Universidad Nacional de Trujillo

Facultad de Ciencias Físicas y Matemáticas

Escuela de Informática

Metodologías Ágiles

Amaro Calderón, Sarah Dámaris
Valverde Rebaza. Jorge Carlos

Trujillo – Perú
2007

Índice

Metodologías Ágiles

1. Introducción

Para asegurar el éxito durante el desarrollo de software no es suficiente contar con notaciones de modelado y herramientas, hace falta un elemento importante: *la metodología de desarrollo*, la cual nos provee de una dirección a seguir para la correcta aplicación de los demás elementos.

Generalmente el proceso de desarrollo llevaba asociado 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 de 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 de las *buenas prácticas* de la Ingeniería del Software, asumiendo el riesgo que ello conlleva. En este contexto, las metodologías ágiles emergen como una posible respuesta para llenar ese vacío metodológico. Por 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.

2. Historia de los Procesos de Desarrollo

Uno de los grandes pasos dados en la industria del software fue aquel en que se plasmó el denominado modelo de cascada ya que sirvió como base para la formulación

del análisis estructurado, el cual fue uno de los precursores en el camino hacia la aplicación de prácticas estandarizadas dentro de la ingeniería de software. Este modelo surge como respuesta al modelo codificar y probar que era el que predominaba en la década de los sesenta. En esta época ya existían modelos iterativos e incrementales pero no eran disciplinados ni estaban formalizados.

A consecuencia de esta realidad, la idea de tener un modelo que ordenara el proceso de desarrollo y que parecía bastante sencillo de llevar a la práctica hizo que el modelo en cascada tuviese gran promoción. Este modelo se basaba en el desarrollo en forma de cascada ya que se requería de la finalización de la etapa anterior para empezar la siguiente. Esto degeneraba en un "congelamiento" temprano de los requerimientos, los cuales al presentar cambios requerían gran esfuerzo en retrabajo. Otra opción era no permitir cambio alguno a los requerimientos una vez que se iniciara el desarrollo lo que traía aparejado que el usuario no veía la aplicación hasta que ya estaba construida y una vez que interactuaba no cubría sus necesidades.

Asimismo, dadas las características inherentes del modelo, la fase de implementación del mismo requería el desarrollo de los módulos en forma independiente con las correspondientes pruebas unitarias, y en la siguiente fase, se realizaba la integración de los mismos. Esto traía grandes inconvenientes debido a que todo estaba probado en forma unitaria sin interacción con los demás módulos. Las sorpresas llegaban cuando se integraban estas piezas para formar la aplicación; lo cual inevitablemente desembocaba en un retraso del proyecto, sacrificando la calidad del mismo.

De esta forma y en forma bastante temprana en algunos casos, fueron surgiendo diversos procesos denominados iterativos que proponían lidiar con la impredictibilidad del software (subsanando muchas de las falencias del modelo en cascada) mitigando los riesgos en forma temprana. Los procesos iterativos de los cuales se desprenderían diversas instancias, como son el modelo iterativo e incremental, el modelo en espiral, el modelo basado en prototipo, el modelo SLCD, el MBASE, el RUP, etc.

Del modelo en espiral desarrollado por Barry Boehm [1] surgió una de las ideas fundamentales que las metodologías posteriores adoptarían: el temprano análisis de riesgos. El modelo en espiral, de carácter iterativo en sus primeras fases, plantea la necesidad de realizar al principio diversas iteraciones dirigidas a mitigar los riesgos más críticos relevados en el proyecto mediante la realización de prototipos o simulaciones de tipo desechables tendientes a probar algún concepto. Una vez que esos prototipos son validados se suceden iteraciones del tipo: determinar objetivos, evaluar, desarrollar, planear. Una vez que se tenía el diseño detallado y validado por el cliente, se implementaba el software siguiendo las etapas de un modelo en cascada. Esta es una falla importante del modelo ya que no se acomoda a la posibilidad de cambios una vez que se inicia la construcción. Todas las críticas que se le hacían al modelo en cascada se aplican a estas fases del modelo en espiral.

Fue el mismo Barry Boehm, quien en un artículo describe tres hitos críticos a ser utilizados en cualquier proyecto para poder planificar y controlar el progreso del mismo, dando visibilidad a los stakeholders. Estos hitos están relacionados con las etapas de avance que se van dando a lo largo de un proyecto de acuerdo a como ocurren las actividades de ingeniería (que componen los espirales del modelo en espiral) a las actividades de producción (que componen la construcción en cascada del software). Su impacto en la industria del software ha sido tan importante que uno de los procesos mas utilizados en la actualidad, el RUP, los incorpora. Estos hitos son:

- Objetivos del Ciclo de Vida
- Arquitectura del Ciclo de Vida
- Capacidad de Operación Inicial

El primer hito finaliza con la definición del alcance del software a construir, la identificación de los stakeholders, y el delineamiento del plan de desarrollo del sistema. El mismo ocurre al final de la fase de *Concepción* según el RUP.

El segundo hito finaliza con el delineamiento de la arquitectura del sistema, la resolución de todos los riesgos críticos del proyecto, y el refinamiento de los objetivos y el alcance del sistema. A partir de este hito, se comienza la construcción en forma masiva del sistema, llegándose a utilizar el máximo de recursos en el proyecto.

Asimismo, comienzan las fases más predecibles en cierta medida del desarrollo. El mismo corresponde al hito final de la fase de *Elaboración* según el RUP.

El último de los hitos corresponde a la entrega del primer release del software, que incorpora la funcionalidad definida en la correspondiente iteración. También se espera el tener material de entrenamiento, como un Manual del Usuario y Manual de Operaciones. Este hito se corresponde con el hito final de la fase de *Construcción* según el RUP. Lo que resultó interesante de estos hitos propuestos por Boehm es que los mismos son independientes del proceso de desarrollo elegido.

Como se ha mencionado, uno de los procesos con más influencia en la comunidad del software ha sido el RUP, el cual, es uno de los primeros procesos que es vendido como un producto. La idea de los creadores del RUP es que el mismo fuera un repositorio de todas las ideas vigentes y las denominadas *buenas prácticas* de la Ingeniería de Software. Sin embargo, al intentar abarcar proyectos de envergaduras tan dispares como podrían ser la construcción de un sistema de radares para portaviones versus la construcción de una registración de usuarios para una pequeña consultora, el RUP pierde la granularidad necesaria para describir en detalle uno de los factores más trascendentes de cualquier desarrollo de software: las personas.

Esta es una de las razones principales del advenimiento de las denominadas Metodologías Ágiles.

3. Las Metodologías Ágiles

A principios de la década del '90, surgió un enfoque que fue bastante revolucionario para su momento ya que iba en contra de toda creencia de que mediante procesos altamente definidos se iba a lograr obtener software en tiempo, costo y con la requerida calidad. El enfoque fue planteado por primera vez por Martin[2] y se dio a conocer en la comunidad de Ingeniería de Software con el nombre de RAD o Rapid Application Development. RAD consistía en un entorno de desarrollo altamente productivo, en el que participaban grupos pequeños de programadores utilizando herramientas que generaban código en forma automática tomando como entradas sintaxis de alto nivel. En general, se considera que

este fue uno de los primeros hitos en pos de la agilidad en los procesos de desarrollo.

La historia de las Metodologías Ágiles y su apreciación como tales en la comunidad de la Ingeniería de Software tiene sus inicios en la creación de una de las metodologías utilizada como arquetipo: XP - eXtreme Programming, que nace de la mente de Kent Beck, tomando ideas recopiladas junto a Ward Cunningham.

Durante 1996, Beck es llamado por Chrysler como asesor del proyecto Chrysler Comprehensive Compensation (C3) payroll system. Dada la poca calidad del sistema que se estaba desarrollando, Beck decide tirar todo el código y empezar de cero utilizando las prácticas que él había ido definiendo a lo largo del tiempo. El sistema, que administra la liquidación de aproximadamente 10.000 empleados, y consiste de 2.000 clases y 30.000 métodos, es puesto en operación en Mayo de 1997 casi respetando el calendario propuesto. Como consecuencia del éxito de dicho proyecto, Kent Beck dio origen a XP iniciando el movimiento de metodologías ágiles al que se anexarían otras metodologías surgidas mucho antes que el propio Beck fuera convocado por Chrysler.

Es así como que este tipo de metodologías fueron inicialmente llamadas "metodologías livianas", sin embargo, aun no contaban con una aprobación pues se le consideraba por muchos desarrolladores como meramente intuitiva. Luego, con el pasar de los años, en febrero de 2001, tras una reunión celebrada en Utah-EEUU, nace formalmente el término "ágil" aplicado al desarrollo de software. En esta misma reunión participan un grupo de 17 expertos de la industria del software, incluyendo algunos de los creadores o impulsores de metodologías de software con el objetivo de 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 de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

Tras esta reunión se creó *The 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. El punto de partida fue el Manifiesto Ágil, un documento que resume la filosofía "ágil".

3.1 El Manifiesto Ágil

El Manifiesto Ágil comienza enumerando los principales valores del desarrollo ágil. Según el Manifiesto se valora:

- **Al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas.** La gente es el principal factor de éxito de un proyecto software. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte automáticamente. Es mejor crear el equipo y que éste configure su propio entorno de desarrollo en base a sus necesidades.
- **Desarrollar software que funciona más que conseguir una buena documentación.** 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**. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha 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 los largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

Los valores anteriores inspiran los doce principios del manifiesto. Son características que diferencian un proceso ágil de uno tradicional. Los dos primeros principios son generales y resumen gran parte del espíritu ágil. El resto tienen que ver con el proceso a seguir y con el equipo de desarrollo, en cuanto metas a seguir y organización del mismo. Los principios son:

I. *La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.*
II. *Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.*
III. *Entregar frecuentemente software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.*
IV. *La gente del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.*
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.*
VIII. *Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.*
IX. *La atención continua a la calidad técnica y al buen diseño mejora la agilidad.*
X. *La simplicidad es esencial.*
XI. *Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.*
XII. *En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más efectivo, y según esto ajusta su comportamiento.*

3.2 Metodologías Ágiles versus Metodologías Tradicionales

La Tabla Nº 1 recoge esquemáticamente las principales diferencias de las metodologías ágiles con respecto a las tradicionales ("no ágiles"). Estas diferencias que afectan no sólo al proceso en sí, sino también al contexto del equipo así como a su organización.

Metodologías Ágiles	Metodologías Tradicionales
Basadas en heurísticas provenientes de prácticas de producción de código.	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo.
Especialmente preparadas para cambios durante el proyecto.	Cierta resistencia a los cambios.
Impuestas internamente (por el equipo).	Impuestas externamente.
Proceso menos controlado, con pocos principios.	Proceso mucho más controlado, con numerosas políticas/normas.
No existe contrato tradicional o al menos	Existe un contrato prefijado.

es bastante flexible.	
El cliente es parte del equipo de desarrollo.	El cliente interactúa con el equipo de desarrollo mediante reuniones.
Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio.	Grupos grandes y posiblemente distribuidos.
Pocos artefactos.	Más artefactos.
Pocos roles	Más roles.
Menos énfasis en la arquitectura del software.	La arquitectura del software es esencial y se expresa mediante modelos.

Tabla Nº1. Diferencias entre Metodologías Ágiles y no Ágiles.

Tener metodologías diferentes para aplicar de acuerdo con el proyecto que se desarrolle resulta una idea interesante. Estas metodologías pueden involucrar prácticas tanto de metodologías ágiles como de metodologías tradicionales. De esta manera podríamos tener una metodología para cada proyecto, la problemática sería definir cada una de las prácticas, y en el momento preciso definir parámetros para saber cual usar.

Es importante tener en cuenta que el uso de un método ágil no es para todos. Sin embargo, una de las principales ventajas de los métodos ágiles es su peso inicialmente ligero y por eso las personas que no estén acostumbradas a seguir procesos encuentran estas metodologías bastante agradables.

3.3 ¿Por qué usar Metodologías Ágiles?

Tomando las ideas de la Tabla Nº 1 podemos decir que las metodologías tradicionales presentan los siguientes problemas a la hora de abordar proyectos:

- Existen unas costosas fases previas de especificación de requisitos, análisis y diseño. La corrección durante el desarrollo de errores introducidos en estas fases será costosa, es decir, se pierde flexibilidad ante los cambios.
- El proceso de desarrollo está encorsetado por documentos firmados.

- El desarrollo es más lento. Es difícil para los desarrolladores entender un sistema complejo en su globalidad.

Las metodologías ágiles de desarrollo están especialmente indicadas en proyectos con requisitos poco definidos o cambiantes. Estas metodologías se aplican bien en equipos pequeños que resuelven problemas concretos, lo que no está reñido con su aplicación en el desarrollo de grandes sistemas, ya que una correcta modularización de los mismos es fundamental para su exitosa implantación. Dividir el trabajo en módulos abordables minimiza los fallos y el coste. Las metodologías ágiles presentan diversas ventajas, entre las que podemos destacar:

- Capacidad de respuesta a cambios de requisitos a lo largo del desarrollo
- Entrega continua y en plazos breves de software funcional
- Trabajo conjunto entre el cliente y el equipo de desarrollo
- Importancia de la simplicidad, eliminado el trabajo innecesario
- Atención continua a la excelencia técnica y al buen diseño
- Mejora continua de los procesos y el equipo de desarrollo

4. Metodologías Ágiles de Desarrollo de Software

En la Tabla Nº2 apreciamos las convergencias y divergencias en la definición de las metodologías ágiles más importantes.

Metodología	Acrónimo	Creación	Tipo de modelo	Característica
Adaptive Software Development	ASD	Highsmith 2000	Prácticas + ciclo de vida	Inspirado en sistemas adaptativos complejos
Agile Modeling	AM	Ambler 2002	Metodología basada en la práctica	Suministra modelado ágil a otros métodos
Cristal Methods	CM	Cockbum 1998	Familia de metodologías	Metodología ágil con énfasis en modelo de ciclos
Agile RUP	dX	Booch, Martin, Newkirk 1998	Framework/Disciplina	XP dado vuelta con artefactos RUP

Dynamic Solutions Delivery Model	DSDM	Stapleton 1997	Framework/modelo de ciclo de vida	Creado por 16 expertos en RAD
Evolutionary Project Management	EVO	Gilb 1976	Framework adaptativo	Primer método ágil existente
eXtreme Programming	XP	Beck 1999	Disciplina en prácticas de ingeniería	Método ágil radical
Feature-Driven Development	FDD	De Luca & Coad 1998 Palmer & Felsing 2002	Metodología	Método ágil de diseño y construcción
Lean Development	LD	Charette 2001, Mary y Tom Poppendieck	Forma de pensar – modelo logístico	Metodología basada en procesos productivos
Rapid Development	RAD	McConnell 1996	Survey de técnicas y modelos	Selección de *best practices*, no método
Microsoft Solutions Framework	MSF	Microsoft 1994	Lineamientos, disciplinas, prácticas	Framework de desarrollo de soluciones
Scrum	Scrum	Sutherland 1994 Schwaber 1995	Proceso – framework de management	Complemento de otros métodos, ágiles o no

Tabla N°2. Tabla de convergencias y divergencias entre las principales metodologías ágiles.

A continuación describiré brevemente algunas de las metodologías ágiles más destacadas hasta el momento.

4.1 XP- eXtreme Programming

XP es la primera metodología ágil y la que le dio conciencia al movimiento actual de metodologías ágiles. De la mano de Kent Beck, XP ha conformado un extenso grupo de seguidores en todo el mundo, disparando una gran cantidad de libros a los que dio comienzo el mismo Beck en [3]. Inclusive Addison-Wesley ha creado una serie de libros denominada *The XP Series*.

La imagen mental de Beck al crear XP [3] era la de perillas en un tablero de control. Cada perilla representaba una práctica que de su experiencia sabía que trabajaba bien. Entonces, Beck decidió girar todas las perillas al máximo para ver que ocurría. Así fue como dio inicio a XP.

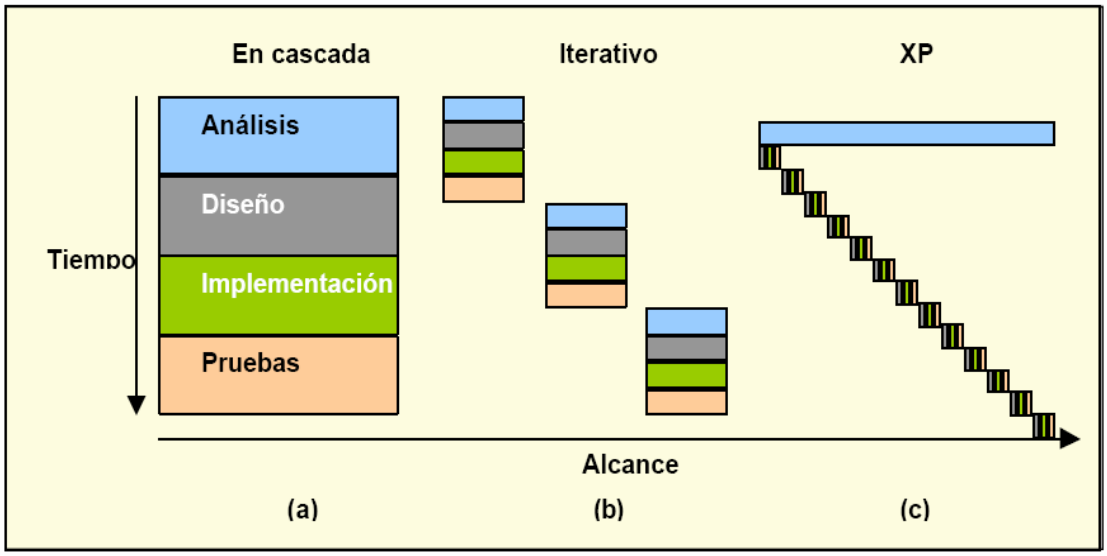

Figura N°1. Evolución de los largos ciclos de desarrollo en cascada (a) a ciclos iterativos más cortos (b) y a la mezcla que hace XP.

XP es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en el desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa en 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, y donde existe un alto riesgo técnico. A continuación presentaremos las características esenciales de XP organizadas en los tres apartados siguientes: historias de usuario, roles, proceso y prácticas.

A. Las Historias de Usuario

Son la técnica utilizada 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, sean requisitos funcionales o no funcionales. El tratamiento de

las historias de usuario es muy dinámico y flexible. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarla en unas semanas. Beck en su libro [3] presenta un ejemplo de ficha (*customer story and task card*) en la cual pueden reconocerse los siguientes contenidos: fecha, tipo de actividad (nueva, corrección, mejora), prueba funcional, número de historia, prioridad técnica y del cliente, referencia a otra historia previa, riesgo, estimación técnica, descripción, notas y una lista de seguimiento con la fecha, estado cosas por terminar y comentarios. A efectos de planificación, las historias pueden ser de una a tres semanas de tiempo de programación (para no superar el tamaño de una iteración). Las historias de usuario son descompuestas en tareas de programación (task card) y asignadas a los programadores para ser implementadas durante una iteración.

B. Roles XP

Los roles de acuerdo con la propuesta original de Beck son:

- **Programador.** El programador escribe las pruebas unitarias y produce el código del sistema.
- **Cliente.** Escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en aportar mayor valor al negocio.
- **Encargado de pruebas (*Tester*)**. Ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.
- **Encargado de seguimiento (*Tracker*)**. Proporciona realimentación al equipo. Verifica el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, para mejorar futuras estimaciones. Realiza el seguimiento del progreso de cada iteración.
- **Entrenador (*Coach*)**. Es responsable del proceso global. Debe proveer guías al equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.
- **Consultor**. Es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto, en el que puedan surgir problemas.
- **Gestor (*Big boss*)**. Es el vínculo entre clientes y programadores, ayuda a que

el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

C. Proceso XP

El ciclo de desarrollo consiste (a grandes rasgos) en los siguientes pasos:

1. El cliente define el valor de negocio a implementar.
2. El programador estima el esfuerzo necesario para su implementación.
3. El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.
4. El programador construye ese valor de negocio.
5. Vuelve al paso 1.

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden. No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse que el sistema tenga el mayor valor de negocio posible con cada iteración.

El ciclo de vida ideal de XP consiste de seis fases: Exploración, Planificación de la Entrega (*Release*), Iteraciones, Producción, Mantenimiento y Muerte del Proyecto.

D. Prácticas XP

La principal suposición que se realiza en XP es la posibilidad de disminuir la mítica curva exponencial del costo del cambio a lo largo del proyecto, lo suficiente para que el diseño evolutivo funcione. Esto se consigue gracias a las tecnologías disponibles para ayudar en el desarrollo de software y a la aplicación disciplinada de las siguientes prácticas.

- **El juego de la planificación**. Hay una comunicación frecuente 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.

- **Entregas pequeñas**. Producir rápidamente versiones del sistema que sean operativas, aunque no cuenten con toda la funcionalidad del sistema. Esta versión ya constituye un resultado de valor para el negocio. Una entrega no debería tardar más 3 meses.
- **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 debería funcionar el sistema (conjunto de nombres que actúen como vocabulario para hablar sobre el dominio del problema, ayudando a la nomenclatura de clases y métodos del sistema).
- **Diseño simple**. Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto.
- **Pruebas**. La producción de código está dirigida por las pruebas unitarias. Éstas son establecidas por el cliente antes de escribirse el código y son ejecutadas constantemente ante cada modificación del sistema.
- **Refactorización (*Refactoring*)**. Es una actividad constante de reestructuración del código con el objetivo de remover duplicación de código, mejorar su legibilidad, simplificarlo y hacerlo más flexible para facilitar los posteriores cambios. Se mejora la estructura interna del código sin alterar su comportamiento externo.
- **Programación en parejas**. Toda la producción de código debe realizarse con trabajo en parejas de programadores. Esto conlleva ventajas implícitas (menor tasa de errores, mejor diseño, mayor satisfacción de los programadores, etc.).
- **Propiedad colectiva del código**. Cualquier programador puede cambiar cualquier parte del código en cualquier momento.
- **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.
- **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.

- **Cliente in-situ**. El cliente tiene que estar presente y disponible todo el tiempo para el equipo. Éste es uno de los principales factores de éxito del proyecto XP. El cliente 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.
- **Estándares de programación**. XP enfatiza que la comunicación de los programadores es a través del código, con lo cual es indispensable que se sigan ciertos estándares de programación para mantener el código legible.

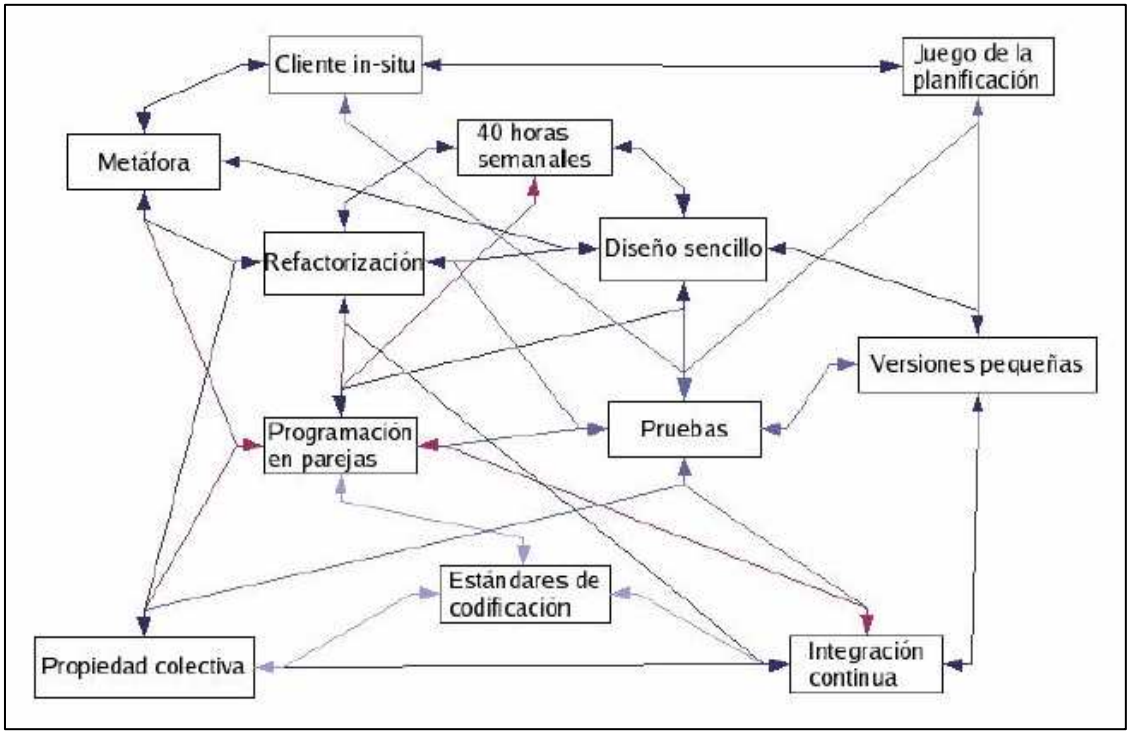

Figura N°2. Las prácticas se refuerzan entre sí.

El mayor beneficio de las prácticas se consigue con su aplicación conjunta y equilibrada puesto que se apoyan unas en otras. Esto se ilustra en la Figura N° 2, donde una línea entre dos prácticas significa que las dos prácticas se refuerzan entre sí. La mayoría de las prácticas propuestas por XP no son novedosas sino que en alguna forma ya habían sido propuestas en ingeniería del software e incluso demostrado su valor en la práctica El mérito de XP es integrarlas de una forma efectiva

y complementarlas con otras ideas desde la perspectiva del negocio, los valores humanos y el trabajo en equipo.

Si bien XP es la metodología ágil de más renombre en la actualidad, se diferencia de las demás metodologías que forman este grupo en un aspecto en particular: el alto nivel de disciplina de las personas que participan en el proyecto.

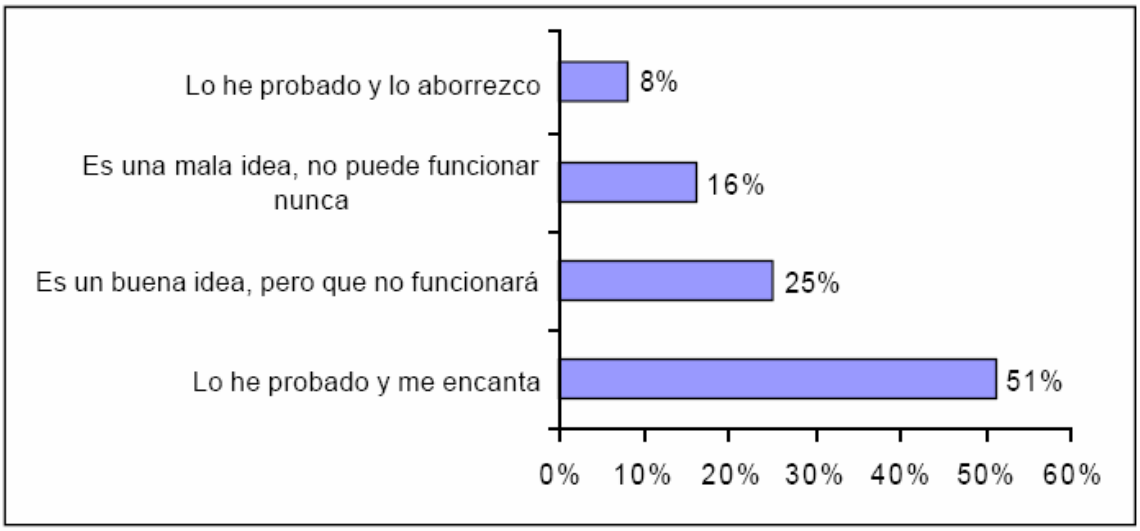

Figura N°3. Encuesta de IBM (octubre de 2000): ¿Qué opina de eXtreme Programming?

4.2 Scrum

Scrum define un proceso empírico, iterativo e incremental de desarrollo que intenta obtener ventajas respecto a los procesos definidos (cascada, espiral, prototipos, etc.) mediante la aceptación de la naturaleza caótica del desarrollo de software, y la utilización de prácticas tendientes a manejar la impredictibilidad y el riesgo a niveles aceptables. El mismo surge en 1986, de un artículo de la Harvard Business Review titulado "The New New Product Development Game" de Hirotaka Takeuchi e Ikujiro Nonaka, que introducía las mejores prácticas más utilizadas en 10 compañías japonesas altamente innovadoras. A partir de ahí y tomando referencias al juego de rugby, Ken Scwaber y Jeff Sutherland formalizan el proceso conocido como Scrum en el año 1995.

Scrum es un método iterativo e incremental que enfatiza prácticas y valores de project management por sobre las demás disciplinas del desarrollo. Al principio del proyecto se define el Product Backlog, que contiene todos los requerimientos funcionales y no funcionales que deberá satisfacer el sistema a construir. Los mismos estarán especificados de acuerdo a las convenciones de la organización ya sea mediante: features, casos de uso, diagramas de flujo de datos, incidentes, tareas, etc. El Product Backlog será definido durante reuniones de planeamiento con los stakeholders. A partir de ahí se definirán las iteraciones, conocidas como Sprint en la juerga de Scrum, en las que se irá evolucionando la aplicación evolutivamente. Cada Sprint tendrá su propio Sprint Backlog que será un subconjunto del Product Backlog con los requerimientos a ser construidos en el Sprint correspondiente. La duración recomendada del Sprint es de un mes.

Dentro de cada Sprint el Scrum Master (equivalente al Líder de Proyecto) llevará a cabo la gestión de la iteración, convocando diariamente al Scrum Daily Meeting que representa una reunión de avance diaria de no más de 15 minutos con el propósito de tener realimentación sobre las tareas de los recursos y los obstáculos que se presentan. Al final de cada Sprint, se realizará un Sprint Review para evaluar los artefactos construidos y comentar el planeamiento del próximo Sprint.

Como se puede observar en la Figura Nº4 la metodología resulta sencilla definiendo algunos roles y artefactos que contribuyen a tener un proceso que maximiza el feedback para mitigar cualquier riesgo que pueda presentarse.

A. Scrum aplicado al Desarrollo de Software

Aunque surgió como modelo para el desarrollo de productos tecnológicos, también se emplea en entornos que trabajan con requisitos inestables y que requieren rapidez y flexibilidad; situaciones frecuentes en el desarrollo de determinados sistemas de software.

Jeff Sutherland aplicó el modelo Scrum al desarrollo de software en 1993 en Easel Corporation (Empresa que en los macro-juegos de compras y fusiones se integraría en VMARK, luego en Informix y finalmente en Ascential Software Corporation). En 1996 lo presentó junto con Ken Schwaber como proceso formal,

también para gestión del desarrollo de software en OOPSLA 96. En el desarrollo de software Scrum está considerado como modelo ágil por la Agile Alliance

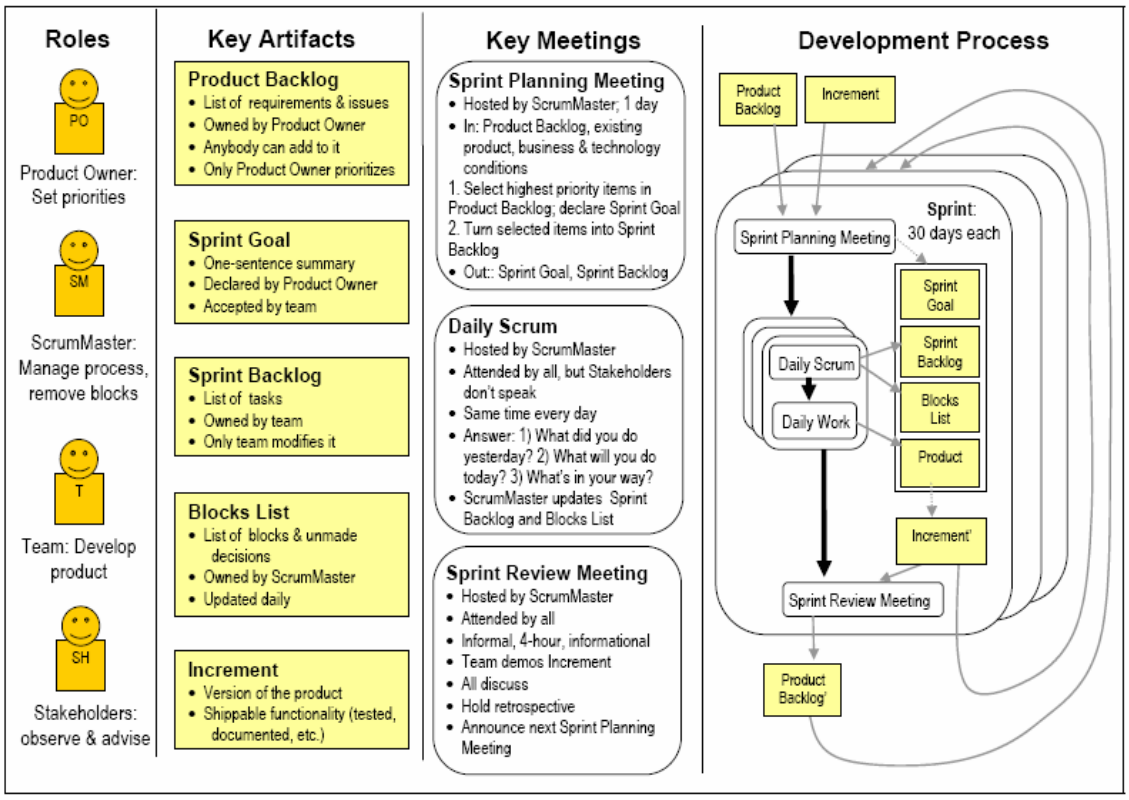

Figura N°4. Descripción de Roles, artefactos, reuniones y proceso de desarrollo de Scrum.

La intención de Scrum es la de maximizar la realimentación sobre el desarrollo pudiendo corregir problemas y mitigar riesgos de forma temprana. Su uso se está extendiendo cada vez más dentro de la comunidad de Metodologías Ágiles, siendo combinado con otras – como XP – para completar sus carencias. Cabe mencionar que Scrum no propone el uso de ninguna práctica de desarrollo en particular; sin embargo, es habitual emplearlo como un framework ágil de administración de proyectos que puede ser combinado con cualquiera de las metodologías mencionadas.

B. Ciclo de Vida de Scrum

El ciclo de vida de Scrum es el siguiente:

1. **Pre-Juego: Planeamiento.** El propósito es establecer la visión, definir expectativas y asegurarse la financiación. Las actividades son la escritura de la visión, el presupuesto, el registro de acumulación o retraso

(backlog) del producto inicial y los ítems estimados, así como la arquitectura de alto nivel, el diseño exploratorio y los prototipos. El registro de acumulación es de alto nivel de abstracción.

2. **Pre-Juego: Montaje (Staging).** El propósito es identificar más requerimientos y priorizar las tareas para la primera iteración. Las actividades son planificación, diseño exploratorio y prototipos.
3. **Juego o Desarrollo**. El propósito es implementar un sistema listo para entrega en una serie de iteraciones de treinta días llamadas "corridas" (sprints). Las actividades son un encuentro de planeamiento de corridas en cada iteración, la definición del registro de acumulación de corridas y los estimados, y encuentros diarios de Scrum.
4. **Pos-Juego: Liberación.** El propósito es el despliegue operacional. Las actividades, documentación, entrenamiento, mercadeo y venta.

Usualmente los registros de acumulación se llevan en planillas de cálculo comunes, antes que en una herramienta sofisticada de gestión de proyectos. Los elementos del registro pueden ser prestaciones del software, funciones, corrección de bugs, mejoras requeridas y actualizaciones de tecnología. Hay un registro total del producto y otro específico para cada corrida de 30 días. En la jerga de Scrum se llaman "paquetes" a los objetos o componentes que necesitan cambiarse en la siguiente iteración.

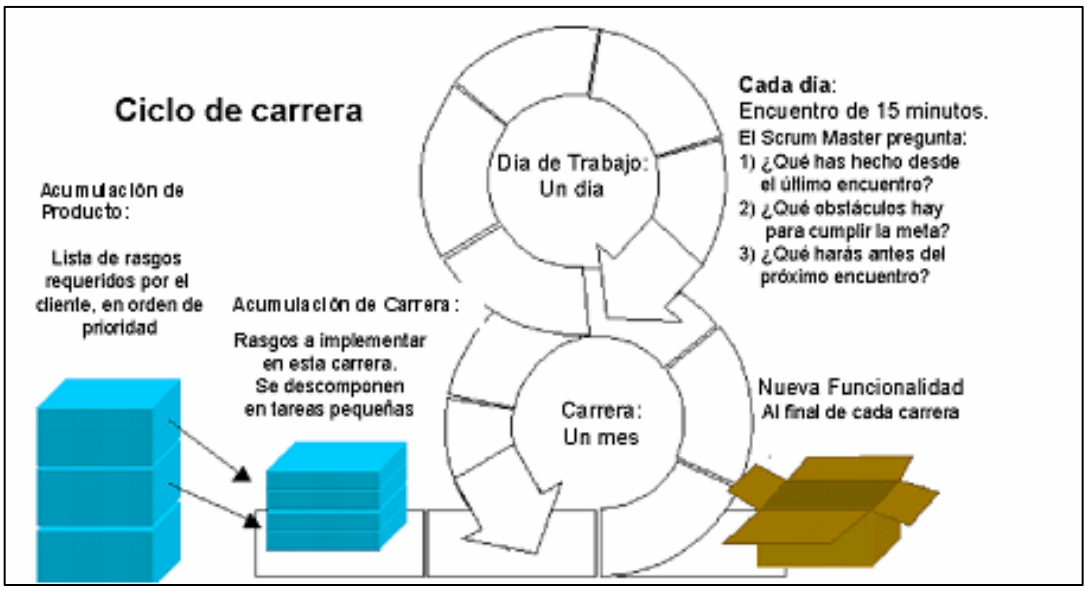

Figura N°5. Ciclo de Carrera o de Vida (Sprint) de Scrum.

La lista de Acumulación del Producto contiene todos los rasgos, tecnología, mejoras y lista de bugs que, a medida que se desenvuelven, constituyen las futuras entregas del producto. Los rasgos más urgentes merecerán mayor detalle, los que pueden esperar se tratarán de manera más sumaria. La lista se origina a partir de una variedad de fuentes. El grupo de mercadeo genera los rasgos y la función; la gente de ventas genera elementos que harán que el producto sea más competitivo; los de ingeniería aportarán paquetes que lo harán más robusto; el cliente ingresará debilidades o problemas que deberán resolverse. El propietario de la administración y el control del backlog en productos comerciales bien puede ser el product manager; para desarrollos in-house podría ser el project manager, o alguien designado por él. Se recomienda que una sola persona defina la prioridad de una tarea; si alguien tiene otra opinión, deberá convencer al responsable. Se estima que priorizar adecuadamente una lista de producto puede resultar dificultosa al principio del desarrollo, pero deviene más fácil con el correr del tiempo.

Al fin de cada iteración de treinta días hay una demostración a cargo del Scrum Master. Las presentaciones en PowerPoint están prohibidas. En los encuentros diarios, las gallinas deben estar fuera del círculo. Todos tienen que ser puntuales; si alguien llega tarde, se le cobra una multa que se destinará a obras de caridad. Es permitido usar artefactos de los métodos a los que Scrum acompañe, por ejemplo Listas de Riesgos si se utiliza UP, Planguage si el método es Evo, o los Planes de Proyecto sugeridos en la disciplina de Gestión de Proyectos de Microsoft Solutions Framework. No es legal, en cambio, el uso de instrumentos tales como diagramas PERT, porque éstos parten del supuesto de que las tareas de un proyecto se pueden identificar y ordenar; en Scrum el supuesto dominante es que el desarrollo es semi-caótico, cambiante y tiene demasiado ruido como para que se le aplique un proceso definido.

Algunos textos sobre Scrum establecen una arquitectura global en la fase de pre-juego; otros dicen que no hay una arquitectura global en Scrum, sino que la arquitectura y el diseño emanan de múltiples corridas. No hay una ingeniería del software prescripta para Scrum; cada quien puede escoger entonces las prácticas de automatización, inspección de código, prueba unitaria, análisis o programación en pares que le resulten adecuadas.

Como ya se ha mencionado antes, es muy habitual que Scrum se complemente con XP; en estos casos, Scrum suministra un marco de management basado en patrones organizacionales, mientras XP constituye la práctica de programación, usualmente orientada a objetos y con fuerte uso de patrones de diseño. Uno de los nombres que se utiliza para esta alianza es XP@Scrum. También son viables los híbridos con otras metodologías ágiles.

4.3 Crystal Clear

Alistair Cockburn es el propulsor detrás de la serie de metodologías Crystal. Las mismas presentan un enfoque ágil, con gran énfasis en la comunicación, y con cierta tolerancia que la hace ideal en los casos en que sea inaplicable la disciplina requerida por XP. Crystal "Clear" es la encarnación más ágil de la serie y de la que más documentación se dispone. La misma se define con mucho énfasis en la comunicación y de forma muy liviana en relación a los entregables. Crystal maneja iteraciones cortas con feedback frecuente por parte de los usuarios/clientes, minimizando de esta forma la necesidad de productos intermedios. Otra de las cuestiones planteadas es la necesidad de disponer de un usuario real aunque sea de forma part time para realizar validaciones sobre la Interfase del Usuario y para participar en la definición de los requerimientos funcionales y no funcionales del software.

Comparar el software con la ingeniería nos conduce a preguntarnos sobre "especificaciones" y "modelos" del software, sobre su completitud, corrección y vigencia. Esas preguntas son inconducentes, porque cuando pasa cierto tiempo no nos interesa que los modelos sean completos, que coincidan con el mundo "real" (sea ello lo que fuere) o que estén al día con la versión actual del lenguaje. Intentar que así sea es una pérdida de tiempo [4]. En contra de esa visión ingenieril a la manera de un Bertrand Meyer, Cockburn ha alternado diversas visiones despreocupadamente contradictorias que alternativamente lo condujeron a adoptar XP en el sentido más radical, a sinergizarse con DSDM o LSD, a concebir el desarrollo de software como una forma comunitaria de poesía o a elaborar su propia familia de Metodologías Crystal.

La familia Crystal dispone un código de color para marcar la complejidad de una metodología: cuanto más oscuro un color, más "pesado" es el método. Cuanto más crítico es un sistema, más rigor se requiere. El código cromático se aplica a una forma tabular elaborada por Cockburn que se usa en muchas metodologías ágiles para situar el rango de complejidad al cual se aplica una metodología. En la Figura Nº6 se muestra una evaluación de las pérdidas que puede ocasionar la falla de un sistema y el método requerido según este criterio. Los parámetros son Comodidad (C), Dinero Discrecional (D), Dinero Esencial (E) y Vidas (L). En otras palabras, la caída de un sistema que ocasione incomodidades indica que su nivel de criticalidad es C, mientras que si causa pérdidas de vidas su nivel es L. Los números del cuadro indican el número de personas afectadas a un proyecto.

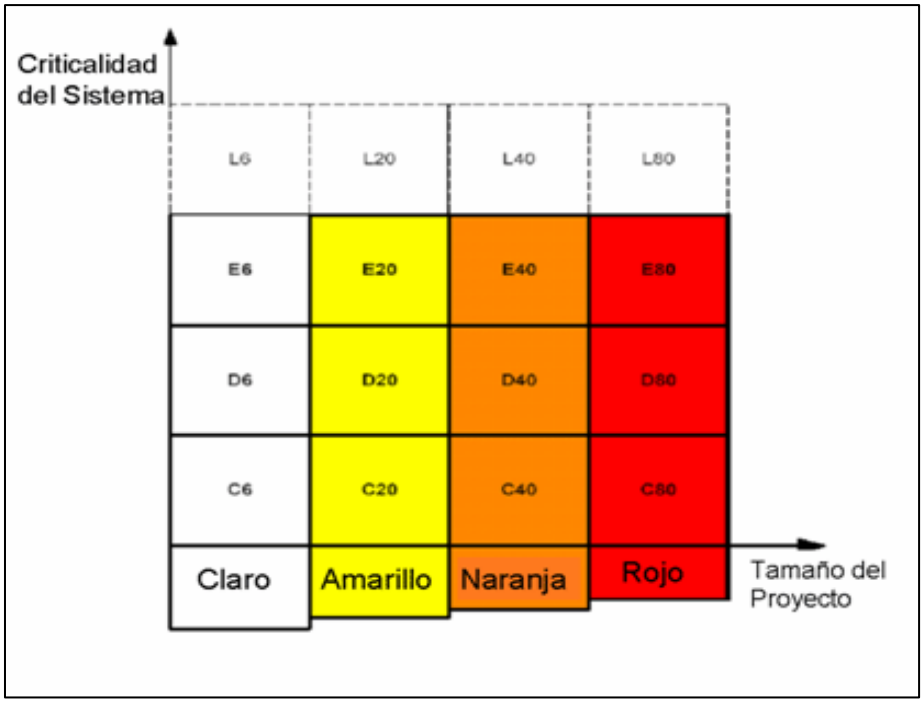

Figura N°6. Familia de Crystal Methods

Los métodos se llaman Crystal evocando las facetas de una gema: cada faceta es otra versión del proceso, y todas se sitúan en torno a un núcleo idéntico. Hay cuatro variantes de metodologías: Crystal Clear ("Claro como el cristal") para equipos de 8 o menos integrantes; Amarillo, para 8 a 20; Naranja, para 20 a 50; Rojo, para 50 a 100. Se promete seguir con Marrón, Azul y Violeta. La más exhaustivamente documentada es Crystal Clear (CC), la misma que puede ser usada en proyectos pequeños de categoría D6, aunque con alguna extensión se aplica también a niveles E8 a D10. El otro método

elaborado en profundidad es el Naranja, apto para proyectos de duración estimada en 2 años. Los otros dos aún se están desarrollando. Como casi todos los otros métodos, CC consiste en valores, técnicas y procesos.

Los siete valores o propiedades de Crystal Clear son:

1. **Entrega frecuente.** Consiste en entregar software a los clientes con frecuencia, no solamente en compilar el código. La frecuencia dependerá del proyecto, pero puede ser diaria, semanal, mensual o lo que fuere. La entrega puede hacerse sin despliegue, si es que se consigue algún usuario cortés o curioso que suministre feedback.
2. **Comunicación osmótica.** Todos juntos en el mismo cuarto. Una variante especial es disponer en la sala de un diseñador senior; eso se llama Experto al Alcance de la Oreja. Una reunión separada para que los concurrentes se concentren mejor es descripta como El Cono del Silencio.
3. **Mejora reflexiva.** Tomarse un pequeño tiempo (unas pocas horas por algunas semanas o una vez al mes) para pensar bien qué se está haciendo, cotejar notas, reflexionar, discutir.
4. **Seguridad personal.** Hablar cuando algo molesta: decirle amigablemente al manager que la agenda no es realista, o a un colega que su código necesita mejorarse, o que sería conveniente que se bañase más seguido. Esto es importante porque el equipo puede descubrir y reparar sus debilidades. No es provechoso encubrir los desacuerdos con gentileza y conciliación. Técnicamente, estas cuestiones se han caracterizado como una importante variable de confianza y han sido estudiadas con seriedad en la literatura.
5. **Foco.** Saber lo que se está haciendo y tener la tranquilidad y el tiempo para hacerlo. Lo primero debe venir de la comunicación sobre dirección y prioridades, típicamente con el Patrocinador Ejecutivo. Lo segundo, de un ambiente en que la gente no se vea compelida a hacer otras cosas incompatibles.
6. **Fácil acceso a usuarios expertos.** Una comunicación de Keil a la ACM demostró hace tiempo, sobre una amplia muestra estadística, la importancia del contacto directo con expertos en el desarrollo de un proyecto. No hay un dogma de vida o muerte sobre esto, como sí lo hay

en XP. Un encuentro semanal o semi-semanal con llamados telefónicos adicionales parece ser una buena pauta. Otra variante es que los programadores se entrenen para ser usuarios durante un tiempo. El equipo de desarrollo, de todas maneras, incluye un Experto en Negocios.

7. **Ambiente técnico con prueba automatizada, management de configuración e integración frecuente.** Microsoft estableció la idea de los builds cotidianos, y no es una mala práctica. Muchos equipos ágiles compilan e integran varias veces al día.

El proceso de Cristal Clear se basa en una exploración refinada de los inconvenientes de los modelos clásicos. Dice Cockburn que la mayoría de los modelos de proceso propuestos entre 1970 y 2000 se describían como secuencias de pasos. Aún cuando se recomendaran iteraciones e incrementos (que no hacían más que agregar confusión a la interpretación) los modelos parecían dictar un proceso en cascada, por más que los autores aseguraran que no era así. El problema con estos procesos es que realmente están describiendo un workflow requerido, un grafo de dependencia: el equipo no puede entregar un sistema hasta que está integrado y corre. No puede integrar y verificar hasta que el código no está escrito y corriendo. Y no puede diseñar y escribir el código hasta que se le dice cuáles son los requerimientos. Un grafo de dependencia se interpreta necesariamente en ese sentido, aunque no haya sido la intención original.

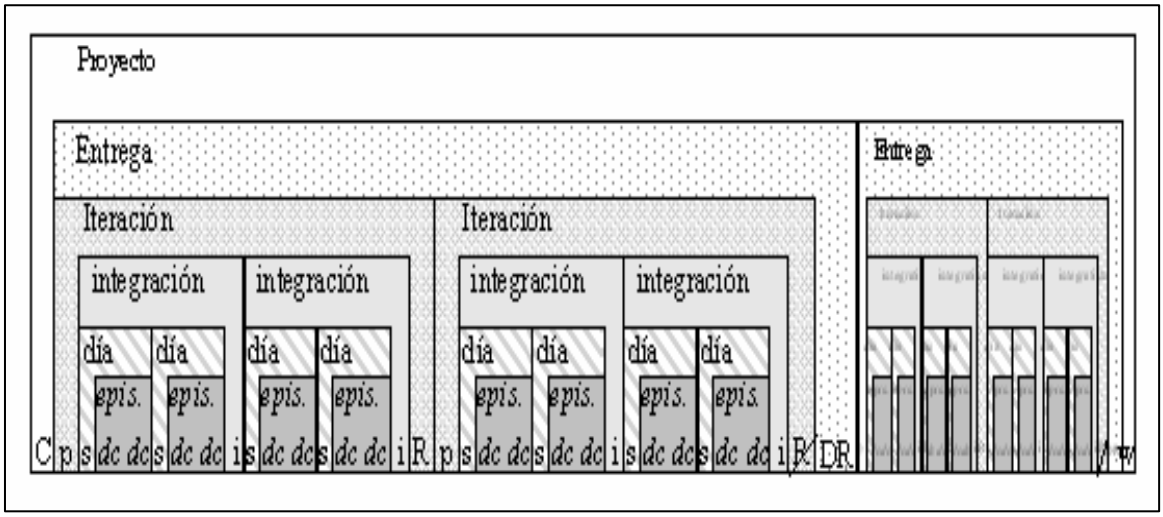

Figura N°7. Ciclos anidados de Crystal Clear

En lugar de esta interpretación lineal, Cristal Clear enfatiza el proceso como un conjunto de ciclos anidados. En la mayoría de los proyectos se perciben siete ciclos: (1) el proyecto, (2) el ciclo de entrega de una unidad, (3) la iteración (nótese que CC requiere múltiples entregas por proyecto pero no muchas iteraciones por entrega), (4) la semana laboral, (5) el período de integración, de 30 minutos a tres días, (6) el día de trabajo, (7) el episodio de desarrollo de una sección de código, de pocos minutos a pocas horas.

Los métodos Crystal no prescriben las prácticas de desarrollo, las herramientas o los productos que pueden usarse, pudiendo combinarse con otros métodos como Scrum, XP y Microsoft Solutions Framework. En un comentario Cockburn confiesa que cuando imaginó a Crystal Clear pensaba proporcionar un método ligero; comparado con XP, sin embargo, Cristal Clear resulto muy pesado, sin embargo es más fácil de aprender e implementar; a pesar de su jerga chocante XP es más disciplinado, piensa Cockburn; pero si un equipo ligero puede tolerar sus rigores, lo mejor será que se mude a XP.

4.4 DSDM – Dynamic Systems Development Method

DSDM es la única de las metodologías aquí planteadas surgida de un Consorcio, formado originalmente por 17 miembros fundadores en Enero de 1994. El objetivo del Consorcio era producir una metodología de dominio público que fuera independiente de las herramientas y que pudiera ser utilizado en proyectos de tipo RAD (Rapid Application Development). El Consorcio, tomando las best practices que se conocían en la industria y la experiencia traída por sus fundadores, liberó la primera versión de DSDM a principios de 1995. A partir de ese momento el método fue bien acogido por la industria, que empezó a utilizarlo y a capacitar a su personal en las prácticas y valores de DSDM. Debido a este éxito, el Consorcio comisionó al Presidente del Comité Técnico, Jennifer Stapleton, la creación de un libro que explorara la realidad de implementar el método.

Dado el enfoque hacia proyectos de características RAD esta metodología encuadra perfectamente en el movimiento de metodologías ágiles. La estructura

del método fue guiada por estos nueve principios:

1. El involucramiento del usuario es imperativo.
2. Los equipos de DSDM deben tener el poder de tomar decisiones.
3. El foco está puesto en la entrega frecuente de productos.
4. La conformidad con los propósitos del negocio es el criterio esencial para la aceptación de los entregables.
5. El desarrollo iterativo e incremental es necesario para converger hacia una correcta solución del negocio.
6. Todos los cambios durante el desarrollo son reversibles.
7. Los requerimientos están especificados a un alto nivel.
8. El testing es integrado a través del ciclo de vida.
9. Un enfoque colaborativo y cooperativo entre todos los interesados es esencial.

DSDM define cinco fases en la construcción de un sistema – ver Figura Nº8. Las mismas son: estudio de factibilidad, estudio del negocio, iteración del modelo funcional, iteración del diseño y construcción, implantación. El estudio de factibilidad es una pequeña fase que propone DSDM para determinar si la metodología se ajusta al proyecto en cuestión. Durante el estudio del negocio se involucra al cliente de forma temprana, para tratar de entender la operatoria que el sistema deberá automatizar. Este estudio sienta las bases para iniciar el desarrollo, definiendo las features de alto nivel que deberá contener el software. Posteriormente, se inician las iteraciones durante las cuales: se bajará a detalle los features identificados anteriormente, se realizará el diseño de los mismos, se construirán los componentes de software, y se implantará el sistema en producción previa aceptación del cliente.

Desde mediados de la década de 1990 hay abundantes estudios de casos, sobre todo en Gran Bretaña, y la adecuación de DSDM para desarrollo rápido está suficientemente probada. El equipo mínimo de DSDM es de dos personas y puede llegar a seis, pero puede haber varios equipos en un proyecto. El mínimo de dos personas involucra que un equipo consiste de un programador y un usuario. El máximo de seis es el valor que se encuentra en la práctica. DSDM se ha aplicado a

proyectos grandes y pequeños. La precondición para su uso en sistemas grandes es su partición en componentes que pueden ser desarrollados por equipos normales.

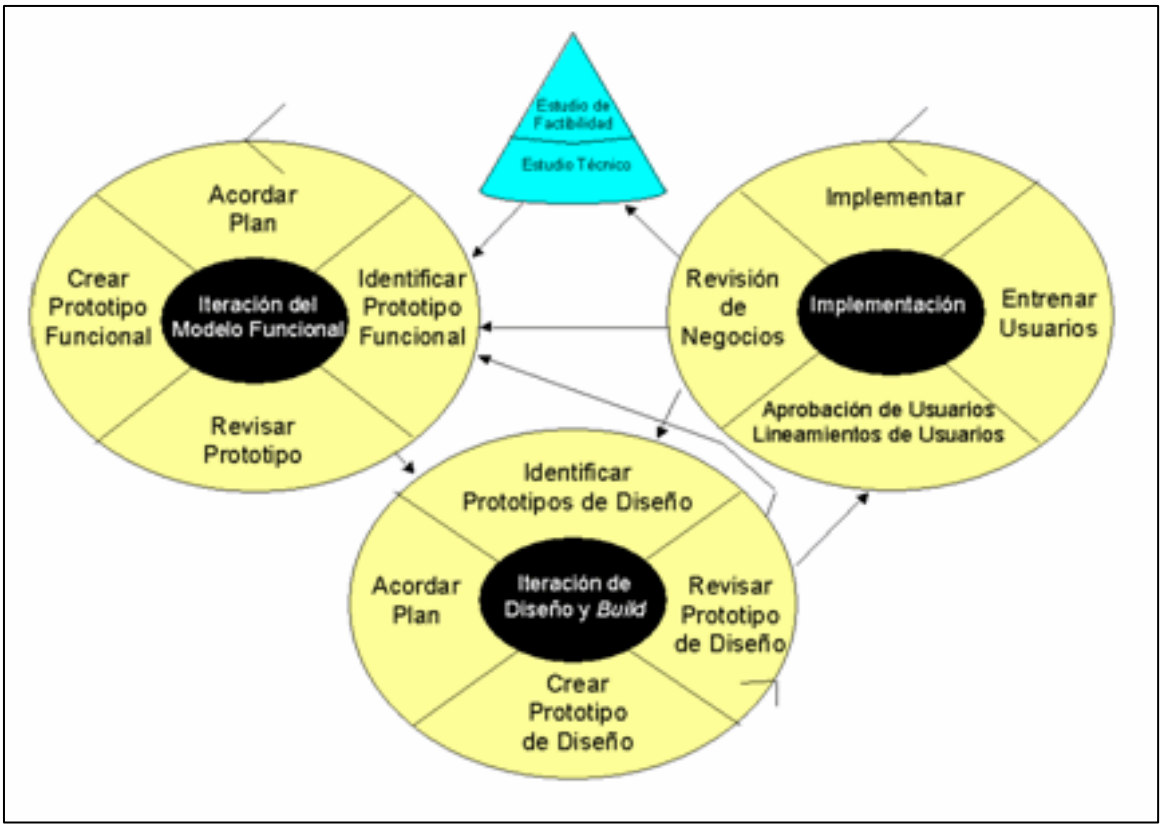

Figura N°8. Fases del Proceso de Desarrollo de DSDM

Se ha elaborado en particular la combinación de DSDM con XP y se ha llamado a esta mixtura EnterpriseXP, término acuñado por Mike Griffiths de Quadrus Developments . Se atribuye a Kent Beck haber afirmado que la comunidad de DSDM ha construido una imagen corporativa mejor que la del mundo XP y que sería conveniente aprender de esa experiencia. También hay documentos conjuntos de DSDM y Rational, con participación de Jennifer Stapleton, que demuestran la compatibilidad del modelo DSDM con RUP, a despecho de sus fuertes diferencias terminológicas.

También hay casos de éxito (particularmente el de Fujitsu European Repair Centre) en que se emplearon Visual Basic como lenguaje, SQL Server como base de datos y Windows como plataforma de desarrollo e implementación.

4.5 FDD – Feature Driven Development

Feature Oriented Programming (FOP) es una técnica de programación guiada por rasgos o características (features) y centrada en el usuario, no en el programador; su objetivo es sintetizar un programa conforme a los rasgos requeridos. En un desarrollo en términos de FOP, los objetos se organizan en módulos o capas conforme a rasgos. FDD, en cambio, es un método ágil, iterativo y adaptativo. A diferencia de otras metodologías ágiles no cubre todo el ciclo de vida sino sólo las fases de diseño y construcción y se considera adecuado para proyectos mayores y de misión crítica. FDD es, además, marca registrada de una empresa, Nebulon Pty. Aunque hay coincidencias entre la programación orientada por rasgos y el desarrollo guidado por rasgos, FDD no necesariamente implementa FOP.

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 De Luca: Java Modeling in Color with UML; luego fue desarrollado con amplitud en un proyecto mayor de desarrollo por DeLuca, Coad y Stephen Palmer. Su implementación de referencia, análogo al C3 de XP, fue el Singapore Project; DeLuca había sido contratado para salvar un sistema muy complicado para el cual el contratista anterior había producido, luego de dos años, 3500 páginas de documentación y ninguna línea de código. Naturalmente, el proyecto basado en FDD fue todo un éxito, y permitió fundar el método en un caso real de misión crítica.

Los principios de FDD son pocos y simples:

- Se requiere un sistema para construir sistemas si se pretende escalar a proyectos grandes.
- Un proceso simple y bien definido trabaja mejor.
- Los pasos de un proceso deben ser lógicos y su mérito inmediatamente obvio para cada miembro del equipo.
- Vanagloriarse del proceso puede impedir el trabajo real.

- Los buenos procesos van hasta el fondo del asunto, de modo que los miembros del equipo se puedan concentrar en los resultados.
- Los ciclos cortos, iterativos, orientados por rasgos (features) son mejores.

Hay tres categorías de rol en FDD: roles claves, roles de soporte y roles adicionales. Los seis roles claves de un proyecto son: (1) administrador del proyecto, quien tiene la última palabra en materia de visión, cronograma y asignación del personal; (2) arquitecto jefe (puede dividirse en arquitecto de dominio y arquitecto técnico); (3) manager de desarrollo, que puede combinarse con arquitecto jefe o manager de proyecto; (4) programador jefe, que participa en el análisis del requerimiento y selecciona rasgos del conjunto a desarrollar en la siguiente iteración; (5) propietarios de clases, que trabajan bajo la guía del programador jefe en diseño, codificación, prueba y documentación, repartidos por rasgos y (6) experto de dominio, que puede ser un cliente, patrocinador, analista de negocios o una mezcla de todo eso.

Los cinco roles de soporte comprenden (1) administrador de entrega, que controla el progreso del proceso revisando los reportes del programador jefe y manteniendo reuniones breves con él; reporta al manager del proyecto; (2) abogado/guru de lenguaje, que conoce a la perfección el lenguaje y la tecnología; (3) ingeniero de construcción, que se encarga del control de versiones de los builds y publica la documentación; (4) herramientista (toolsmith), que construye herramientas ad hoc o mantiene bases de datos y sitios Web y (5) administrador del sistema, que controla el ambiente de trabajo o productiza el sistema cuando se lo entrega.

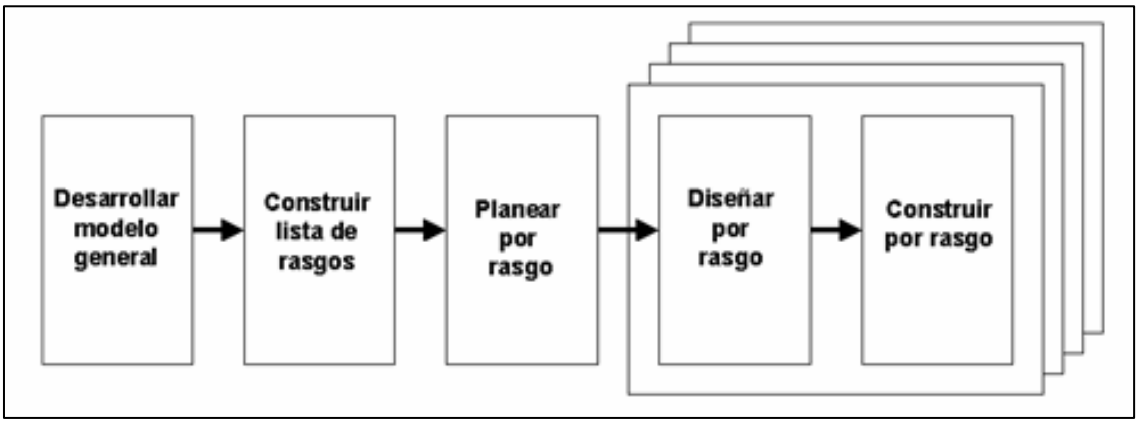

Figura N°9. Proceso FDD

Los tres roles adicionales son los de verificadores, encargados del despliegue y escritores técnicos. Un miembro de un equipo puede tener otros roles a cargo, y un solo rol puede ser compartido por varias personas.

Algunos agilistas sienten que FDD es demasiado jerárquico para ser un método ágil, porque demanda un programador jefe, quien dirige a los propietarios de clases, quienes dirigen equipos de rasgos. Otros críticos sienten que la ausencia de procedimientos detallados de prueba en FDD es llamativa e impropia. Los promotores del método aducen que las empresas ya tienen implementadas sus herramientas de prueba, pero subsiste el problema de su adecuación a FDD. Un rasgo llamativo de FDD es que no exige la presencia del cliente.

FDD se utilizó por primera vez en grandes aplicaciones bancarias a fines de la década de 1990. Los autores sugieren su uso para proyectos nuevos o actualizaciones de sistemas existentes, y recomiendan adoptarlo en forma gradual.

4.6 ASD – Adaptive Software Development

James Highsmith, consultor de Cutter Consortium, desarrolló ASD hacia el año 2000 con la intención primaria de ofrecer una alternativa a la idea de que la optimización es la única solución para problemas de complejidad creciente. Este método ágil pretende abrir una tercera vía entre el "desarrollo monumental de software" y el "desarrollo accidental", o entre la burocracia y la adhocracia. Deberíamos buscar más bien, afirma Highsmith, "el rigor estrictamente necesario"; para ello hay que situarse en coordenadas apenas un poco fuera del caos y ejercer menos control que el que se cree necesario.

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. El pensador de quien Highsmith toma estas ideas es John Holland, el creador del algoritmo genético y probablemente el investigador actual más

importante en materia de procesos emergentes. Holland se pregunta, entre otras cosas, cómo hace un macro-sistema extremadamente complejo, no controlado de arriba hacia abajo en todas las variables intervinientes (como por ejemplo la ciudad de Nueva York o la Web) para mantenerse funcionando en un aparente equilibrio sin colapsar.

La respuesta, que tiene que ver con la auto-organización, la adaptación al cambio y el orden que emerge de la interacción entre las partes, remite a examinar analogías con los sistemas adaptativos complejos por excelencia, esto es: los organismos vivientes (o sus análogos digitales, como las redes neuronales auto-organizativas de Teuvo Kohonen y los autómatas celulares desde Von Neumann a Stephen Wolfram). Para Highsmith, los proyectos de software son sistemas adaptativos complejos y la optimización no hace más que sofocar la emergencia necesaria para afrontar el cambio. En los sistemas complejos no es aplicable el análisis, porque no puede deducirse el comportamiento del todo a partir de la conducta de las partes, ni sumarse las propiedades individuales para determinar las características del conjunto: el oxígeno es combustible, el hidrógeno también, pero cuando se combinan se obtiene agua, la cual no tiene esa propiedad.

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.

Aspectos claves de ASD son:

1. Un conjunto no estándar de "artefactos de misión" (documentos para tí y para mí), incluyendo una visión del proyecto, una hoja de datos, un perfil de misión del producto y un esquema de su especificación
2. Un ciclo de vida, inherentemente iterativo.
3. Cajas de tiempo, con ciclos cortos de entrega orientados por riesgo.

Un ciclo de vida es una iteración; este ciclo se basa en componentes y no en tareas, es limitado en el tiempo, orientado por riesgos y tolerante al cambio. Que se base en componentes implica concentrarse en el desarrollo de software que trabaje, construyendo el sistema pieza por pieza. En este paradigma, el cambio es bienvenido y necesario, pues se concibe como la oportunidad de aprender y ganar así una ventaja competitiva; de ningún modo es algo que pueda ir en detrimento del proceso y sus resultados.

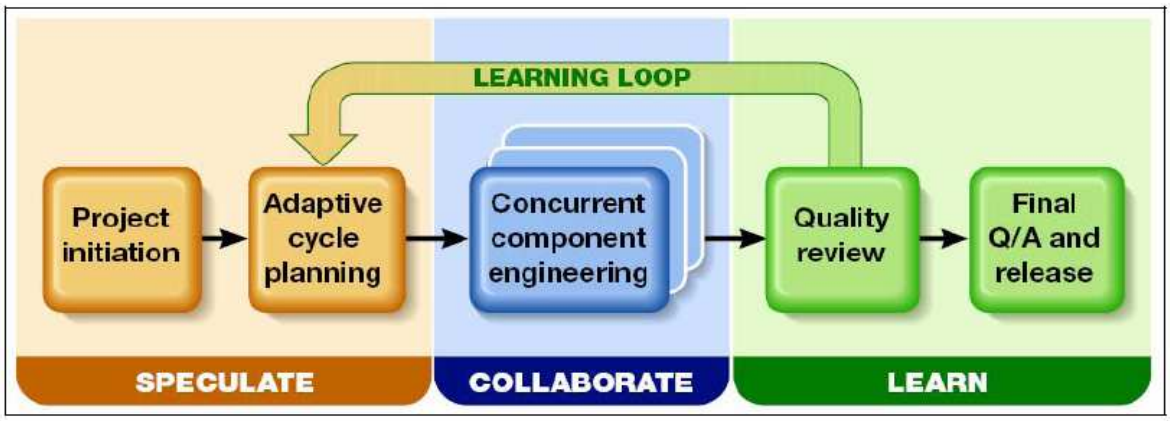

Figura N°10. Actividades del Ciclo de vida Adaptativo

Hay ausencia de estudios de casos del método adaptativo, aunque las referencias literarias a sus principios son abundantes. Como ASD no constituye un método de ingeniería de ciclo de vida sino una visión cultural o una epistemología, no califica como framework suficiente para articular un proyecto. Más visible es la participación de Highsmith en el respetado Cutter Consortium, del cual es director del Agile Project Management Advisory Service. Entre las empresas que han requerido consultoría adaptativa se cuentan AS Bank de Nueva Zelanda, CNET, GlaxoSmithKline, Landmark, Nextel, Nike, Phoenix International Health, Thoughworks y Microsoft.

5. Crítica de a las Metodologías Ágiles

Es natural que exista oposición a las metodologías ágiles y que en ocasiones la reacción frente a ellos adoptará una fuerte actitud combativa. Si los ágiles han sido implacables en sus criticas a las metodologías rigurosas, los adversarios que se han ganado noi se han quedado atrás en sus réplicas.

Uno de los ataques más duros y tempranos contra las metodologías ágiles proviene de una carta de Steven Rakitin a la revista Computer, bajo la rúbrica "El Manifiesto genera cinismo". Refiriéndose a la estructura opositiva del Manifiesto, Rakitin expresa que en su experiencia, los elementos de la derecha (vinculados a la mentalidad planificadora) son esenciales, mientras que los de la izquierda sólo sirven como excusas para que los hackers escupan código irresponsablemente sin ningún cuidado por la disciplina de ingeniería. Así como hay una lectura literal y una estrecha del canon de CMM, Rakitin practica una "interpretación hacker" de propuestas de valor tales como "responder al cambio en vez de seguir un plan" y encuentra que es un generador de caos. La interpretación hacker de ese mandamiento sería algo así como: "¡Genial! Ahora tenemos una razón para evitar la planificación y codificar como nos dé la gana".

Más tarde, un par de libros de buena venta, Questioning Extreme Programming de Pete McBreen y Extreme Programming Refactored: The case against XP de Matt Stephens y Doug Rosenberg promovieron algunas dudas sobre XP pero no llegaron a constituirse como impugnaciones convincentes por su uso de evidencia circunstancial, su estilo redundante, su tono socarrón y su falta de método argumentativo.

Inesperadamente, el promotor de RAD Steve McConnell se ha opuesto con vehemencia a las ideas más radicales del movimiento ágil. Se ha manifestado contrario tanto a las estrategias técnicas como a las tácticas promocionales; dice McConnell: "Esta industria tiene una larga historia en cuanto a pegarse a cuanta moda sale", pero "luego se encuentra, cuando los bombos y platillos se enfrían, que estas tendencias no logran el éxito que sus evangelistas prometieron". Los cargos de McConnell contra XP son numerosos. En primer lugar, diferentes proyectos requieren distintos procesos; no se puede escribir software de aviónica para un contrato de defensa de la misma manera en que se escribe un sistema de inventario para un negocio de alquiler de videos. Además, las reglas de XP son excesivamente rígidas; casi nadie aplica las 12 en su totalidad, y pocas de ellas son nuevas. En diversas presentaciones públicas y en particular en SD West 2004, McConnell ha considerado la programación sin diseño previo, el uso atropellado de XP, la programación automática y la costumbre de llamar "ágil" a cualquier práctica como algunas de las peores ideas en construcción de software del año 2000.

6. Conclusiones

- No existe una metodología universal para hacer frente con éxito a cualquier proyecto de desarrollo de software.
- Las metodologías tradicionales históricamente han intentado abordar la mayor cantidad de situaciones del contexto del proyecto, exigiendo un esfuerzo considerable para ser adaptadas, sobre todo en proyectos pequeños y de requisitos cambiantes.
- Las metodologías ágiles ofrecen una solución casi a medida para una gran cantidad de proyectos.
- Las metodologías ágiles se caracterizan por su sencillez, tanto en su aprendizaje como en su aplicación; sin embargo, gozan tanto de ventajas como de inconvenientes.
- Las metodologías ágiles permiten a los pequeños grupos de desarrollo concentrarse en la tarea de construir software fomentando prácticas de fácil adopción y en un entorno ordenado que permiten que los proyectos finalicen exitosamente.
- Se pueden distinguir dos rangos distintos de conjuntos de metodologías ágiles. Por un lado están las metodologías más declarativas y programaticas como XP, Scrum, LD, entre otras; y por otro lado se encuentran las metodologías finamente elaboradas como DSDM, Cristal, etc.
- XP es una de las metodologías ágiles más extendidas y populares, además es considerada como una metodología posmoderna cuyas grandes capacidades se generan a través de procesos emergentes.
- Scrum implementa en sus prácticas de desarrollo una estrategia de caos controlado, permitiendo maximizar la realimentación sobre el desarrollo pudiendo corregir problemas y mitigar riesgos de forma temprana.
- A pesar de las continuas criticas que las metodologías ágiles sufren, son usadas por muchas grandes empresas y se han utilizado en grandes sistemas, lo que hace prever que estas metodologías han llegado para quedarse.

7. Referencias

[1]. Boehm, Barry W., "A Spiral Model of Software Development and Enhancement", IEEE Computer, Vol.21, No 15 (Mayo 1988), pp.61-72.

[2]. Martin, J., Rapid Application Development, Macmillan Inc., New York, 1991.

[3]. Beck, Kent, Extreme Programming Explained, Addison-Wesley The XP Series, 2000.

[4]. Alistair Cockburn. "Balancing Lightness with Sufficiency". Septiembre del 2000. Disponible en: http://alistair.cockburn.us/crystal/articles/blws/balancinglightnesswithsufficiency.html, Setiembre de 2000.

[5]. Pérez S. Jesús, Metodologías Ágiles: La ventaja competitiva de estar preparado para tomar decisiones lo más tarde posible y cambiarlas en cualquier momento. Articulo de Agile Spain. Disponible en: http://www.spinec.org/wp-content/metodologiasagiles_01.pdf

[6]. Grupo ISSI, Metodologías Ágiles en el Desarrollo de Software, Artículo de Grupo ISSI (Noviembre 2003). Disponible en: http://issi.dsic.upv.es/tallerma/actas.pdf

[7]. Acebal F.César, Cueva L. Juan, eXtreme Programming (XP): un nuevo método de desarrollo de software, Articulo de Novática. Disponible en: http://www.ati.es/novatica/2002/156/156-8.pdf

[8]. Letelier P., Penadés C., Metodologías ágiles para el desarrollo de Software: eXtreme Programming (XP), Universidad Politécnica de Valencia Disponible en: http://www.willydev.net/descargas/masyxp.pdf

[9]. Shenone M. Hernán, Diseño de una metodología Ágil de Desarrollo de Software, Tesis de Grado en Ingeniería en Informática, Universidad de Buenos Aires. Disponible en: http://www.fi.uba.ar/materias/7500/schenonetesisdegradoingenieriainformatica.pdf

[10]. Página oficial de SCRUM: SCRUM It's About Common Sense Disponible en: http://www.controlchaos.com/

[11]. Página de Microsoft: MSDN en Español, Métodos Heterodoxos en Desarrollo de Software, Artículo de Carlos Reynoso – Universidad de Buenos Aires, Abril del 2004. Disponible en: http://www.microsoft.com/spanish/msdn/arquitectura/roadmap_arq/heterodox.asp#12

[12]. Pagina oficial de Quadrus Developments Disponible en: http://www.enterprisexp.org

[13]. Pagina oficial de DSDM Disponible en: http://www.dsdm.org