniciosa para la salud de la profesión. Patrones y XP se convirtieron de inmediato en temas de discusión en la industria y en la Red[5]. Al tema de los Patrones el mundo académico lo está tratando como un asunto respetable desde hace un tiempo, en cambio XP recién ahora se está legitimando como un tópico serio de investigación. La mayor parte de los documentos han provenido y aún provienen de los practicantes, los críticos y los consultores que impulsan o rechazan sus postulados. Pero el crecimiento de los métodos ágiles y su penetración es vertiginoso y pocas veces visto ya que en tres o cuatro años, según el Cutter Consortium, el 50% de las empresas define como “ágiles” más de la mitad de los métodos empleados en sus proyectos [1].

En las dos últimas décadas las notaciones de modelado y las herramientas han intentado ser la base fundamental para un desarrollo de software exitoso, sin embargo, estas metas no se alcanzaron debidas principalmente a que quedó postergada la metodología de desarrollo propiamente dicha. Las notaciones y herramientas son nada en sí mismas si no se proveen directivas para su aplicación. Así al comenzar esta década se ha prestado gran interés al desarrollo de metodologías que sean diferentes a las que hasta hace poco llevaban asociadas un marcado énfasis en el control del proceso mediante una rigurosa definición de roles, actividades y artefactos, incluyendo modelado y documentación detallada. Dicho esquema se denomina “tradicional” es muy útil, efectivo y necesario para proyectos de desarrollo de software grandes respecto al tiempo y recursos.

[4] XP - Del anglosajón eXtreme Programming
[5] Referida a la red Internet.

XP, es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en el desarrollo del software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y proporcionando un buen clima de trabajo. XP se basa en la realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, además donde existe un alto riesgo técnico [2].

La programación extrema es la más representativa entre las metodologías. La tabla 1, muestra los detalles de la comparación entre esta y los demás métodos ágiles, descritos en el acápite anterior [2].

Tabla 1. Índice de agilidad de las metodologías ágiles [2]

CRITERIO DE COMPARACIÓN	CMM	ASD	Crystal	DSDM	FDD	LD	Scrum	XP
Sistema como algo cambiante	1	5	4	3	3	4	5	5
Colaboración	2	5	5	4	4	4	5	5
Características Metodología (CM)								
Resultados	2	5	5	4	4	4	5	5
Simplicidad	1	4	4	3	5	3	5	5
Adaptabilidad	2	5	5	3	3	4	4	3
Excelencia técnica	4	3	3	4	4	4	3	4
Prácticas de colaboración	2	5	5	4	3	3	4	5
Media CM	2,2	4,4	4,4	3,6	3,8	3,6	4,2	4,4
Media Total	1,7	4,8	4,5	3,6	3,6	3,9	4,7	4,8

(Los valores más altos representan una mayor agilidad)

De acuerdo a los resultados representados en la Tabla 1, se concluye que los tres primeros métodos con mayor agilidad y de acuerdo a escala, son: XP, SCRUM y ASD.

Las características de la programación extrema, son [1]:

a) *Comunicación:* Los programadores están en constante comunicación con los clientes para satisfacer sus requisitos y responder rápidamente a los cambios de los mismos. Muchos problemas que surgen en los proyectos se deben a que después de concretar los requisitos que debe cumplir el programa no hay una revisión de los mismos, pudiendo dejar olvidados puntos importantes.

b) *Simplicidad:* Codificación y diseños simples y claros. Muchos diseños son tan complicados que cuando se quieren ampliar resulta bastante dificultoso hacerlo o se tienen que desechar y partir de cero.

c) *Realimentación:* Mediante la realimentación se ofrece al cliente la posibilidad de conseguir un sistema apto a sus necesidades ya que se va mostrando el proyecto a tiempo para ser cambiado y retroceder a una fase anterior para rediseñarlo al gusto del cliente.

d) *Coraje:* Se debe tener coraje o valentía para cumplir los tres puntos anteriores; Hay que tener valor para comunicarse con el cliente y enfatizar algunos puntos, a pesar de que esto pueda dar la sensación de ignorancia por parte del programador, hay que tener coraje para mantener un diseño simple y no optar por el camino más fácil y por último hay que tener valor y confiar en que la realimentación sea efectiva.

a. Historias de Usuario

Es la técnica utilizada en XP para especificar los requisitos del software. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer. El tratamiento de las historias de usuario es bastante dinámica y flexible, en cualquier momento pueden romperse, reemplazarse por otras más específicas o generales. Existen varias sugerencias respecto a las plantillas de las tarjetas utilizadas para esta actividad, sin embargo no existe un consenso, en muchos casos solo se propone utilizar un nombre y una descripción del requisito escrita en breves líneas.

b. Roles

Los roles de acuerdo a la propuesta original de Beck (2000), son [3]:

a) *Cliente (Customer)*: Es quien escribe las historias de usuario y las pruebas funcionales para validar su implementación, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración.

b) *Programador (Programmer)*: El programador escribe las pruebas unitarias y produce el código del sistema.

c) *Encargado de pruebas (Tester)*: Ayuda a escribir al cliente las pruebas funcionales. Ejecuta las pruebas regularmente y difunde los resultados en el equipo.

d) *Encargado de seguimiento (Tracker)*: Su responsabilidad es verificar el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, también realiza el seguimiento del progreso de cada iteración y evalúa si los objetivos son alcanzables con las restricciones de tiempo y recursos disponibles. Determina cuándo es necesario realizar algún cambio para lograr los objetivos de cada iteración.

e) *Entrenador (Coach)*: Es responsable del proceso global. Es necesario que conozca a fondo el proceso XP para proveer guías a los miembros del equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.

f) *Consultor (Consultant)*: Es un miembro externo del equipo con un conocimiento específico en algún tema específico para el proyecto. Guía al equipo para resolver un problema específico.

g) *Gestor (Big Boss)*: Es el vehículo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es la de coordinación.

c. Proceso

El ciclo de vida ideal de XP consiste en seis fases:

Fase I: Exploración (Exploration Phase)

En esta fase, los clientes plantean a grandes rasgos las historias de usuario que son de interés para la primera entrega del producto. Al mismo tiempo el equipo de desarrollo se familiariza con las herramientas, tecnologías y prácticas que se utilizarán en el proyecto. La fase de exploración toma de pocas semanas a pocos meses, dependiendo del tamaño y familiaridad que tengan los programadores con la tecnología.

Fase II: Planificación de la entrega (Planning Phase)

El cliente establece la prioridad de cada historia de usuario y correspondientemente los programadores realizan estimación del esfuerzo necesario de cada una de ellas. Tanto cliente como equipo de desarrollo determinan el contenido de la primera entrega y se determina un cronograma en conjunto con el cliente. Una entrega debería obtenerse en no más de tres meses. Esta fase dura unos pocos días.

Fase III: Iteraciones (Iterations to Release Phase)

Esta fase incluye varias iteraciones sobre el sistema antes de ser entregado. El Plan de entrega está compuesto por iteraciones que como máximo duran tres semanas. En la primera iteración se puede intentar establecer una arquitectura del sistema que pueda ser utilizada durante el resto del proyecto. Esto se logra escogiendo las historias de usuario que contengan los elementos necesarios que den lugar a la creación de esta arquitectura, sin embargo, esto no siempre es posible ya que es el cliente quien decide qué historias se implementarán en cada iteración. Al final de la última iteración el sistema estará listo para entrar en producción.

Fase IV: Producción (Production)

Esta fase requiere de pruebas adicionales y revisiones de rendimiento antes de que el sistema sea trasladado al entorno del cliente. Al mismo tiempo, se toman decisiones

sobre la inclusión de nuevas características a la versión actual, debido a cambios durante esta fase.

Fase V: Mantenimiento (Maintenance)

Mientras la primera versión se encuentra en producción, el proyecto XP debe mantener el sistema en funcionamiento al mismo tiempo que desarrolla nuevas iteraciones. Para realizar esto se requiere de tareas de soporte para el cliente. De esta forma, la velocidad de desarrollo puede bajar después de la puesta en marcha del sistema en producción.

Fase VI: Muerte del Proyecto (Death)

Esta fase es aplicada cuando el cliente no tiene más historias de usuario para ser incluidas en el sistema. Se genera la documentación final del sistema y no se realizan cambios adicionales en la arquitectura.

La Figura 1, muestra el ciclo de vida de un proyecto XP:

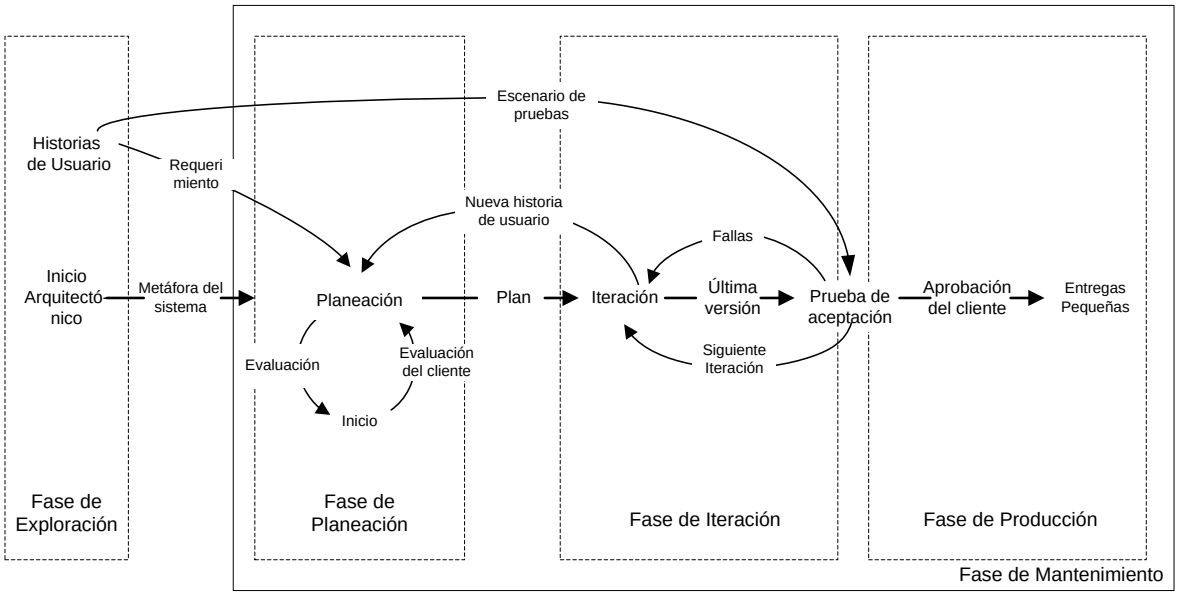

Figura 1. Ciclo de vida del Proyecto XP [3]

d. Prácticas

A través de las prácticas XP, se pretende reducir el costo del cambio a lo largo del proyecto. Este propósito se consigue gracias a las tecnologías disponibles para ayudar en el desarrollo del software y a la aplicación disciplinada de las prácticas. Letelier y Penadés (2004), describen las prácticas XP de la siguiente manera:

a) El juego de la planificación

Es un espacio frecuente de comunicación entre el cliente y los programadores. El equipo técnico realiza una estimación del esfuerzo requerido para la implementación de las historias de usuario y los clientes deciden sobre el ámbito y tiempo de las entregas y de cada iteración. Esta práctica se puede ilustrar como un juego, donde existen dos tipos de jugadores: Cliente y Programador. El cliente establece la prioridad de cada historia de usuario. Los programadores estiman el esfuerzo asociado a cada historia de usuario. Se ordenan las historias de usuario según prioridad y esfuerzo, y se define el contenido de la entrega y/o iteración, apostando por enfrentar la de más valor y riesgo cuanto antes. Este juego se realiza durante la planificación de la entrega, en la planificación de cada iteración y cuando sea necesario reconducir el proyecto.

b) Entregas pequeñas

La idea es producir rápidamente versiones del sistema que sean operativas, aunque obviamente no cuenten con toda la funcionalidad pretendida para el sistema. Una primera entrega no debería tardar más 3 meses, considerando proyectos de gran envergadura.

c) Metáfora

En XP no se enfatiza la definición temprana de una arquitectura estable para el sistema. Dicha arquitectura se asume evolutiva y los posibles inconvenientes que se generarían por no contar con ella explícitamente en el comienzo del proyecto se solventan con la existencia de una metáfora. El sistema es definido mediante una metáfora o un conjunto de metáforas compartidas por el cliente y el equipo de desarrollo. Una metáfora es una historia compartida que describe cómo debe funcionar el sistema.

d) Diseño simple

Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto, de tal manera que se consiga un diseño fácilmente entendible y operable que a la larga costará menos tiempo y esfuerzo desarrollar.

e) Pruebas

Se deben crear pruebas de funcionamiento con entornos de desarrollo antes de programar. Estas pruebas son obtenidas de la especificación de requisitos ya que en ella se especifican las pruebas que deben pasar las distintas funcionalidades del programa, procurando codificar pensando en las pruebas que debe pasar cada funcionalidad. Por otro lado, deben crear pruebas de aceptación usados por los clientes para comprobar que las distintas historias de usuario cumplen su cometido.

f) Refactorización

La refactorización es una actividad constante de reestructuración del código con el objeto de remover código duplicado, mejorar su legibilidad, simplificarlo y hacerlo flexible para facilitar los posteriores cambios. Esta actividad puede ser entendida también como una reprogramación, pues no se puede imponer todo en un inicio, pero en el transcurso del tiempo este diseño evoluciona conforme cambia la funcionalidad del sistema. Para mantener un diseño apropiado, es necesario realizar actividades de mantenimiento durante el ciclo de vida del proyecto. De hecho, este mantenimiento sobre el diseño es incluso más importante que el diseño inicial. Un concepto pobre al inicio puede ser corregido con esta actividad continua, pero sin ella, un buen diseño inicial perderá su consistencia.

g) Programación en parejas

Toda la producción de código del prototipo debe realizarse con trabajo en parejas de programadores. Las principales ventajas de introducir este estilo de programación son: muchos errores son detectados conforme vayan siendo introducidos en el código (inspecciones de código continuas), por consiguiente disminuye la tasa de errores del

producto final, los diseños son mejores y el tamaño del código menor (continua discusión de ideas de los programadores), los problemas de programación se resuelven más rápido, se posibilita la transferencia de conocimientos de programación entre los miembros del equipo, varias personas entienden las diferentes partes del sistema, los programadores conversan mejorando así el flujo de información y la dinámica del equipo.

Considerando que una de las características de la programación extrema es el reducido número de miembros que componen el grupo de desarrollo, un mínimo de dos miembros, se recurre a un programador adicional con quien se coordina las tareas de desarrollo de la presente propuesta.

h) Propiedad colectiva del código

Cualquier programador puede cambiar cualquier parte del código en cualquier momento. Esta práctica motiva a todos a contribuir con nuevas ideas en todos los segmentos del sistema, evitando a la vez que algún programador sea imprescindible para realizar cambios en algún segmento de código.

i) Integración Continua

Cada pieza de código es integrada en el sistema una vez que esté lista. Así, el sistema puede llegar a ser integrado y construido varias veces en un mismo día. Todas las pruebas son ejecutadas, de acuerdo al proceso de pruebas mencionado en los acápites anteriores, y tienen que ser aprobadas para que el nuevo código sea incorporado definitivamente. La integración continua a menudo reduce la fragmentación de los esfuerzos de los desarrolladores por falta de comunicación sobre lo que puede ser reutilizado o compartido.

j) 40 horas por semana

Se debe trabajar un máximo de 40 horas por semana. No se trabajan horas extras en dos semanas seguidas. Si esto ocurre, probablemente está ocurriendo un problema que debe corregirse. El trabajo extra desmotiva al equipo. Los proyectos que requieren trabajo extra para intentar cumplir con los plazos suelen al final ser entregados con retraso.

k) Cliente in-situ

El cliente tiene que estar presente y disponible todo el tiempo para el equipo, lo que no siempre es posible. Gran parte del éxito del proyecto XP se debe a que es el cliente quien conduce constantemente el trabajo hacia lo que aportará mayor valor de negocio y los programadores pueden resolver de manera inmediata cualquier duda asociada. La comunicación oral es más efectiva que la escrita, ya que esta última toma mucho tiempo en generarse y puede tener más riesgo de ser mal interpretada.

l) Estándares de Programación

Se enfatiza la comunicación de los programadores a través del código, con lo cual es indispensable que se sigan ciertos estándares de programación (del equipo, de la organización u otros estándares reconocidos para los lenguajes de programación utilizados). Los estándares de programación mantienen el código legible para los miembros del equipo, facilitando los cambios.

6. FASES DE DESARROLLO DEL PROYECTO XP

Los principios de la programación extrema, detallados en párrafos anteriores, son clasificados en cuatro fases que determinan las actividades para el desarrollo de un proyecto XP de manera organizada [8]:

6.1. Fase de planificación del proyecto

a) *Historias de Usuario*

El primer paso de cualquier proyecto que siga la metodología XP es definir las historias de usuario con el cliente. Las historias de usuario tienen la misma finalidad que los casos de uso en el paradigma orientado a objetos, pero con algunas diferencias: Constan de 3 ó 4 líneas escritas por el cliente en un lenguaje no técnico sin hacer mucho hincapié en los detalles; no se debe hablar ni de posibles algoritmos para su implementación ni de diseños de base de datos adecuados, etc. Son usadas para estimar tiempos de desarrollo del segmento de la aplicación que describen.

También se utilizan en la fase de pruebas, para verificar si el programa cumple con lo que especifica la historia de usuario. Cuando llega la hora de implementar una historia de usuario, el cliente y los desarrolladores se reúnen para concretar y detallar lo que tiene que hacer dicha historia. El tiempo de desarrollo ideal para una historia de usuario es entre 1 y 3 semanas.

b) *Planificación*

Plan de publicaciones: Después de tener ya definidas las historias de usuario es necesario crear un plan de publicaciones, en inglés "Release plan", donde se indiquen las historias de usuario que se crearán para cada versión del programa y las fechas en las que se publicarán estas versiones. Esta actividad es una planificación donde los desarrolladores y clientes establecen los tiempos de implementación ideales de las historias de usuario, la prioridad con la que serán implementadas y las historias que serán implementadas en cada versión del programa. Después de de planificar deben estar claros los siguientes cuatro factores: los objetivos que se deben cumplir (que son principalmente las historias por desarrollar en cada versión), el tiempo que tardarán en desarrollarse y publicarse las versiones del programa, el número de personas que trabajarán en el desarrollo y cómo se evaluará la calidad del trabajo realizado.

c) *Iteraciones*

Todo proyecto que siga la metodología XP debe dividirse en iteraciones de aproximadamente 3 semanas de duración. Al comienzo de cada iteración los clientes deben seleccionar las historias de usuario definidas en la fase de planificación, que serán implementadas. También se seleccionan las historias de usuario que no pasaron la prueba de aceptación que se realizó al terminar la iteración anterior. Estas historias de usuario son divididas en tareas de entre 1 y 3 días de duración que se asignarán a los programadores.

d) *Velocidad del proyecto*

La velocidad del proyecto es una medida que representa la rapidez con la que se desarrolla el proyecto; estimarla es muy sencillo: basta con contar el número de historias de usuario que se pueden implementar en una iteración; de esta forma, se sabrá el cupo de historias que se pueden desarrollar en las distintas iteraciones. Usando la velocidad del proyecto se controlará que todas las tareas se puedan desarrollar en el tiempo del que dispone la iteración. Es conveniente reevaluar esta medida cada 3 ó 4 iteraciones y si se aprecia que no es adecuada hay que negociar con el cliente una nueva planificación.

e) *Programación en pareja*

Este principio correspondiente a la fase de planificación, recomienda la programación en parejas pues incrementa la productividad y la calidad del software desarrollado. El trabajo en pareja involucra a dos programadores trabajando en el mismo equipo; mientras uno codifica haciendo hincapié en la calidad de la función o método que está implementando, el otro analiza si ese método o función es adecuado y está bien diseñado. De esta forma se consigue un código y diseño con gran calidad.

f) *Reuniones diarias*

Los desarrolladores se reúnen diariamente y exponen sus problemas, soluciones e ideas de forma conjunta. Las reuniones tienen que ser fluidas y todos los miembros del grupo de desarrollo, incluyendo al cliente, deben tener voz y voto.

6.2. Fase de diseño

a) *Diseños simples*

La metodología XP sugiere conseguir diseños simples y sencillos. Hay que procurar hacerlo todo lo menos complicado posible para conseguir un diseño fácilmente entendible e implementable que a la larga costará menos tiempo y esfuerzo desarrollar.

b) *Glosarios de términos*

Se recurre a glosarios de términos y una correcta especificación de los nombres de métodos y clases que ayudan a comprender el diseño y facilitar posteriores ampliaciones y la reusabilidad del código.

c) *Riesgos*

Si surgen problemas potenciales durante el diseño XP, se sugiere utilizar una pareja de desarrolladores para que investiguen y reduzcan al máximo el riesgo que supone ese problema.

d) *Funcionalidad extra*

Nunca se debe añadir funcionalidad extra al programa aunque se piense que en un futuro será utilizada. Sólo el 10% de la misma es utilizada, lo que implica que el desarrollo de alguna funcionalidad extra es un desperdicio de tiempo y recursos.

e) *Refactorizar*

Refactorizar es mejorar y modificar la estructura y codificación de códigos ya creados sin alterar su funcionalidad. Refactorizar supone revisar de nuevo estos códigos para procurar optimizar su funcionamiento. Es muy común reusar códigos ya creados que contienen funcionalidades que no serán usadas y diseños obsoletos. Esto es un error porque puede generar código completamente inestable y muy mal diseñado; por este motivo, es necesario refactorizar cuando se va a utilizar código ya creado.

6.3. Fase de codificación

Como se mencionó antes, el cliente es una parte más del equipo de desarrollo; su presencia es indispensable en las distintas fases de XP. A la hora de codificar una historia de usuario su presencia es aún más necesaria. Los clientes son los que crean las historias de usuario y negocian los tiempos en los que serán implementadas. Antes del desarrollo de cada historia de usuario el cliente debe especificar detalladamente lo que ésta hará y también tendrá que

estar presente cuando se realicen las pruebas que verifiquen que la historia implementada cumplan la funcionalidad especificada.

La codificación debe hacerse en base a estándares de codificación ya creados. Programar bajo estándares mantiene el código consistente y facilita su comprensión y escalabilidad.

Crear pruebas para el funcionamiento de los distintos códigos implementados ayudará a desarrollar dicho código. Crear antes estas pruebas ayuda a saber qué es exactamente lo que tiene que hacer el código a implementar y permite conocer que una vez implementado pasará dichas pruebas sin problemas ya que dicho código ha sido diseñado para ese fin. Se puede dividir la funcionalidad que debe cumplir una tarea a programar en pequeñas unidades, de esta forma se crearán primero las pruebas para cada unidad y a continuación se desarrollará dicha unidad, así poco a poco se conseguirá un desarrollo que cumpla todos los requisitos especificados.

Como ya se explicó anteriormente, XP opta por la programación en parejas ya que permite la generación de código estable y con gran calidad.

XP sugiere un modelo de trabajo usando repositorios de código dónde las parejas de programadores en pocas horas los códigos implementados y corregidos junto a las pruebas que deben pasar. De esta forma el resto de programadores que necesiten códigos ajenos trabajarán siempre con las últimas versiones. Para mantener un código consistente, publicar un código en un repositorio es una acción exclusiva para cada pareja de programadores.

XP también propone un modelo de desarrollo colectivo en el que todos los programadores están implicados en todas las tareas; cualquiera puede modificar o ampliar una clase o método de otro programador si es necesario y subirla al repositorio de código. Permitir al resto de los programadores modificar códigos que no son suyos no supone ningún riesgo ya que para que un código pueda ser publicado en el repositorio tiene que pasar las pruebas de funcionamiento definidos para el mismo.

La optimización del código siempre se debe dejar para el final. Hay que hacer que funcione y que sea correcto, posteriormente se puede optimizar.

XP afirma que la mayoría de los proyectos que necesiten más tiempo extra que el planificado para ser finalizados no podrán ser terminados a tiempo haga lo que se haga, aunque se

añadan más desarrolladores y se incrementen los recursos. La solución que plantea XP es realizar una nueva planificación para concretar los nuevos tiempos de publicación y de velocidad del proyecto.

6.4. Fase de pruebas

Uno de los pilares de la metodología XP es el uso de pruebas para comprobar el funcionamiento del código que se vaya implementando.

El uso de las pruebas en XP es el siguiente:

- Crear las aplicaciones que realizarán las pruebas con un entorno de desarrollo específico para la prueba.

- Someter a prueba las distintas clases del sistema omitiendo los métodos más triviales.

Crear las pruebas que pasará el código antes de implementarlos; en el apartado anterior se explicó la importancia de crear las pruebas antes que el código.

Un punto importante es crear pruebas que no tengan ninguna dependencia del código que en un futuro evaluará. Se debe crear las pruebas abstrayéndose del futuro código, de esta forma se asegura la independencia de la prueba respecto al código que evalúa.

Las pruebas también deben ser subidas al repositorio de código acompañados del código que verifican. Ningún código puede ser publicado en el repositorio sin que haya pasado la prueba de funcionamiento, de esta forma, se asegura el uso colectivo del código.

El uso de las pruebas es adecuado para observar la refactorización. Las pruebas permiten verificar que un cambio en la estructura del código no tiene porqué cambiar su funcionamiento.

- Pruebas de aceptación. Las pruebas mencionadas anteriormente sirven para evaluar las distintas tareas en las que ha sido dividida una historia de usuario. Para asegurar el funcionamiento final de una determinada historia de usuario se deben crear "Pruebas de aceptación"; estas pruebas son creadas y usadas por los clientes para comprobar que las distintas historias de usuario cumplen su cometido.

7. COMENTARIO

Las primeras décadas del desarrollo de software se caracterizaron por la ausencia de la aplicación de métodos para el desarrollo del producto que garantice su calidad, esta situación permitió entrever la enorme deficiencia en ellos que para mantenerse en vida recurrían a soluciones ad hoc que no eran más que paliativos a las frecuentes fallas. Los requerimientos, pocas veces cumplidos, eran otro elemento disconforme durante aquel tiempo. Las herramientas y los modelos parecían ser la solución a los problemas, aún así la situación era insostenible. No fue sino hasta los años 80 cuando se dio la aparición de la ingeniería del software como un conjunto de métodos y técnicas para desarrollar y mantener software de calidad. No obstante, con el pasar del tiempo se ha observado que incluso los métodos tradicionales de desarrollo del software se ven limitados ante requerimientos cambiantes y los reducidos tiempos de desarrollo que exigen los proyectos en la actualidad. Es entonces, cuando se da lugar a las llamadas metodologías ágiles, que a partir de valores y principios pretender dar soluciones a limitaciones como las que enfrentan los métodos usuales.

Cabe reconocer que las metodologías ágiles no son la panacea de las soluciones a todos los problemas de la ingeniería del software, sin embargo es innegable que se acomodan a las exigencias de desarrollo cuyos requerimientos suelen ser incompletos, variables y los tiempos de entrega bastante reducidos. Mediante las metodologías ágiles, además, es posible reducir las fallas, cumplir con los costos y el tiempo de desarrollo gracias a las iteraciones cortas que permiten presentar una nueva versión del producto software al finalizar cada una de ellas.

Por lo observado y descrito en el presente trabajo, se entiende que las metodologías ágiles permitieron cambiar el enfoque tradicional de las metodologías clásicas de desarrollo del producto software por métodos que permiten incluir mayor flexibilidad, simplicidad y adaptabilidad, lo que permite cumplir con los requerimientos del cliente satisfactoriamente. Estos métodos circunscriben mayor énfasis a los miembros del equipo de desarrollo incluyendo al cliente como parte del equipo, lo que garantiza el cumplimiento de los requisitos.

8. BIBLIOGRAFÍA

[1] Fowler M. The New Methodology, Chief Scientist, Thought Works. 2003.

[2] Letelier P, Penadés M. *Metodologías ágiles para el desarrollo de software: eXtreme Programming (XP)*. 2004; Universidad Politécnica de Valencia. Valencia.

[3] Ambler, S. *Agile Modeling – Effective Practices for eXtreme Programming and the Unified Process.* New York: John Wiley & Sons, Inc.; 2002.

[4] Beck, K. *Extreme Programming Explained Embrace Change,* Traducido al español como "*Una explicación de la programación extrema - Aceptar el cambio"*. New York: Addison Wesley; 2000.

[5] Jim Highsmith. Agile Software Development Ecosystems. Boston, Addison Wesley, 2002.

[7] Rising, L, and N. Janoff, The Scrum Software Development Process for Small Teams, IEEE Software, Julio/Agosto 2000, páginas 26-32.

Referencias Electrónicas

[6] Mary Poppendieck. "Lean Programming". [en línea] 2001 [fecha de acceso 27 de mayo de 2006]. URL disponible en:

http://www.agilealliance.org/articles/articles/LeanProgramming.htm.

[8] Leyva J. *Fundamentos de XP.* [en línea] 2005 [fecha de acceso 26 de septiembre de 2005]. URL disponible en: http:// juanleyva.metricasweb.com/

[9] Flowler M. *Is Design Dead?.* [en línea] 2003 [fecha de acceso 26 septiembre de 2005]. URL disponible en. http://www.martinfowler.com/articles/designDead.html