Maida, Esteban Gabriel ; Pacienzia, Julián

Metodologías de desarrollo de software

Tesis de Licenciatura en Sistemas y Computación
Facultad de Química e Ingeniería "Fray Rogelio Bacon"

FACULTAD DE QUÍMICA E INGENIERIA "FRAY ROGELIO BACON"

PONTIFICIA UNIVERSIDAD CATÓLICA ARGENTINA SANTA MARIA DE LOS BUENOS AIRES

METODOLOGIAS DE DESARROLLO DE SOFTWARE

Tesis Final de Licenciatura en Sistemas y Computación

Maida, Esteban Gabriel

Pacienzia, Julián

Diciembre 2015

Cátedra Seminario de Sistemas

Agradecimientos y dedicatorias

La presente Tesis es un esfuerzo en el cual, directa o indirectamente, participaron varias personas, leyendo, opinando, corrigiendo, teniéndonos paciencia, dándonos ánimo, acompañándonos en los momentos de crisis y en los momentos de felicidad.

Por estos motivos queremos agradecer a nuestras familias y a la gente que siempre apoyaron nuestros proyectos y ambiciones.

A nuestros amigos y compañeros que hicieron de estos años, los mejores de nuestras vidas.

A todos los profesores de nuestra carrera profesional porque han aportado su granito de arena en nuestra formación, dedicándonos todo el tiempo que fuera necesario para que hoy seamos personas hechas y derechas, basándonos en la verdad, la justicia y la humildad.

Resumen

En la actualidad la rapidez y el dinamismo en la industria del software han hecho replantear los cimientos sobre los que se sustenta el desarrollo de software tradicional. Estudios recientes y el mismo mercado actual está marcando la tendencia en la ingeniería del software teniendo como características principales atender a las necesidades de rapidez, flexibilidad y variantes externas que hacen de nuestro entorno una ventaja más competitiva al aumentar la productividad y satisfacer las necesidades del cliente en el menor tiempo posible para proporcionar mayor valor al negocio.

Ante esta situación, el grado de adaptación de las metodologías tradicionales a estos entornos de trabajo no eran del todo eficientes y no cubrían las necesidades del mercado actual.

En la actualidad existen una gran cantidad de metodologías para el desarrollo de software, separadas en dos grandes grupos; las metodologías tradicionales o pesadas y las metodologías agiles.

Las metodologías tradicionales se basan en las buenas prácticas dentro de la ingeniería del software, siguiendo un marco de disciplina estricto y un riguroso proceso de aplicación.

Las metodologías agiles, en cambio, representan una solución a los problemas que requieren una respuesta rápida en un ambiente flexible y con cambios constantes, haciendo caso omiso de la documentación rigurosa y los métodos formales.

El objetivo de esta investigación es presentar e introducir sobre las existentes metodologías para el desarrollo de software y los paradigmas que marcan la diferencia entre un método estructurado y un método ágil para así poder identificar cual se adapta de manera más eficiente a un proyecto determinado.

INDICE

3. INGENIERIA DE SOFTWARE

4. METODOLOGIAS CLASICAS

4.1. INTRODUCCION
4.2. VENTAJAS Y DESVENTAJAS
4.3. TIPOS DE METODOLOGIAS
4.3.1. WATERFALL (CASCADA)
4.3.2. PROTOTYPING
4.3.3. SPIRAL
4.3.4. INCREMENTAL
4.3.5. RAD

5. METODOLOGIAS AGILES

5.1. INTRODUCCION
5.2. BENEFICIOS
5.3. TIPOS DE METODOLOGIAS
5.3.1. PROGRAMACION EXTREMA
5.3.2. SCRUM
5.3.3. CRYSTAL
5.3.4. KANBAN
5.3.5. FEATURE DRIVEN DEVELOPMENT (FDD)
5.3.6. ADAPTIVE SOFTWARE DEVELOPMENT (ASD)
5.3.7. LEAN DEVELOPMENT (LD) Y LEAN SOFTWARE DEVELOPMENT (LSD)
5.3.8. PROCESO UNIFICADO DE DESARROLLO SOFTWARE

6. ESTUDIO Y ANALISIS DE CASOS REALES

6.1. CASOS DE EXITO
6.2. CASOS DE FRACASO

7. CONCLUSION

8. REFERENCIAS BIBLIOGRAFICAS

9. APENDICE

1. INTRODUCCION

1.1. DEFINICION DEL PROBLEMA

Actualmente las metodologías de ingeniería de software pueden considerarse como una base necesaria para la ejecución de cualquier proyecto de desarrollo de software que se considere serio, y que necesite sustentarse en algo más que la experiencia y capacidades de sus programadores y equipo. Estas metodologías son necesarias para poder realizar un proyecto profesional, tanto para poder desarrollar efectiva y eficientemente el software, como para que sirvan de documentación y se puedan rendir cuentas de los resultados obtenidos.

Un amplio y buen conocimiento de estas metodologías servirá de base teórica y permitirá comprender completamente todo lo que requiere el análisis, diseño, desarrollo e implantación de un sistema. Además es importante, por la demanda que se tiene hoy en día por parte de muchas empresas, el conocimiento de algunas metodologías de desarrollo de software en específico.

Lo más importante en una primera etapa es poder identificar qué metodología de ingeniería de software se adecúa de la mejor manera a nuestro proyecto, para así lograr el mejor resultado en tiempo y forma.

1.2. PROPOSITO DE LA INVESTIGACION

La presente tesis se orienta a realizar una contribución en el área de metodología para el diseño, desarrollo y evaluación de software, necesarios para abordar proyectos con una metodología diferente a la estructurada.

1.3. OBJETIVOS GENERALES

La selección y aplicación de una metodología particular para el desarrollo de software, se centra en el uso de un enfoque sistemático de pasos y etapas a seguir para el cumplimiento de los objetivos en común.

El objetivo general de la puesta en práctica de una metodología del software es construir un producto de alta calidad de una manera oportuna. Dicha selección implica un conjunto de principios fundamentales que se deben seguir y cumplir. Estos incluyen actividades explícitas para el entendimiento del problema y la comunicación con el cliente, métodos definidos para representar un diseño, mejores prácticas para la implementación de la solución y estrategias y tácticas sólidas para las pruebas.

Para conseguir el objetivo de construir productos de alta calidad dentro de la planificación, las metodologías en general emplean una serie de prácticas para:

- Entender el problema
- Diseñar una solución
- Implementar la solución correctamente
- Probar la solución
- Gestionar las actividades anteriores para conseguir alta calidad

La utilización de la metodología adecuada, representa un proceso formal que incorpora una serie de métodos bien definidos para el análisis, diseño, implementación y pruebas del software y sistemas. Además, abarca una amplia colección de métodos y técnicas de gestión de proyectos para el aseguramiento de la calidad y la gestión de la configuración del software.

1.4. ALCANCES

Se realiza una exposición de los dos grandes grupos de metodologías, estructuradas y ágiles, en toda su extensión, desde las primeras etapas del análisis hasta la implementación final del software y su seguimiento. El alcance de la tesis trata de introducir al conocimiento de metodologías de trabajo más modernas de acuerdo a los desafíos presentados en el mundo actual.

1.5. LIMITACIONES

Este trabajo permite la introducción al conocimiento teórico para comprender las fases y etapas de las metodologías para el desarrollo de software existente. También se tratarán algunos casos prácticos en el capítulo 6.

1.6. METODOLOGIAS

El desarrollo de software no es una tarea fácil. Prueba de ello es que existen numerosas propuestas metodológicas que inciden en distintas dimensiones del proceso de desarrollo. Por una parte tenemos aquellas propuestas más tradicionales que se centran especialmente en el control del proceso, estableciendo rigurosamente las actividades involucradas, los artefactos que se deben producir, y las herramientas y notaciones que se usarán. Estas propuestas han demostrado ser efectivas y necesarias en un gran número de proyectos, pero también han presentado problemas en muchos otros. Una posible mejora es incluir en los procesos de desarrollo más actividades, más artefactos y más restricciones, basándose en los puntos débiles detectados. Sin embargo, el resultado final sería un proceso de desarrollo más complejo que puede incluso limitar la propia habilidad del equipo para llevar a cabo el proyecto. Otra aproximación es centrarse en otras dimensiones, como por

ejemplo el factor humano o el producto software. Esta es la filosofía de las metodologías ágiles, las cuales dan mayor valor al individuo, a la colaboración con el cliente y al desarrollo incremental del software con iteraciones muy cortas. Este enfoque está mostrando su efectividad en proyectos con requisitos muy cambiantes y cuando se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. Las metodologías ágiles están revolucionando la manera de producir software, y a la vez generando un amplio debate entre sus seguidores y quienes por escepticismo o convencimiento no las ven como alternativa para las metodologías tradicionales.

Un objetivo claro ha sido encontrar procesos y metodologías, que sean sistemáticas, predecibles y repetibles, a fin de mejorar la productividad en el desarrollo y la calidad del producto software.
La evolución de la disciplina de ingeniería del software ha traído consigo propuestas diferentes para mejorar los resultados del proceso de construcción. Las metodologías tradicionales haciendo énfasis en la planificación y las metodologías ágiles haciendo énfasis en la adaptabilidad del proceso, delinean las principales propuestas presentes.

En el próximo capítulo trataremos algunos conceptos básicos referidos al marco teórico, tales como, definición de ingeniería, software, metodologías y paradigmas de la ingeniería. En el capítulo 3 (tres) abordaremos temas relacionados específicamente a la ingeniería del software, como por ejemplo, cuáles son los objetivos fundamentales, administración de un proyecto, recursos y el ciclo de vida. Continuaremos con el capítulo nº 4 (cuatro) donde explicaremos las metodologías clásicas, con sus ventajas y desventajas, para luego proseguir con el siguiente capítulo donde trataremos temas exclusivos de las metodologías agiles. Posteriormente veremos estudios y análisis de casos reales, como casos de éxitos, de fracasos y análisis críticos. Finalmente sacaremos nuestras propias conclusiones respecto a todo el trabajo realizado.

2. MARCO TEORICO

2.1. DEFINICION DE INGENIERIA

La ingeniería es el conjunto de conocimientos y técnicas, científicas aplicadas al desarrollo, implementación, mantenimiento y perfeccionamiento de estructuras (tanto físicas como teóricas) para la resolución de problemas que afectan la actividad cotidiana de la sociedad.

La ingeniería es la actividad de transformar el conocimiento en algo práctico. (Autores varios).

2.2. SOFTWARE

El software es el equipamiento lógico e intangible de un sistema informático, que comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas, en contraposición a los componentes físicos que son llamados hardware.

En otras palabras es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación. (Autores varios).

2.2.1.METODOLOGIA

Una metodología es un conjunto integrado de técnicas y métodos que permite abordar de forma homogénea y abierta cada una de las actividades del ciclo de vida de un proyecto de desarrollo. Es un proceso de software detallado y completo. (Autores varios).

Las metodologías se basan en una combinación de los modelos de proceso genéricos. Definen artefactos, roles y actividades, junto con prácticas y técnicas recomendadas.

La metodología para el desarrollo de software es un modo sistemático de realizar, gestionar y administrar un proyecto para llevarlo a cabo con altas posibilidades de éxito. Una metodología para el desarrollo de software comprende los procesos a seguir sistemáticamente para idear, implementar y mantener un producto software desde que surge la necesidad del producto hasta que cumplimos el objetivo por el cual fue creado.

Si esto se aplica a la ingeniería del software, podemos destacar que una **metodología**:

- Optimiza el proceso y el producto software.
- Métodos que guían en la planificación y en el desarrollo del software.
- Define qué hacer, cómo y cuándo durante todo el desarrollo y mantenimiento de un proyecto.

Una metodología define una estrategia global para enfrentarse con el proyecto. Entre los elementos que forman parte de una metodología se pueden destacar:

- Fases: tareas a realizar en cada fase o etapa.
- Productos: E/S de cada fase, documentos.

- Procedimientos y herramientas: apoyo a la realización de cada tarea.
- Criterios de evaluación: del proceso y del producto. Saber si se han logrado los objetivos.

Una metodología de desarrollo de software es un marco de trabajo que se usa para estructurar, planificar y controlar el proceso de desarrollo de sistemas de información. Una gran variedad de estos marcos de trabajo han evolucionado durante los años, cada uno con sus propias fortalezas y debilidades. Una metodología de desarrollo de sistemas no tiene que ser necesariamente adecuada para usarla en todos los proyectos. Cada una de las metodologías disponibles es más adecuada para tipos específicos de proyectos, basados en consideraciones técnicas, organizacionales, de proyecto y de equipo.

Una metodología de desarrollo de software o metodología de desarrollo de sistemas en ingeniería de software es un marco de trabajo que se usa para estructurar, planificar y controlar el proceso de desarrollo de un sistema de información.

El marco de trabajo de una metodología de desarrollo de software consiste en:

- Una filosofía de desarrollo de software, con el enfoque o enfoques del proceso de desarrollo de software.
- Múltiples herramientas, modelos y métodos para ayudar en el proceso de desarrollo de software.

Estos marcos de trabajo están con frecuencia vinculados a algunos tipos de organizaciones, que se encargan del desarrollo, soporte de uso y promoción de la metodología. La metodología con frecuencia se documenta de alguna manera formal.

2.2.2.IMPORTANCIA DEL SOFTWARE

El software es ahora la clave del éxito de muchas empresas y negocios, ya que sin él sería casi imposible el mantenimiento y crecimiento de los mismos. Lo que diferencia una compañía de otra es la suficiencia, exactitud y oportunidad de la información dada por el software.

El desarrollo de software se ha convertido en una industria con crecimiento vertical en los últimos años. Hace un par de décadas se sostenía la teoría de que los países que poseían los mejores recursos naturales estaban destinados a ser los más ricos y poderosos del mundo. Indudablemente los recursos naturales tienen un papel importante en la economía de los países, sin embargo poco a poco se fue acuñando una nueva ideología que se sintetiza en lo siguiente: "*El que posee la información y el conocimiento y hace mejor uso de él, es el que tiene el poder*".

2.2.3.PROBLEMAS DEL SOFTWARE

La planificación y estimación de costos frecuentemente son imprecisas.

- Falta de productividad.
- La calidad del software es a veces inaceptable.

Estos problemas al final crean insatisfacción y falta de confianza de los clientes. Los problemas anteriores son sólo manifestación de otras dificultades:

- No tenemos tiempo de recoger datos sobre el proceso de desarrollo del software.
- Los proyectos de desarrollo de software se llevan a cabo con sólo una vaga indicación de los requisitos del cliente.
- La calidad del software es normalmente cuestionable.
- El mantenimiento de software es muy costoso y no se le ha considerado un aspecto importante.

Los problemas anteriores son solucionables, dándoles un enfoque de ingeniería al desarrollo de software.

2.2.4.CARACTERISTICAS BASICAS

El software es un elemento del sistema que es lógico. Por tanto, el software tiene características considerablemente distintas al hardware:

- El software se desarrolla, no se fabrica en un sentido clásico.
- El software no se desgasta con el tiempo.
- La mayoría de software se construye a medida, en vez de ensamblar componentes existentes.
- Está creando en base a la lógica puramente.

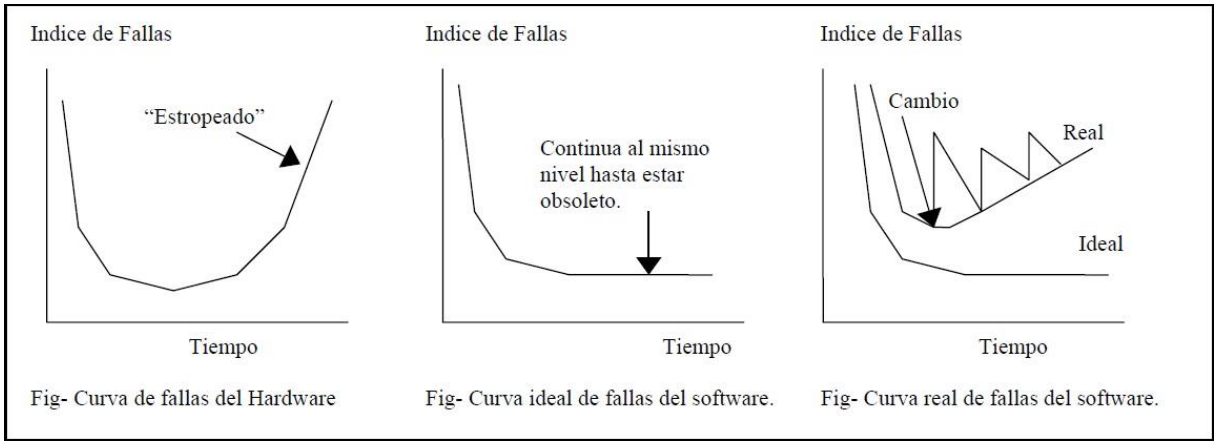

Figura Nº 1 – Comparación de índices de fallas entre hardware y software.

2.2.5.CLASIFICACION

El software pude clasificarse en tres grandes grupos:

Software de sistema: Se llama *Software de Sistema* o *Software de Base* al conjunto de programas que sirven para interactuar con el sistema, confiriendo control sobre el hardware, además de dar soporte a otros programas que administran los recursos y proporcionan una interfaz de uso.

El mejor ejemplo en este sentido son los populares sistemas operativos como Windows, Linux o Mac OS. Este tipo de sistemas incluye entre otros:

- Sistemas operativos
- Controladores de dispositivos
- Herramientas de diagnóstico
- Herramientas de Corrección y Optimización
- Servidores de información
- Programas utilitarios

Software de programación: Es el conjunto de herramientas que permiten al programador desarrollar programas informáticos, usando diferentes alternativas y lenguajes de programación, de una manera práctica. Estos incluyen básicamente:

- Editores de texto
- Compiladores
- Intérpretes
- Enlazadores
- Depuradores
- Entornos de Desarrollo Integrados (IDE): Agrupan las anteriores herramientas, usualmente en un entorno visual, de forma tal que el programador no necesite introducir múltiples comandos para compilar, interpretar, depurar, etc. Habitualmente cuentan con una avanzada interfaz gráfica de usuario (GUI).

_____

Figura Nº 1 Extraída de [1]

Software de aplicación: son los programas diseñados para o por los usuarios para facilitar la realización de tareas específicas en la computadora, como pueden ser las aplicaciones ofimáticas u otros tipos de software especializados. Incluye entre muchos otros:

- Aplicaciones para Control de sistemas y automatización industrial
- Aplicaciones ofimáticas (procesador de texto, hoja de cálculo, programa de presentación)
- Software educativo
- Software empresarial
- Bases de datos
- Telecomunicaciones (por ejemplo Internet y toda su estructura lógica)
- Software médico
- Software de cálculo numérico y simbólico.
- Software de diseño asistido (CAD)
- Software de control numérico (CAM)

2.3. QUE ES LA INGENIERIA DE SOFTWARE

Haciendo una recopilación de todos los conceptos que se han dado sobre la Ingeniería de software, la podemos definir como la disciplina o área de la informática, que hace uso razonable de los principios de ingeniería con el objetivo de obtener soluciones informáticas económicamente factible y que se adapte a las necesidades de las empresas reales, tomando en cuenta los procesos de producción y mantenimiento de software que son desarrollados y modificados en el tiempo y con los costos estimados.

Esta ingeniería trata con áreas muy diversas de la informática y de las Ciencias de la Computación, tales como construcción de compiladores, Sistemas Operativos, o desarrollos Intranet/Internet, abordando todas las fases del ciclo de vida del desarrollo de cualquier tipo de Sistema de Información y aplicables a infinidad de áreas (negocios, investigación científica, medicina, producción, logística, banca, etc.).

Algunas definiciones, dadas a través del tiempo son:

- *"Ingeniería de Software es el estudio de los principios y metodologías para el desarrollo y mantenimiento de sistemas de software"* (Zelkovitz, 1978)
- *"Ingeniería de software es la aplicación práctica del conocimiento científico al diseño y construcción de programas de computadora y a la documentación asociada requerida para desarrollar, operar y mantenerlos. Se conoce también como Desarrollo de Software o Producción de Software"* (Bohem, 1976).
- *"Ingeniería de Software trata del establecimiento de los principios y métodos de la ingeniería a fin de obtener software de modo rentable, que sea fiable y trabaje en máquinas reales"* (Bauer, 1972).
- *"Es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software; es decir, la aplicación de la ingeniería al software"* (IEEE, 1993).

En conclusión podemos decir que los cuatro autores anteriores, de manera diferente describen en si el principal objetivo de la ingeniería de software, la cual es el establecimiento y puesta en práctica de los principios y metodologías que nos lleven a un desarrollo eficiente de software en todas las etapas desde sus inicios hasta su implementación y mantenimiento.

2.4. QUE ES UNA METODOLOGIA Y PARA QUE SE UTILIZA

La metodología hace referencia al conjunto de procedimientos racionales utilizados para alcanzar un objetivo que requiera habilidades y conocimientos específicos.

La metodología es una de las etapas específicas de un trabajo o proyecto que parte de una posición teórica y conlleva a una selección de técnicas concretas o métodos acerca del procedimiento para el cumplimiento de los objetivos. Es el conjunto de métodos que se utilizan en una determinada actividad con el fin de formalizarla y optimizarla. Determina los pasos a seguir y cómo realizarlos para finalizar una tarea.

2.5. METODOLOGIA TRADICIONAL

Desarrollar un buen software depende de un gran número de actividades y etapas, donde el impacto de elegir la metodología para un equipo en un determinado proyecto es trascendental para el éxito del producto.

Las metodologías tradicionales son denominadas, a veces, de forma despectiva, como metodologías pesadas.

Centran su atención en llevar una documentación exhaustiva de todo el proyecto, la planificación y control del mismo, en especificaciones precisas de requisitos y modelado y en cumplir con un plan de trabajo, definido todo esto, en la fase inicial del desarrollo del proyecto.

Estas metodologías tradicionales imponen una disciplina rigurosa de trabajo sobre el proceso de desarrollo del software, con el fin de conseguir un software más eficiente. Para ello, se hace énfasis en la planificación total de todo el trabajo a realizar y una vez que está todo detallado, comienza el ciclo de desarrollo del producto software. Se centran especialmente en el control del proceso, mediante una rigurosa definición de roles, actividades, artefactos, herramientas y notaciones para el modelado y documentación detallada. Además, las metodologías tradicionales no se adaptan adecuadamente a los cambios, por lo que no son métodos adecuados cuando se trabaja en un entorno, donde los requisitos no pueden predecirse o bien pueden variar.

Otra de las características importantes dentro de este enfoque, son los altos costes al implementar un cambio y la falta de flexibilidad en proyectos donde el entorno es volátil.

2.6. METODOLOGIAS AGILES

Este enfoque nace como respuesta a los problemas que puedan ocasionar las metodologías tradicionales y se basa en dos aspectos fundamentales, retrasar las decisiones y la planificación adaptativa. Basan su fundamento en la adaptabilidad de los procesos de desarrollo.

Un modelo de desarrollo ágil, generalmente es un proceso *Incremental* (entregas frecuentes con ciclos rápidos), también *Cooperativo* (clientes y desarrolladores trabajan constantemente con una comunicación muy fina y constante), *Sencillo* (el método es fácil de aprender y modificar para el equipo) y finalmente *Adaptativo* (capaz de permitir cambios de último momento). Las metodologías ágiles proporcionan una serie de pautas y principios junto a técnicas pragmáticas que hacen que la entrega del proyecto sea menos complicada y más satisfactoria tanto para los clientes como para los equipos de trabajo, evitando de esta manera los caminos burocráticos de las metodologías tradicionales, generando poca documentación y no haciendo uso de métodos formales.

Estas metodologías ponen de relevancia que la capacidad de respuesta a un cambio es más importante que el seguimiento estricto de un plan.

2.7. DIFERENCIAS ENTRE METODOLOGIA TRADICIONAL Y AGIL

En las metodologías tradicionales el principal problema es que nunca se logra planificar bien el esfuerzo requerido para seguir la metodología. Pero entonces, si logramos definir métricas que apoyen la estimación de las actividades de desarrollo, muchas prácticas de metodologías tradicionales podrían ser apropiadas. El no poder predecir siempre los resultados de cada proceso no significa que estemos frente a una disciplina de azar. Lo que significa es que estamos frente a la necesidad de adaptación de los procesos de desarrollo que son llevados por parte de los equipos que desarrollan software.

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 por cada proyecto, la problemática sería definir cada una de las prácticas, y en el momento preciso definir parámetros para saber cuál usar.

Es importante tener en cuenta que el uso de un método ágil no vale para cualquier proyecto. 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.

En la tabla que se muestra a continuación aparece una comparativa entre estos dos grupos de metodologías.

Tabla Nº 1 – Comparación entre Metodología Ágil y Tradicional

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 preparados 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 es bastante flexible	Existe un contrato prefijado
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
Poca documentación	Documentación exhaustiva
Muchos ciclos de entrega	Pocos ciclos de entrega

Como se muestra en la Tabla Nº 1, se puede apreciar que las metodologías ágiles, son más baratas en tiempo y recursos, obteniendo los mismos o mejores resultado ante las metodologías tradicionales.

Tabla Nº 1 Extraída de [1]

2.8. PARADIGMAS DE LA INGENIERIA DEL SOFTWARE

La ingeniería de software es reconocida como una disciplina legítima, digna de tener una investigación seria, un estudio cuidadoso y generando una gran controversia. En la industria el ingeniero del software ha sustituido al programador como título de trabajo preferente. Los modelos de procesos de software, métodos de ingeniería de software y herramientas se han adoptado con éxito en el amplio espectro de las aplicaciones industriales. Los gestores y usuarios reconocen la necesidad de un enfoque más disciplinado del software.

Un paradigma de programación es un modelo básico de diseño y desarrollo de programas, que permite producir programas con una directriz específica, tales como: estructura modular, fuerte cohesión, alta rentabilidad, etc.

Existen tres categorías de paradigmas de programación:

a. Los que soportan técnicas de programación de bajo nivel.
b. Los que soportan métodos de diseño de algoritmos.
c. Los que soportan soluciones de programación de alto nivel.

Los paradigmas relacionados con la programación de alto nivel se agrupan en tres categorías de acuerdo con la solución que aportan para resolver el problema:

a. Solución procedimental u operacional. Describe etapa a etapa el modo de construir la solución. Es decir señala la forma de obtener la solución.
b. Solución demostrativa. Es una variante de la procedimental. Especifica la solución describiendo ejemplos y permitiendo que el sistema generalice la solución de estos ejemplos para otros casos. Aunque es fundamentalmente procedimental, el hecho de producir resultados muy diferentes a ésta, hace que sea tratada como una categoría separada.
c. Solución declarativa. Señala las características que debe tener la solución, sin describir cómo procesarla. Es decir señala qué se desea obtener pero no cómo obtenerlo.

***Paradigma**: El concepto de paradigma se utiliza en la vida cotidiana como sinónimo de **"ejemplo"** o para hacer referencia a algo que se toma como **"modelo"**.*

3. INGENIERIA DE SOFTWARE

3.1. INTRODUCCION

En este capítulo se desean presentar los fundamentos en que se basa el software, los métodos, las herramientas y los procedimientos que provee la ingeniería de software a fin de considerarlos para el desarrollo de los programas en general. Se describen y analizan las etapas del proceso de la ingeniería del software, desde la recopilación de los requerimientos hasta la implementación y posterior mantenimiento.

Uno de los problemas más importantes con los que se enfrentan los ingenieros en software y los programadores en el momento de desarrollar un software de aplicación, es la falta de marcos teóricos comunes que puedan ser usados por todas las personas que participan en el desarrollo del proyecto informático.

"La ingeniería del software surge a partir de las ingenierías de sistemas y de hardware, y considera tres elementos claves: que son los métodos, las herramientas y los procedimientos que facilitan el control del proceso de desarrollo de software y brinda a los desarrolladores las bases de la calidad de una forma productiva" (Pressman, 1993).

La ingeniería de software está compuesta por una serie de modelos que abarcan los métodos, las herramientas y los procedimientos. Estos modelos se denominan frecuentemente paradigmas de la ingeniería del software y la elección de un paradigma se realiza básicamente de acuerdo a la naturaleza del proyecto y de la aplicación, los controles y las entregas a realizar.

Para la construcción de un sistema de software, el proceso puede describirse sintéticamente como: la obtención de los requisitos del software, el diseño del sistema de software (diseño preliminar y diseño detallado), la implementación, las pruebas, la instalación, el mantenimiento y la ampliación o actualización del sistema.

El proceso de construcción está formado por etapas que son: la obtención de los requisitos, el diseño del sistema, la codificación y las pruebas del sistema. Desde la perspectiva del producto, se parte de una necesidad, se especifican los requisitos, se obtiene el diseño del mismo, el código respectivo y por último el sistema de software. Algunos autores sostienen que el nombre ciclo de vida ha sido relegado en los últimos años, utilizando en su lugar proceso de software, cambiando la perspectiva de producto a proceso.

El software o producto, en su desarrollo pasa por una serie de etapas que se denominan ciclo de vida, siendo necesario, definir en todas las etapas del ciclo de vida del producto, los procesos, las actividades y las tareas a desarrollar.

Un ciclo de vida establece el orden de las etapas del proceso de software y los criterios a tener en cuenta para poder pasar de una etapa a la siguiente. El tema del ciclo de vida lo han tratado algunas organizaciones profesionales y organismos internacionales como la IEEE (Institute of Electrical and electronics Engineers) y la ISO/IEC (International Standards Organization/International Electrochemical Commission), que han publicado normas tituladas “Standard for Developing Software Life Cycle Proccesses” (Estándar IEEE para el desarrollo de procesos del ciclo de vida del software) (IEEE, 1991) y “Software life-cycle process” (Proceso de ciclo de vida del software) (ISO, 1994).

3.2. OBJETIVOS

Los objetivos principales de la ingeniería de software son los siguientes:

- Identificar las principales metodologías disponibles para la recolección y manejo de requerimientos que deben cumplir los sistemas en desarrollo.
- Conocer las principales herramientas de verificación y validación de software y su utilidad en las diferentes fases del desarrollo de sistemas.
- Diseñar aplicaciones informáticas que se ajusten a las necesidades de las organizaciones.
- Dirigir y coordinar el desarrollo de aplicaciones complejas.
- Intervenir en todas las fases del ciclo de vida de un producto.
- Estimar los costes de un proyecto y determinar los tiempos de desarrollo.
- Hacer el seguimiento de costes y plazos.
- Dirigir equipos de trabajo de desarrollo software.
- Organizar la realización de pruebas que verifiquen el correcto funcionamiento de los programas y que se ajustan a los requisitos de análisis y diseño.
- Diseñar, construir y administrar bases de datos.
- Dirigir y asesorar a los programadores durante el desarrollo de aplicaciones.
- Introducir procedimientos de calidad en los sistemas, evaluando métricas e indicadores y controlando la calidad del software producido.
- Organizar y supervisar el trabajo de su equipo de los técnicos de mantenimiento y los ingenieros de sistemas y redes.

Hoy en día, los productos de software se construyen con un nivel de urgencia que no se veía en años anteriores. La prioridad más alta de las compañías es reducir el tiempo de salida al mercado, que es la base del desarrollo rápido.

La ingeniería de software es percibida por algunos como demasiado formal, que consume demasiado tiempo y demasiado estructurada para la flexibilidad necesaria durante el desarrollo de hoy en día. Las personas que hacen estas críticas exponen que no se pueden permitir la formalidad de un enfoque de ingeniería para construir software porque necesitan desarrollar productos de forma rápida. Las personas que lanzan tales objeciones ven la ingeniería como una disciplina estática y piensan que no se puede adaptar a las necesidades cambiantes del negocio y la industria. La verdad es, sin embargo, que la ingeniería del software es adaptativa y por lo tanto, relevante para cualquiera que construya un producto software.

La ingeniería del software es adaptativa y no una metodología rígida. Es una filosofía que puede ser adaptada y aplicada a todas las actividades y dominios de aplicación del desarrollo de software.

La ingeniería del software proporciona una amplia colección de opciones que los profesionales pueden elegir para construir productos de alta calidad. Sin embargo, no hay un enfoque de ingeniería individual o un conjunto de procesos, métodos o herramientas de ingeniería del software para construir un producto software.

El enfoque de ingeniería del software, incluyendo los procesos, métodos y herramientas puede y debería ser adaptada al producto, la gente que lo construye y el entorno del negocio.

Todo este conjunto de ideas y opiniones llevaron a lo que hoy en día son las metodologías ágiles.

Entre los principios más destacados de la ingeniería del software, podemos señalar los siguientes:

- Haz de la calidad la razón de trabajar.
- Una buena gestión es más importante que una buena tecnología.
- Las personas y el tiempo no son intercambiables.
- Seleccionar el modelo de ciclo de vida adecuado.
- Entregar productos al usuario lo más pronto posible.
- Determinar y acotar el problema antes de escribir los requisitos.
- Realizar un diseño.
- Minimizar la distancia intelectual.
- Documentar.
- Las técnicas son anteriores a las herramientas.
- Primero hazlo correcto, luego hazlo rápido.
- Probar, probar y probar.
- Introducir las mejoras y modificaciones con cuidado.
- Asunción de responsabilidades.
- La entropía del software es creciente.
- La gente es la clave del éxito.
- Nunca dejes que tu jefe o cliente te convenza para hacer un mal trabajo.
- La gente necesita sentir que su trabajo es apreciado.
- La educación continua es responsabilidad de cada miembro del equipo.
- El compromiso del cliente es el factor más crítico en la calidad del software.
- Tu mayor desafío es compartir la visión del producto final con el cliente.
- La mejora continua de tu proceso de desarrollo de software es posible y esencial.
- Tener procedimientos escritos de desarrollo de software puede ayudar a crear una cultura compartida de buenas prácticas.
- La calidad es el principal objetivo; la productividad a largo plazo es una consecuencia de una alta calidad.
- Haz que los errores los encuentre un colaborador, no un cliente.
- Una clave en la calidad en el desarrollo de software es realizar iteraciones en todas las fases del desarrollo excepto en la codificación.
- La gestión de errores y solicitud de cambios es esencial para controlar la calidad y el mantenimiento.
- Si mides lo que haces puedes aprender a hacerlo mejor.
- Haz lo que tenga sentido; no recurras a los dogmas.
- No puedes cambiar todo de una vez. Identifica los cambios que se traduzcan en los mayores beneficios, y comienza a implementarlos.

3.3. DISTRIBUCION DEL ESFUERZO EN UN PROYECTO DE SOFTWARE

El ciclo de vida de un proyecto de software está dividido en varias etapas. Cada una comprende un cierto porcentaje de esfuerzo en donde conjuntamente conforman lo que se denomina el proyecto en sí.

Cada etapa va a requerir de una porción de esfuerzo distinta a las demás las cuales son detalladas a continuación:

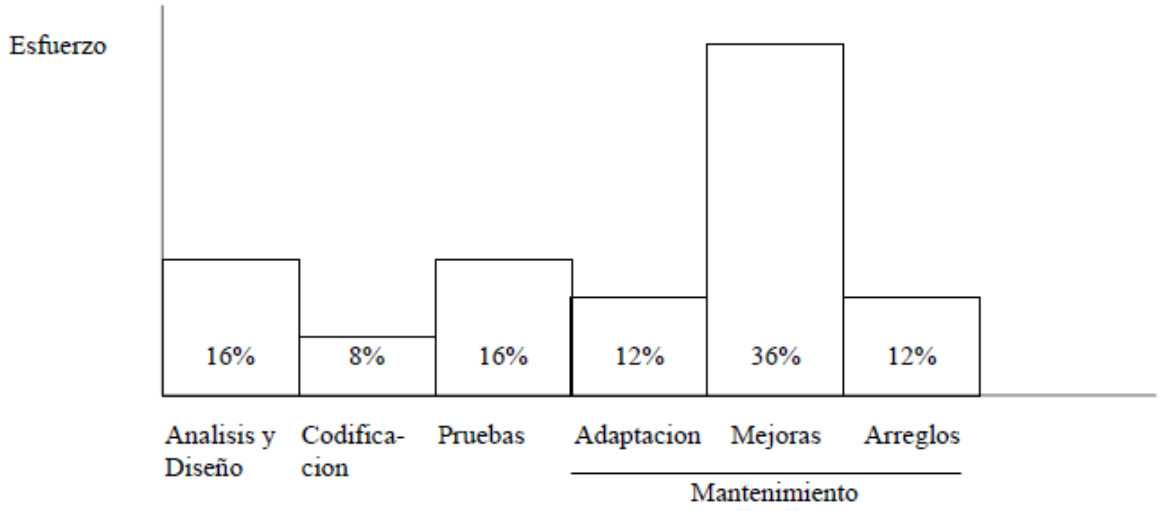

Figura Nº 2 – Distribución porcentual del esfuerzo de un proyecto de software.

Como se puede ver en la Figura Nº 2 el mantenimiento está constituido por todas las actividades posteriores a la liberación inicial del producto, las cuales son, el mejoramiento de las capacidades del producto, adaptación del producto a nuevos ambientes de cómputo y la depuración de errores.

- Gran porcentaje del esfuerzo total se dedica a mejorar el producto
- Asignar poco tiempo a las pruebas piloto y de aceptación es una de las razones de sobrepasar el costo y tiempo de entrega de un producto
- El mantenimiento gasta más recursos que las actividades de desarrollo.

Por lo tanto, se puede decir que el mantenimiento de un software se lleva la mayor porción de la vida de un sistema, ya que este se va a mantener durante la existencia del sistema hasta que se vuelva obsoleto.

Figura Nº 2 Extraída de [2]

3.4. ADMINISTRACION DE PROYECTOS DE SOFTWARE

3.4.1.INTRODUCCION

Las actividades técnicas y gerenciales son igualmente importantes para el éxito de un proyecto de programación. Las actividades de la administración de un proyecto comprenden los métodos para organizar y seguir el curso del proyecto; estimación de costos, políticas de asignación de recursos, control de presupuesto, determinación de avances, ajustes al calendario de trabajo, procedimientos de control de calidad, comunicación con el cliente, etc.

La administración de proyectos de desarrollo de software consiste en gestionar el desarrollo de un producto, dentro del plazo previsto y con los fondos establecidos. Como esto requiere recursos humanos, la administración del proyecto involucra no sólo la organización técnica y las habilidades organizativas, sino también el arte de dirigir un equipo de personas. La administración de un proyecto no es una actividad menor, puede ser tan transcendente como desarrollar la arquitectura.

La administración de un proyecto comprende:

- Estructura (Elementos organizativos involucrados)
- Proceso administrativo (Responsabilidades y supervisión de participantes)
- Proceso de desarrollo (métodos, herramientas, lenguajes, documentación y apoyo)
- Programa (organización de los tiempos en los que deben realizarse los trabajos)

Algunos problemas importantes identificados en la administración de software son:

- Planeación de proyectos de software pobres.
- Procedimientos de selección de gerentes de proyecto pobres.
- La medición de proyectos es pobre.
- Falta de procedimientos para vigilar el avance del proyecto.
- Falta de estándares para medir la calidad del desempeño y cantidad de producción esperada.

Algunos métodos sugeridos para solucionar estos problemas son:

- Entrenar y educar a la dirección, jefes de proyecto y constructores.
- Obligar al uso de estándares, procedimientos y documentación.
- Definir objetivos de la calidad deseada.
- Desarrollar estimaciones de calendario y costos de forma exacta y verdadera.
- Seleccionar jefes de proyecto basados en su capacidad para administrar proyectos más que en su habilidad técnica.

3.4.2.FACTORES DE LA ADMINISTRACION DE UN PROYECTO

La administración de un proyecto debe controlar los siguientes factores:

- El costo total del proyecto (aumentar o disminuir los gastos).
- Las capacidades del proyecto (añadir o eliminar características funcionales).
- La calidad del producto (aumentar el tiempo entre fallos de una cierta severidad).
- La duración del proyecto (reducir el tiempo programado un 20% o posponer un mes la fecha de terminación).

La calidad, la capacidad, los costos y los tiempos de realización son magnitudes que hay que gestionar a los largo de un proyecto. El grado en el que estos cuatro factores pueden controlarse depende de la naturaleza del proyecto y de quien o quienes los administra.

- Aunque los costos pueden estar prefijados de antemano, frecuentemente se dispone de flexibilidad, ya que en la práctica muy rara vez se cumple con los costos establecidos en una primera instancia.
- La capacidad del proyecto puede renegociarse en función de la evolución del proyecto.
- La calidad también puede variar. Cuando la calidad se establece baja, se disminuye los costos de corto plazo, pero se incrementan los costos de largo plazo debido al costo de mantenimiento y la insatisfacción de los clientes. Si se establece una calidad excesiva, el costo de desarrollo se puede hacer inaguantable.
- Negociar el tiempo frente a cualquiera de las otras magnitudes es también algo habitual.

3.4.3.SECUENCIA DE ACTIVIDADES DE ADMINISTRACIÓN DE UN PROYECTO

- Comprender el contenido, alcance y tiempos del proyecto.

Se refiere sólo a un entendimiento global de los objetivos del proyecto y no en reunir los requisitos que es función de los técnicos.

- Identificar el proceso de desarrollo (métodos, herramientas, lenguajes, documentación, ayudas).

Es la decisión de qué metodología de desarrollo usar (cascada, espiral, por incrementos, etc.)

- Determinar la estructura organizativa (elementos de la organización involucrados).

Esto incluye identificar las unidades, departamentos, compañías, líderes disponibles, etc. Una vez identificadas las partes y sus capacidades hay que decidir cómo deben interactuar para realizar el trabajo.

- Identificar el proceso administrativo (establecer las responsabilidades de los participantes).

Esto incluye determinar quién reportará a quién e identificar el modelo de organización

- Programar el proceso (organigramas en los que se fijan los tiempos de ejecución de cada actividad).

Programar que actividades deben realizarse y en qué tiempo.

- Establecer un equipo de personas (se busca y contrata el equipo de personas).

Se debe complementar la dotación de personal de acuerdo con las actividades que debe ejecutar cada grupo

- Analizar los riesgos y buscar sus paliativos.

Los aspectos negativos que ocurren sin ser esperados, son las principales causas de que los proyectos fracasen. La identificación de los riegos y la búsqueda preventiva de soluciones es una garantía de éxito del proyecto.

- Enumerar los productos que debe generar el proyecto.

Antes de iniciarse el proyecto desde el punto de vista técnico debe establecerse sobre el organigrama los productos de documentación o de código que deben generarse.

3.4.4. VALORES DEL CAPITAL HUMANO

El ingrediente principal para producir software es el equipo humano:

- Profesionalidad: cumplir con las responsabilidades sociales.
- Trabajo en equipo: organización de las funciones e interacciones.
- Liderazgo: marca la dirección del trabajo basado en la experiencia.

El ingrediente principal requerido para producir software de alta calidad es la gente. Para esto se cuenta con las actitudes de los ingenieros y también con la coordinación en el tiempo para realizar el proyecto. Esto requiere una combinación de profesionalidad, trabajo en equipo y liderazgo.

3.4.5.ORGANIZACIÓN DE UN EQUIPO

1. Se selecciona un líder
   - Asegura que se activen todos los aspectos del proyecto.
   - Resuelve las diferencias.
   - Propone las primeras tentativas.
   - Busca que el equipo lo acepte.

2. Se designan y documentan las responsabilidades
   - Líder del equipo: Propone y mantiene.
   - Responsable de gestión de la configuración.
   - Responsable de calidad.
   - Responsable de administración de requisitos.
   - Responsable de diseño.
   - Responsable de implementación.

3. Designar y respaldar a cada responsable
   - Cada responsable debe estar respaldado por otro, que lo suple en caso de baja o ausencia.

3.5. RECURSOS

3.5.1.RECURSOS HUMANOS Y PARTICIPANTES

Son todas aquellas personas que intervienen y participan activamente en el ciclo de vida del desarrollo del software desde cualquier instancia del proyecto. El número de personas requerido para un proyecto de software sólo puede ser determinado después de realizar una estimación del esfuerzo de cada etapa implicada en el mismo (Analistas, Desarrolladores, PM, Líder técnico, Arquitectos, etc.).

3.5.2.RECURSOS DE SOFTWARE

Son aquellos componentes de un software que son usados en otras aplicaciones de la misma índole, ya sea para reducir costos o tiempo (IDE, API, Herramientas de gestión, etc.).

3.5.3.RECURSOS DE ENTORNO

Es el entorno de las aplicaciones (hardware) el cual proporciona el medio físico para desarrollar las aplicaciones (software), esto hace que este recurso es indispensable.

3.6. CICLO DE VIDA DE UN PROYECTO DE SISTEMA

El ciclo de vida de un proyecto de sistema está dividido en varias etapas. Cada etapa describe las actividades que hay que realizar para obtener un conjunto concreto de productos de desarrollo del software, las cuales se nombran a continuación:

3.6.1. RECOPILACION DE LOS REQUERIMIENTOS

En esta primera etapa se realiza una recopilación y/o encuesta con el cliente, la cual nos permite obtener una visión de alto nivel sobre el proyecto, poniendo énfasis en la descripción del problema desde el punto de vista de los clientes y desarrolladores. También se considera la posibilidad de una planificación de los recursos sobre una escala de tiempos.

3.6.2. ANALISIS

El propósito principal de esta etapa es conseguir una comprensión más precisa de los requisitos y una descripción de los mismos que sea fácil de mantener y que nos ayude a estructurar el sistema completo, incluyendo la arquitectura.

El análisis de requerimientos facilita al ingeniero de sistemas especificar la función y comportamiento de los programas, indicar la interfaz con otros elementos del sistema y establecer las ligaduras de diseño que debe cumplir el programa.

También permite al ingeniero refinar la asignación de software y representar el dominio de la información que será tratada por el programa. Así como también brinda al diseñador la representación de la información y las funciones que pueden ser traducidas en datos, arquitectura y diseño procedimental.

Finalmente, la especificación de requerimientos suministra al técnico y al cliente, los medios para valorar la calidad de los programas, una vez que se haya construido.

En la medida que logremos una clara comprensión de lo anterior obtendremos una arquitectura estable y sólida que nos facilite una comprensión en profundidad de los requisitos.

3.6.3. LIMITACIONES

En esta etapa se va a detallar la frontera del proyecto, es decir, cuál es el alcance de nuestro sistema.

Todo cambio que esté fuera de las limitaciones se deberá tratar como un cambio de alcance en la etapa de mantenimiento.

3.6.4.ESPECIFICACIONES

La obtención de especificaciones a partir del cliente u otros actores intervinientes es un proceso humano muy interactivo e iterativo. Normalmente a medida que se captura la información, se la analiza y realimenta con el cliente, refinándola, puliéndola y corrigiendo si es necesario. El analista siempre debe llegar a conocer la temática y el problema a resolver, dominarlo, hasta cierto punto, hasta el ámbito que el futuro sistema a desarrollar lo abarque. Por ello el analista debe tener alta capacidad para comprender problemas de muy diversas áreas o disciplinas de trabajo. El analista se debe compenetrar con el área de negocio del cliente, para comprender cómo ella trabaja y maneja su información, desde niveles muy bajos e incluso llegando hasta los gerenciales.

Dada la gran diversidad de campos a cubrir, los analistas suelen ser asistidos por especialistas o usuarios/clientes, es decir, gente que conoce profundamente el área para la cual se desarrollará el software.

Al contrario de los analistas, los clientes no tiene por qué saber nada de software, ni de diseños, ni otras cosas relacionadas, sólo se debe limitar a aportar objetivos, datos e información (de mano propia o de sus registros, equipos, empleados, etc.) al analista, y guiado por él, para que, en primera instancia defina un documento funcional y/o caso de uso.

3.6.5.DISEÑO Y ARQUITECTURA

Una vez realizada la etapa de análisis y especificación, se modela el sistema y definimos su estructura (incluida la arquitectura) para que soporte todos los requisitos, incluyendo los requisitos no funcionales y otras restricciones.

Con toda esta información recopilada en los puntos anteriores, los profesionales técnicos traducen la información de alto nivel, en esquemas, diagramas, etc. de bajo nivel, para luego éstos ser comprendidos por el área de desarrollo.

3.6.6.PROGRAMACION

Convertir un diseño a código puede ser interpretada como la parte más obvia e importante del trabajo de ingeniería de software, pero no necesariamente es la que demanda mayor trabajo y ni la más complicada. La complejidad y la duración de esta etapa está íntimamente relacionada al o a los lenguajes de programación utilizados, así también como a la calidad del diseño previamente realizado.

3.6.7.PRUEBAS DE SOFTWARE

La prueba del software es un elemento crítico para la garantía de la calidad del software. El objetivo de la etapa de pruebas es garantizar la calidad del producto desarrollado.

Esta etapa implica:

- Verificar la interacción de componentes.
- Verificar la integración adecuada de los componentes.
- Verificar que todos los requisitos se han implementado correctamente.
- Identificar y asegurar que los defectos encontrados se han corregido antes de entregar el software al cliente.
- Diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.

La prueba no es una actividad sencilla, no es una etapa del proyecto en la cual se asegura la calidad, sino que la prueba debe ocurrir durante todo el ciclo de vida. Podemos probar la funcionalidad de los primeros prototipos, la estabilidad, cobertura y rendimiento de la arquitectura, probar el producto final, etc.

La prueba es un proceso que se enfoca sobre la lógica interna del software y las funciones externas. La prueba es un proceso de ejecución de un programa con la intención de descubrir un error que probablemente no fue previsto en las fases iníciales del desarrollo del software.

Tipos de pruebas:

- Pruebas de unidad: La prueba de unidad se centra en el módulo. Usando la descripción del diseño detallado como guía, se prueban los caminos de control importantes con el fin de descubrir errores dentro del ámbito del módulo. La prueba de unidad hace uso intensivo de las técnicas de prueba de caja blanca. Este tipo de prueba la realiza, generalmente, el desarrollador luego de la codificación de un módulo.

- Prueba de integración: El objetivo es tomar los módulos testeados en la prueba de unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.
  Hay dos formas de integración:
  - Integración no incremental: Se combinan todos los módulos por anticipado y se prueba todo el programa en conjunto.
  - Integración incremental: El programa se construye y se prueba en pequeños segmentos.

  En la prueba de integración el foco de atención es el diseño y la construcción de la arquitectura del software.
  Las técnicas que más prevalecen son las de diseño de casos de prueba de caja negra, aunque se pueden llevar a cabo unas pocas pruebas de caja blanca.

- Prueba del sistema: verifica que cada elemento encaja de forma adecuada y que se alcanza la funcionalidad y el rendimiento del sistema total. La prueba del sistema está constituida por una serie de pruebas diferentes cuyo propósito primordial es ejercitar profundamente el sistema basado en computadora. Algunas de estas pruebas son:

- Prueba de validación: proporciona una seguridad final de que el software satisface todos los requerimientos funcionales y de rendimiento. Además, valida los requerimientos establecidos comparándolos con el sistema que ha sido construido. Durante la validación se usan exclusivamente técnicas de prueba de caja negra.
- Prueba de recuperación: se fuerza a un fallo del software y se verifica que la recuperación se lleva a cabo apropiadamente.
- Prueba de seguridad: verificar los mecanismos de protección.
- Prueba de resistencia: se somete el sistema a operaciones anormales.
- Prueba de rendimiento: prueba el rendimiento del software en tiempo de ejecución.
- Prueba de instalación: se centra en asegurar que el sistema software desarrollado se puede instalar en diferentes configuraciones hardware y software y bajo condiciones excepciones, por ejemplo con espacio de disco insuficiente o continuas interrupciones.

- Pruebas de regresión: Las pruebas de regresión son una estrategia de prueba en la cual las pruebas que se han ejecutado anteriormente se vuelven a realizar en la nueva versión modificada, para asegurar la calidad después de añadir la nueva funcionalidad. El propósito de estas pruebas es asegurar que los defectos identificados en la ejecución anterior de la prueba se hayan corregido, que los cambios realizados no introduzcan nuevos o viejos defectos.
  La prueba de regresión puede implicar la re-ejecución de cualquier tipo de prueba.
  Normalmente, las pruebas de regresión se llevan a cabo durante cada iteración, ejecutando otra vez las pruebas de la iteración anterior.

3.6.8.IMPLEMENTACION

Dentro del ciclo de vida se encuentra la fase de implementación de un sistema, es la fase más costosa y que consume más tiempo, se dice que es costosa porque muchas personas, herramientas y recursos, están involucrados en el proceso y consume mucho tiempo porque se completa todo el trabajo realizado previamente durante el ciclo de vida.

En la fase de implementación se instala el nuevo sistema de información para que empiece a trabajar y se capacita a sus usuarios para que puedan utilizarlo.

La instalación puede realizarse según cuatro métodos: Directo, paralelo, piloto y en fases.

- Método directo: Se abandona el sistema antiguo y se adopta inmediatamente el nuevo. Esto puede ser sumamente riesgoso porque si algo no funciona correctamente, es imposible volver al sistema anterior y las correcciones deberán hacerse bajo la marcha. Regularmente con un sistema nuevo suelen surgir problemas de pequeña y gran escala. Si se trata de grandes

sistemas, un problema puede significar una catástrofe, perjudicando o retrasando el desempeño entero de la organización.

- Método paralelo: Los sistemas de información antiguo y nuevo operan juntos hasta que el nuevo demuestra ser confiable. Este método es de bajo riesgo. Si el sistema nuevo falla, la organización puede mantener sus actividades con el sistema antiguo. Esto puede representar un alto costo al requerir contar con personal y equipo para laborar con los dos sistemas, por lo que este método se reserva específicamente para casos en los que el costo de una falla sería muy considerable.

- Método piloto: Pone a prueba el nuevo sistema sólo en una parte de la organización. Al comprobar su efectividad, se implementa en el resto de la organización. El método es menos costoso que el paralelo, aunque más riesgoso. En este caso el riesgo es controlable al limitarse a ciertas áreas, sin afectar toda la empresa.

- Método en fases: La implementación del sistema se divide en partes o fases, que se van realizando a lo largo de un periodo de tiempo, sucesivamente. Una vez iniciada la primera fase, la segunda no se inicia hasta que la primera se ha completado con éxito. Así se continúa hasta que se finaliza con la última fase. Es costoso porque se hace más lenta la implementación, pero sin duda tiene el menor riesgo.

3.6.9.DOCUMENTACION

Es la guía o comunicación escrita en sus diferentes formas, ya sea en modelaciones (UML), modelado de negocio, RUP, diagramas, pruebas, manuales de usuario, manuales técnicos, etc., todo con el propósito de eventuales correcciones, utilización, mantenimiento futuro y ampliaciones al sistema.

La importancia de la documentación radica en que a menudo un sistema escrito por una persona es modificado por otra. Es por ello que la documentación sirve para facilitar la etapa de mantenimiento.

La documentación se compone de tres partes:

- Documentación Interna: Son los comentarios o mensajes que se añaden al código fuente para hacer más claro el entendimiento de los procesos que lo conforman, incluyendo las precondiciones y las pos condiciones de cada función.

- Documentación Externa: se define en un documento escrito con los siguientes puntos:
    - Descripción del Problema
    - Datos del Autor
    - Algoritmo (diagrama de flujo o Pseudocódigo)

- Diccionario de Datos
- Código Fuente (programa)

- Manual de Usuario: Describe paso a paso la manera cómo funciona el programa, con el fin de que el usuario lo pueda manejar para que obtenga el resultado deseado.

3.6.10. MANTENIMIENTO

El mantenimiento consiste en mantener y mejorar el software para enfrentar errores descubiertos y nuevos requisitos. Esto puede llevar más tiempo incluso que el desarrollo inicial del software.

Una pequeña parte de este trabajo consiste en arreglar errores o mayormente conocidos como ***Bugs***. La mayor parte consiste en extender el sistema para que sea más útil y completo.

La fase de mantenimiento de software es una parte explícita del modelo en cascada del proceso de desarrollo de software el cual fue desarrollado durante el movimiento de programación estructurada en computadores. El otro gran modelo, el Desarrollo en espiral desarrollado durante el movimiento de ingeniería de software orientada a objeto no hace una mención explícita de la fase de mantenimiento.

En un ambiente formal de desarrollo de software, la organización o equipo de desarrollo tendrán algún mecanismo para documentar y rastrear defectos y deficiencias. El Software tan igual como la mayoría de otros productos, es típicamente lanzado con un conjunto conocido de defectos y deficiencias.

Las deficiencias conocidas son normalmente documentadas en una carta de consideraciones operacionales o notas de lanzamiento (release notes) es así que los usuarios del software serán capaces de trabajar evitando las deficiencias conocidas y conocerán cuando el uso del software sería inadecuado para tareas específicas.

***Bugs**: Defecto de software, es el resultado de un fallo o deficiencia durante el proceso de creación de programas de software.*

4. METODOLOGIAS CLASICAS

4.1. INTRODUCCION

En febrero de 2001, tras una reunión celebrada en Utah-EEUU, nace el término ágil aplicado al desarrollo de 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 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 es fue el manifiesto Ágil, un documento que resume la filosofía ágil.

Principales ideas de la metodología ágil:

- Se encarga de valorar al individuo y las iteraciones del equipo más que a las herramientas o los procesos utilizados.
- Se hace mucho más importante crear un producto software que funcione, que escribir mucha documentación.
- El cliente está en todo momento colaborando en el proyecto.
- Es más importante la capacidad de respuesta ante un cambio realizado que el seguimiento estricto de un plan.

4.2. VENTAJAS Y DESVENTAJAS

Ventajas:

- La primera y más importante, es que estas metodologías ofrecen una rápida respuesta a cambios de requisitos a lo largo del desarrollo del proyecto gracias a su proceso iterativo, es tan importante realizar una buena recolecta de requisitos, como después poder modificarlos evitando grandes pérdidas en cuanto a costes, motivación, tiempo.
- El cliente, si quiere colaborar, puede observar cómo va avanzando el proyecto, y por supuesto, opinar sobre su evolución gracias a las numerosas reuniones que realiza el equipo con el cliente. Esto le da tranquilidad.
- Uniendo las dos anteriores, se puede deducir que al utilizar estas metodologías, los cambios que quiera realizar el cliente van a tener un menor impacto, ya que se va a entregar en un

pequeño intervalo de tiempo un pequeño "trozo" del proyecto al cliente, y si éste quiere cambiarlo, solo se habrá perdido unas semanas de trabajo. Con las metodologías tradicionales las entregas al cliente se realizaban tras la realización de una gran parte del proyecto, eso quiere decir que el equipo ha estado trabajando meses para que luego un mínimo cambio que quiera realizar el cliente, conlleve la pérdida de todo ese trabajo.

- Importancia de la simplicidad al eliminar trabajo innecesario

Otros beneficios de las metodologías ágiles.

- **Simplificación de la sobrecarga de procesos**

Los equipos que trabajan para crear productos regulados por estándares de la industria, deben demostrar el cumplimiento de esas normas. Estos equipos suelen adoptar importantes sobrecargas de trabajo para asegurarse de que cumplen con los estrictos mandatos de código. Las metodologías ágiles pueden ayudarles a cumplir los estándares de la industria con menos sobrecarga utilizando iteraciones más cortas y empaquetadas. El beneficio neto es un proceso que:

  - Puede adaptarse a los cambios que, inevitablemente, surgirán
  - Requiere menos sobrecarga en el proceso end-to-end
  - Implica menos trabajo a medida que se acerca la fecha final

- **Calidad mejorada**

Las prácticas de desarrollo ágil proporcionan la funcionalidad suficiente como para satisfacer las expectativas de los accionistas con una alta calidad. Piensa en lo que significa "ofrecer lo suficiente". Eso es, proporcionar la mínima funcionalidad con la máxima calidad. La mínima funcionalidad no implica necesariamente una pobre funcionalidad, implica lo suficiente como para conseguir que el trabajo se realice. Los accionistas suelen pensar que saben lo que quieren cuando especifican altos niveles de requerimientos para un producto TI o de software. Sin embargo, la mayoría de las veces, cuando ven el producto final, éste no resuelve el problema. Simplemente, no se han imaginado de forma precisa el mismo, o el problema ha cambiado o, incluso, la tecnología no era tan buena como parecía. También puede ser que el producto no funcione realmente del modo en que las partes interesadas tenían intención, incluso aunque pensaran que habían descrito los requerimientos de manera clara. Desarrollar iteraciones en poco tiempo y demostrar a los accionistas los productos pronto y con frecuencia, permite tanto a los accionistas como a los equipos de desarrollo ponerse de acuerdo y coincidir en que el producto cumple con cada una de sus necesidades.

- **Mejorar la previsibilidad a través de una mejor gestión del riesgo**

Cuando los equipos de desarrollo no cumplen con sus fechas de lanzamiento, a menudo se debe a muchas razones completamente justificables. Puede ser que el equipo no hubiera entendido bien lo difícil que sería utilizar la nueva tecnología; los requerimientos podían no estar del todo claros o los clientes cambiaron de opinión cuando el trabajo estaba prácticamente finalizado. En cualquier caso, los negocios demandan productos que cumplan con las fechas de entrega, de modo que los planes de negocio directamente relacionados con ellos puedan cumplirse. Hay muchas formas en las que las metodologías ágiles pueden ayudar a que los proyectos TI cumplan con las fechas de lanzamiento previstas.

  - **Dar prioridad a los riegos**. Las prácticas ágiles priorizan los aspectos de desarrollo de alto riesgo permitiendo así una reducción de los riesgos iniciales.

  - **Evaluación de riesgos en paralelo.** Para áreas de riesgo donde debería haber múltiples soluciones y el equipo no se pone de acuerdo a la hora de tomar el camino más adecuado, se debe tener en cuenta la posibilidad de optar por el desarrollo multi-set. Esto requiere que diferentes equipos trabajen en paralelo resolviendo el mismo problema con diferentes soluciones. La mayoría de los equipos no considerarán ni tan siquiera esta forma de trabajar, porque están convencidos de que el tiempo y el coste requeridos para hacer una evaluación paralela son demasiado grandes. De hecho, el desarrollo paralelo de alternativas es probable que traiga consigo las decisiones clave a seguir.

- **Mejor perfil de productividad**

Los equipos ágiles son más productivos que aquellos que utilizan métodos tradicionales a lo largo de todo el ciclo de desarrollo. El desarrollo tradicional suele mostrar un patrón de trabajo parecido a un palo de hockey, empezando con un ciclo de diseño de abajo arriba, moviéndose hasta una fase de prototipo para pasar luego a un ciclo de desarrollo largo, seguido por otro ciclo totalmente impredecible en el que se integran las piezas para pasar, por último, a la fase de prueba final. A medida que el proyecto progresa, los equipos tienen que trabajar juntos de manera más coherente, confiando en que todas las piezas trabajen juntas tal y como esperan. Pero lo cierto es que esto raramente ocurre, de modo que la interacción entre los equipos aumenta a medida que lo hacen los problemas de integración. Por último, la fase de pruebas lleva todo esto al límite. Trabajando juntos como un único equipo en fases verticales del producto desde el principio del ciclo de producción, se evita el ciclo de productividad tradicional de palo de hockey. Los equipos ágiles tienden a ser muy productivos desde la primera iteración hasta su lanzamiento y su ritmo tiene que ser gestionado de modo que no se produzca agotamiento. Los equipos ágiles que mantienen este código de trabajo con cada iteración, permiten realizar pruebas de rendimiento y sistemas desde el principio, pudiendo empezar en las primeras iteraciones. De este modo, defectos críticos como problemas de integración se descubren

antes, la calidad general del producto es mayor y el equipo funciona de manera más productiva durante todo el ciclo de desarrollo.

- **Capacidad para aprovechar las inversiones realizadas**

Adoptar las técnicas ágiles de manera satisfactoria precisa un importante soporte de herramientas. La mayoría de los equipos invierten fuertemente en buenas herramientas de software y necesitan elevar esa inversión y reducir la cantidad de cambios cuando empiezan a adoptar las metodologías ágiles. La inversión más crítica es una buena planificación y una herramienta de gestión del trabajo que proporcione una cartera de pedidos del equipo visible y que asocie el trabajo con cada elemento de trabajo en cartera. Idealmente, esa herramienta de planificación se integra con otras herramientas de desarrollo, lo que permite a los equipos mantener la trazabilidad de la cartera de pedidos en otros aspectos, incluyendo:

  - **Los requerimientos que la impulsan.**
  - **La arquitectura bajo desarrollo.**
  - **El software de la solución.**
  - **Las pruebas que validan la solución.**

Las herramientas de Rational trabajan muy bien juntas y con otros productos para ayudar a entregar con éxito proyectos ágiles. IBM Rational Team Concert es el componente central de colaboración y flujo de trabajo de la solución IBM Rational para ingeniería de software y sistemas. Proporciona una interfaz Open Services for Lifecycle Collaboration (OSLC) que permite a las herramientas capacitadas OSLC integrarse sin problemas. Rational Team Concert proporciona al equipo planificación ágil, visible y probada. De hecho, su adopción para simplemente ese propósito ha sido viral en IBM Software Group. Rational Team Concert proporciona gestión de elementos, planificación de equipos asociada a cargas de trabajo y visibilidad de proyectos, todos ellos, elementos críticos en los proyectos ágiles. Los elementos de trabajo en Rational Team Concert son muy flexibles en su uso y pueden asociarse con lanzamientos, equipos y demás.

Desventajas:

- Falta de documentación del diseño. Al no haber documentación es el código (junto con sus comentarios) lo que se toma como documentación.
- Problemas derivados de la comunicación oral. No hace falta decir que algo que está escrito “no se puede borrar”, en cambio, algo dicho es muy fácil crear ambigüedad.
- Fuerte dependencia de las personas.
- Falta de reusabilidad derivada de la falta de documentación
- Restricciones en cuanto a tamaño de los proyectos
- Problemas derivados del fracaso de los proyectos ágiles. Si un proyecto ágil fracasa no hay documentación o hay muy poca; lo mismo ocurre con el diseño. La comprensión del sistema se queda en las mentes de los desarrolladores.

4.3. TIPOS DE METODOLOGIAS

4.3.1. WATERFALL (CASCADA)

En Ingeniería de software el desarrollo en cascada, es denominado así por la posición de las fases en el desarrollo de esta, que parecen caer en cascada "por gravedad" hacia las siguientes fases. Es el enfoque metodológico que ordena rigurosamente las etapas del proceso para el desarrollo de software, de tal forma que el inicio de cada etapa debe esperar a la finalización de la etapa anterior. Al final de cada etapa, el modelo está diseñado para llevar a cabo una revisión final, que se encarga de determinar si el proyecto está listo para avanzar a la siguiente fase. Este modelo fue el primero en originarse y es la base de todos los demás modelos de ciclo de vida.

Este modelo comenzó a diseñarse en 1966 y se terminó alrededor de 1970. El principal problema de esta aproximación es el que no podemos esperar que las especificaciones iníciales sean correctas y completas y que el usuario puede cambiar de opinión sobre una u otra característica.

Además los resultados no se pueden ver hasta muy avanzado el proyecto por lo que cualquier cambio debido a un error puede suponer un gran retraso además de un alto coste de desarrollo.

Como es evidente esto es solo un modelo teórico, si el usuario cambia de opinión en algún aspecto tendremos que volver hacia atrás en el ciclo de vida.

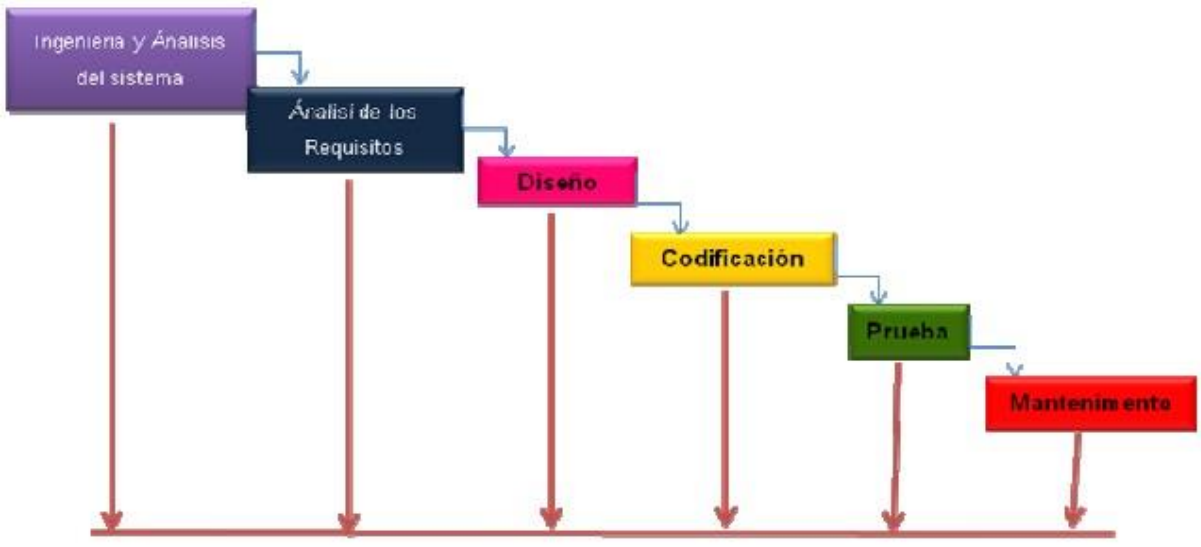

Figura Nº 3 – Modelo del ciclo de vida en Cascada (Waterfall).

Figura Nº 3 Extraída de [3]

Ingeniería y Análisis del Sistema: debido que el software es siempre parte de un sistema mayor, el trabajo comienza estableciendo los requisitos de todos los elementos del sistema y luego asignando algún subconjunto de estos requisitos al software.

Análisis de los requisitos del software: el proceso de recopilación de los requisitos se centra e intensifica especialmente en el software. El ingeniero de software (Analistas) debe comprender el ámbito de la información del software, así como la función, el rendimiento y las interfaces requeridas.

Diseño: el diseño del software se enfoca en cuatro atributos distintos del programa: la estructura de los datos, la arquitectura del software, el detalle procedimental y la caracterización de la interfaz. El proceso de diseño traduce los requisitos en una representación del software con la calidad requerida antes de que comience la codificación.

Codificación: el diseño debe traducirse en una forma legible para la máquina. El paso de codificación realiza esta tarea. Si el diseño se realiza de una manera detallada la codificación puede realizarse mecánicamente.

Prueba: una vez que se ha generado el código comienza la prueba del programa. La prueba se centra en la lógica interna del software, y en las funciones externas, realizando pruebas que aseguren que la entrada definida produce los resultados que realmente se requieren.

Mantenimiento: el software sufrirá cambios después de que se entrega al cliente. Los cambios ocurrirán debido a se han encontrado errores, a que el software deba adaptarse a cambios del entorno externo (sistema operativo o dispositivos periféricos), o debido a que el cliente requiera ampliaciones funcionales o del rendimiento.

Desventajas:

- Los proyectos reales raramente siguen el flujo secuencial que propone el modelo, siempre hay iteraciones y se crean problemas en la aplicación del paradigma.

- Normalmente, es difícil para el cliente establecer explícitamente al principio todos los requisitos. El ciclo de vida clásico lo requiere y tiene dificultades en acomodar posibles incertidumbres que pueden existir al comienzo de muchos productos.

- El cliente debe tener paciencia. Hasta llegar a las etapas finales del proyecto, no estará disponible una versión operativa del programa. Un error importante no detectado hasta que el programa esté funcionando puede ser desastroso.

4.3.2. PROTOTYPING

Un prototipo es una versión preliminar de un sistema de información con fines de demostración o evaluación.

El prototipo de requerimientos es la creación de una implementación parcial de un sistema, para el propósito explícito de aprender sobre los requerimientos del sistema. Un prototipo es construido de una manera rápida tal como sea posible.

Esto es dado a los usuarios, clientes o representantes de ellos, posibilitando que ellos experimenten con el prototipo. Estos individuos luego proveen la retroalimentación sobre lo que a ellos les gustó y no les gustó acerca del prototipo proporcionado, quienes capturan en la documentación actual de la especificación de requerimientos la información entregada por los usuarios para el desarrollo del sistema real. El prototipo puede ser usado como parte de la fase de requerimientos (determinar requerimientos) o justo antes de la fase de requerimientos (como predecesor de requerimientos). En otro caso, el prototipo puede servir su papel inmediatamente antes de algún o todo el desarrollo incremental en modelos incremental o evolutivo.

El prototipo ha sido usado frecuentemente en los 90, porque la especificación de requerimientos para sistemas complejos tiende a ser relativamente dificultoso de cursar. Muchos usuarios y clientes encuentran que es mucho más fácil proveer retroalimentación convenientemente basada en la manipulación, leer una especificación de requerimientos potencialmente ambigua y extensa.

Diferente del modelo evolutivo donde los requerimientos mejor entendidos están incorporados, un prototipo generalmente se construye con los requerimientos entendidos más pobremente.

En caso que ustedes construyan requerimientos bien entendidos, el cliente podría responder con "sí, así es", y nada podría ser aprendido de la experiencia.

- Es un método menos formal de desarrollo.
- El prototipo es una técnica para comprender las especificaciones.
- Un prototipo puede ser eliminado.
- Un prototipo puede llegar a ser parte del producto final.

Ventajas:

- Útiles cuando los requerimientos son cambiantes.
- Cuando no se conoce bien la aplicación.
- Cuando el usuario no se quiere comprometer con los requerimientos.
- Cuando se quiere probar una arquitectura o tecnología.
- Cuando se requiere rapidez en el desarrollo.

Desventajas:

- No se conoce cuando se tendrá un producto aceptable.
- No se sabe cuántas iteraciones serán necesarias.
- Da una falsa ilusión al usuario sobre la velocidad del desarrollo.
- Se puede volver el producto aún y cuando no esté con los estándares.

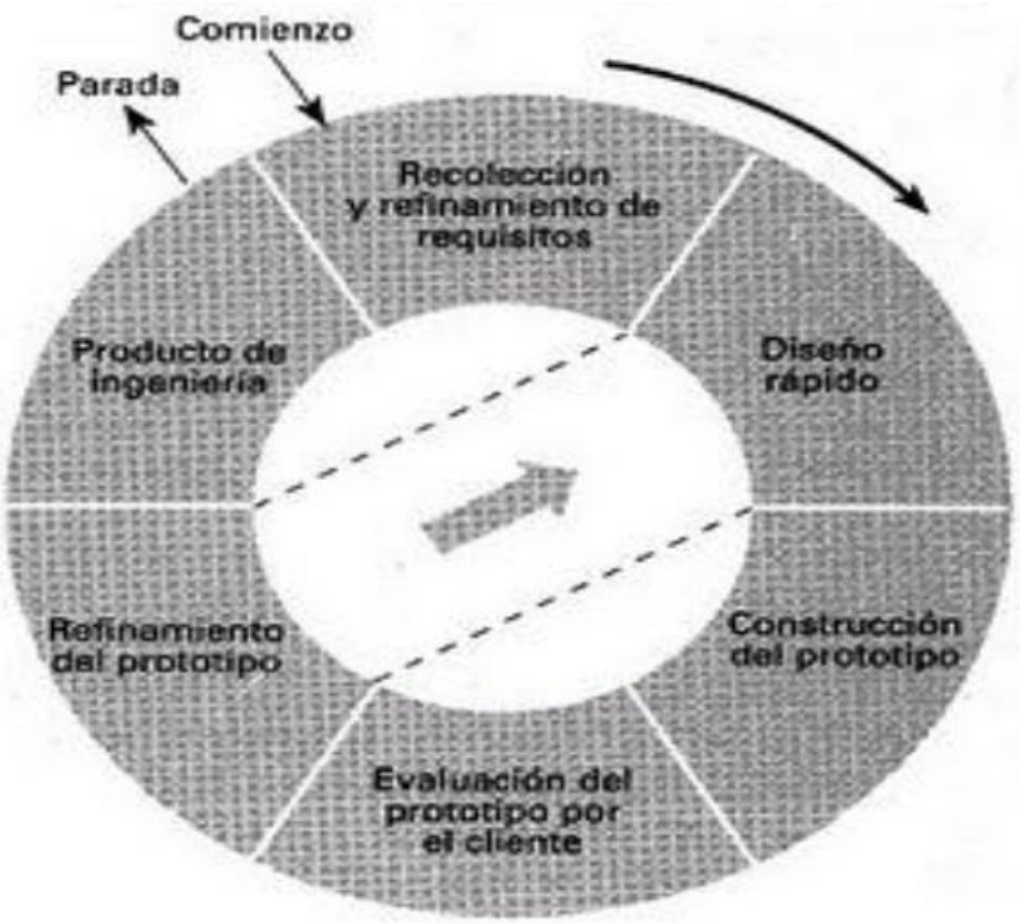

Figura Nº 4 – Modelo del ciclo de vida Prototipo.

Figura Nº 4 Extraída de [4]

4.3.3. SPIRAL

Toma las ventajas del modelo de desarrollo en cascada y el de prototipos añadiéndole el concepto de análisis de riesgo.

Se definen cuatro actividades:

Planificación: en la que se recolectan los requisitos iniciales o nuevos requisitos a añadir en esta iteración.

Análisis de riesgo: basándonos en los requisitos decidimos si somos capaces o no de desarrollar el software y se toma la decisión de continuar o no continuar.

Ingeniería: en el que se desarrolla un prototipo basado en los requisitos obtenidos en la fase de planificación.

Evaluación del cliente: el cliente comenta el prototipo. Si está conforme con él se acaba el proceso, si no se añaden los nuevos requisitos en la siguiente iteración.

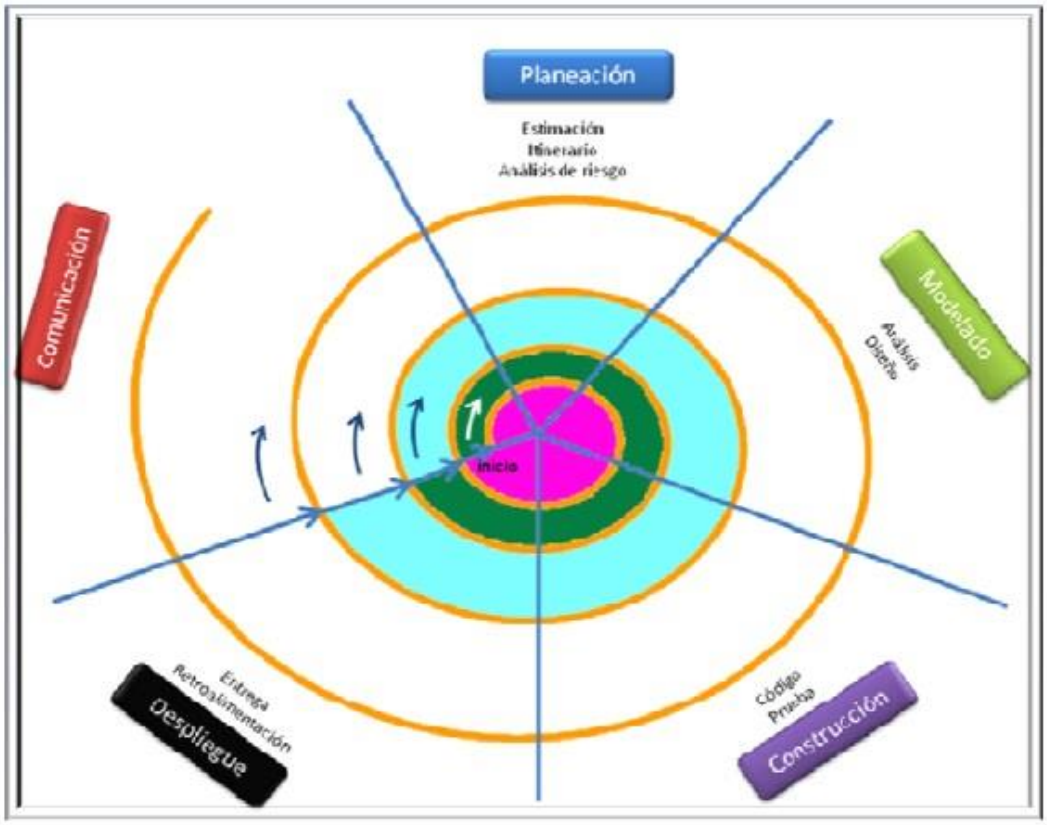

Figura Nº 5 – Modelo del ciclo de vida en Espiral.

Figura Nº 5 Extraída de [5]

El esquema del ciclo de vida para estos casos puede representarse por un bucle en espiral, donde los cuadrantes son, habitualmente, fases de especificación, diseño, realización y evaluación (o conceptos y términos análogos).

En cada fase el producto gana "madurez" (aproximación al final deseado) hasta que en una iteración se logre el objetivo deseado y este sea aprobado se termina las iteraciones.

4.3.4. INCREMENTAL

Permite construir el proyecto en etapas incrementales en donde cada etapa agrega funcionalidad.

Estas etapas, consisten en requerimientos, diseño, codificación, pruebas y entrega. Permite entregar al cliente un producto más rápido en comparación del modelo en cascada.

- Reduce los riegos ya que provee visibilidad sobre el progreso de las nuevas versiones.
- Provee retroalimentación a través de la funcionalidad mostrada.
- Permite atacar los mayores riesgos desde el inicio.
- Se pueden hacer implementaciones parciales si se cuenta con la suficiente funcionalidad.
- Las pruebas y la integración son constantes.
- El progreso se puede medir en periodo cortos de tiempo.
- Resulta más sencillo acomodar cambios al acortar el tamaño de los incrementos.
- Se puede planear en base a la funcionalidad que se quiere entregar primero.
- Por su versatilidad requiere de una planeación cuidadosa tanto a nivel administrativo como técnico.

Ventajas:

- La solución se va mejorando en forma progresiva a través de las múltiples iteraciones, incrementa el entendimiento del problema y de la solución por medio de los refinamientos sucesivos.
- Los clientes no esperan hasta el fin del desarrollo para utilizar el sistema. Pueden empezar a usarlo desde el primer incremento.
- Los clientes pueden aclarar los requisitos que no tengan claros, conforme ven las entregas del sistema.
- Se disminuye el riesgo de fracaso de todo el proyecto, ya que se puede distribuir en cada incremento.
- Las partes más importantes del sistema son entregadas primero, por lo cual se realizan más pruebas en estos módulos y se disminuye el riesgo de fallos.

Desventaja:

- Requiere de mucha planeación, tanto administrativa como técnica
- Requiere de metas claras para conocer el estado del proyecto.
- Es un proceso de desarrollo de software, creado en respuesta a las debilidades del modelo tradicional de cascada.

Para apoyar el desarrollo de proyectos por medio de este modelo se han creado Framework (entornos de trabajo), de los cuales los dos más famosos son el Rational Unified Process (RUP) y el Dynamic Systems Development Method.

El desarrollo incremental e iterativo es también una parte esencial de un tipo de programación conocido como Extreme Programming y los demás frameworks de desarrollo rápido de software.

Figura Nº 6 – Modelo del ciclo de vida Incremental.

Figura Nº 6 Extraída de [6]

4.3.5. RAD

La metodología de desarrollo conocida como diseño rápido de aplicaciones RAD (rapid application development) ha tomado gran auge debido a la necesidad que tienen las instituciones de crear aplicaciones funcionales en un plazo de tiempo corto. RAD es un ciclo de desarrollo diseñado para crear aplicaciones de computadoras de alta calidad.

El método comprende el desarrollo interactivo, la construcción de prototipos y el uso de utilidades CASE (Computer Aided Software Engineering). Tradicionalmente, el desarrollo rápido de aplicaciones tiende a englobar también la usabilidad, utilidad y la rapidez de ejecución.

Hoy en día se suele utilizar para referirnos al desarrollo rápido de interfaces gráficas de usuario tales como Glade, o entornos de desarrollo integrado completos. Algunas de las plataformas más conocidas son Visual Studio, Lazarus, Gambas, Delphi, Foxpro, Anjuta, Game Maker, Velneo o Clarion. En el área de la autoría multimedia, software como Neosoft Neoboo y MediaChance Multimedia Builder proveen plataformas de desarrollo rápido de aplicaciones, dentro de ciertos límites.

Fases del RAD

- **Modelado de gestión**: el flujo de información entre las funciones de gestión se modela de forma que responda a las siguientes preguntas: ¿Qué información conduce el proceso de gestión? ¿Qué información se genera? ¿Quién la genera? ¿A dónde va la información? ¿Quién la procesó?

- **Modelado de datos**: el flujo de información definido como parte de la fase de modelado de gestión se refina como un conjunto de objetos de datos necesarios para apoyar la empresa. Se definen las características (llamadas atributos) de cada uno de los objetos y las relaciones entre estos objetos.

- **Modelado de proceso**: los objetos de datos definidos en la fase de modelado de datos quedan transformados para lograr el flujo de información necesario para implementar una función de gestión. Las descripciones del proceso se crean para añadir, modificar, suprimir, o recuperar un objeto de datos. Es la comunicación entre los objetos.

- **Generación de aplicaciones**: El DRA asume la utilización de técnicas de cuarta generación. En lugar de crear software con lenguajes de programación de tercera generación, el proceso DRA trabaja para volver a utilizar componentes de programas ya existentes (cuando es posible) o a crear componentes reutilizables (cuando sea necesario). En todos los casos se utilizan herramientas automáticas para facilitar la construcción del software.

- **Pruebas de entrega**: Como el proceso DRA enfatiza la reutilización, ya se han comprobado muchos de los componentes de los programas. Esto reduce tiempo de pruebas. Sin embargo, se deben probar todos los componentes nuevos y se deben ejercitar todas las interfaces a fondo.

Características de RAD:

Equipos Híbridos:

- Equipos compuestos por alrededor de seis personas, incluyendo desarrolladores y usuarios de tiempo completo del sistema así como aquellas personas involucradas con los requisitos.
- Los desarrolladores de RAD deben ser "renacentistas": analistas, diseñadores y programadores en uno.

Herramientas Especializadas:

- Desarrollo "visual"
- Creación de prototipos falsos (simulación pura)
- Creación de prototipos funcionales
- Múltiples lenguajes
- Calendario grupal
- Herramientas colaborativas y de trabajo en equipo
- Componentes reusables
- Interfaces estándares (API)

Timeboxing:

- Las funciones secundarias son eliminadas como sea necesario para cumplir con el calendario.

Prototipos iterativos y evolucionarios:

- Reunion JAD (Joint Application Development):
  - Se reúnen los usuarios finales y los desarrolladores.
  - Lluvia de ideas para obtener un borrador inicial de los requisitos.

- Iterar hasta acabar:
  - Los desarrolladores construyen y depuran el prototipo basado en los requisitos actuales.
  - Los diseñadores revisan el prototipo.
  - Los clientes prueban el prototipo, depuran los requisitos.
  - Los clientes y desarrolladores se reúnen para revisar juntos el producto, refinar los requisitos y generar solicitudes de cambios.
  - Los cambios para los que no hay tiempo no se realizan. Los requisitos secundarios se eliminan si es necesario para cumplir el calendario.

Ventajas:

- Comprar puede ahorrar dinero en comparación con construir.
- Los entregables pueden ser fácilmente trasladados a otra plataforma.
- El desarrollo se realiza a un nivel de abstracción mayor.
- Visibilidad temprana.
- Mayor flexibilidad.
- Menor codificación manual.
- Mayor involucramiento de los usuarios.
- Posiblemente menos fallas.
- Posiblemente menor costo.
- Ciclos de desarrollo más pequeños.
- Interfaz gráfica estándar.

Desventajas:

- Comprar puede ser más caro que construir.
- Costo de herramientas integradas y equipo necesario.
- Progreso más difícil de medir.
- Menos eficiente.
- Menor precisión científica.
- Riesgo de revertirse a las prácticas sin control de antaño.

- Más fallas (por síndrome de “codificar a lo bestia”).
- Prototipos pueden no escalar, un problema mayúsculo.
- Funciones reducidas (por “timeboxing”).
- Dependencia en componentes de terceros: funcionalidad de más o de menos, problemas legales.

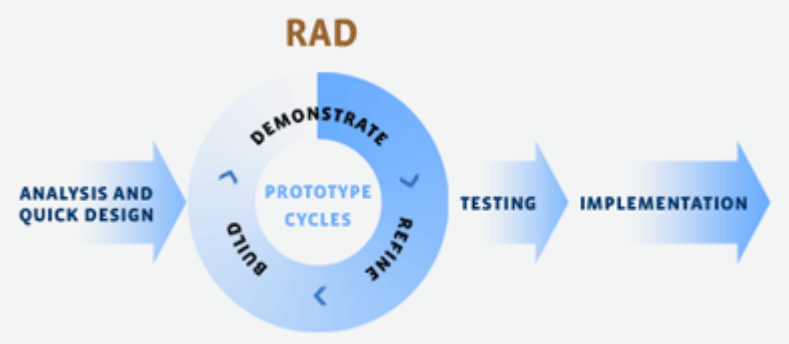

Figura Nº 7 – Modelo del ciclo de vida RAD.

Figura Nº 7 Extraída de [7]

5. METODOLOGIAS AGILES

5.1. INTRODUCCION

En las dos últimas décadas las notaciones de modelado y posteriormente las herramientas pretendieron ser la clave para el éxito en el desarrollo de software, no obstante, las expectativas no fueron satisfechas del todo. 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 eficientes para su aplicación.

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 de software ha demostrado ser efectivo y necesario en proyectos de gran tamaño donde por lo general se exige un alto grado de ceremonia en el proceso. Sin embargo, este enfoque no resulta ser el más adecuado y eficiente 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 esto conlleva. Esto puede dar la impresión de que se van a hacer mal las cosas o de dejarlo a medias pero lo cierto es que ser “ágiles" no significa renunciar a formalismos ni dejar de ser estrictos, rigurosos y metódicos.

En esta profesión y sobre todo en el área del desarrollo de software, la teoría nos orienta a ser extremadamente estrictos al momento de hacer el análisis y documentación de nuestro sistema, pero en la práctica actual no se puede perder tanto tiempo realizando estas tareas porque cuando se finaliza el sistema, éste ya ha cambiado y el cliente se ha aburrido. Ser abducido y caer en esta nueva dinámica, es muy fácil, cuando la presión nos cae encima y los plazos de entrega se acercan y cada vez se acortan más.

La alta competitividad actual hace que los sistemas de información se tengan que desarrollar de forma rápida para adaptarse a la organización. Las prisas hacen que lo primero que se deseche sea un análisis exhaustivo y se sustituye por uno superficial o simplemente se elimina. Éste es sin duda el gran error, ya que deseamos tener un sistema desarrollado rápidamente pero con lo que realmente nos encontramos es con un sistema lleno de errores, inmanejable y que no se puede mantener.

Es difícil cambiar las reglas del mercado mundial, así que lo que se ha pensado es adaptar las metodologías de especificación y desarrollo a este entorno cambiante y lleno de presiones, en el que obtener un resultado rápido, algo que se pueda ver, mostrar y sobre todo utilizar, se ha vuelto crucial para el éxito de las organizaciones. La metodología necesariamente ha de ser ágil, debe tener un ciclo corto de desarrollo y debe incrementar las funcionalidades en cada iteración del mismo preservando las existentes, ayudando al negocio en lugar de darle la espalda.

Es para este entorno para el que han nacido las metodologías ágiles y como consecuencia se creó el Manifiesto Ágil.

El Manifiesto Ágil describe, básicamente, cuales son los principios sobre los que se basan los métodos reconocidos como ágiles.

Éste manifiesto sugiere un enfoque orientado a la participación de los usuarios y clientes, más que hacia los procesos y herramientas, trabajando más en el software y menos en la documentación, colaborando más con los clientes en vez de estar negociando y respondiendo a los cambios sacrificando el plan de trabajo si es necesario.

En el "Manifiesto ágil" se definen los cuatro valores principales por las que se deberían guiar las metodologías ágiles.

1. Al individuo y sus interacciones más que al proceso y las herramientas.
2. Desarrollar software que funciona más que obtener una documentación exhaustiva.
3. La colaboración con el cliente más que la negociación de un contrato.
4. Responder a los cambios más que seguir una planificación.

El significado de cada uno de estos valores son los siguientes:

1. Al individuo y sus interacciones más que al proceso y las herramientas.

   Sin duda, la herramienta fundamental de la ingeniería del software y del desarrollo de aplicaciones es el cerebro humano y nuestra capacidad para desarrollar y desenvolvernos con nuestro entorno.

   Una persona sola no realiza un proyecto, necesita de un entorno en el que desarrollar su trabajo y de un equipo con el que colaborar. Estas interacciones deben también cuidarse. Un factor clave para el éxito es construir un buen equipo, que se lleven bien entre ellos, y que además sepan cómo construir su propio entorno de desarrollo. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte a él. Esto nos resta eficiencia, es mejor que el equipo se configure sobre la base de sus necesidades y sus características personales.

   Además, las interacciones que haga el equipo con el usuario final deberían ser igual de fluidas siendo este usuario un miembro más del equipo, con un objetivo común, que es conseguir que el proyecto funcione y sea útil para él.

   Si todo esto funciona bien, la elección de las herramientas y el proceso mismo de desarrollo pasan a estar en un plano totalmente secundario en la ecuación del éxito del proyecto.

2. Desarrollar software que funciona más que obtener una documentación exhaustiva.

   Uno de los objetivos de una buena documentación es poder ir a consultarla cuando haya que modificar algo del sistema o surja alguna incertidumbre. Sin duda es un arma de doble filo, una buena documentación actualizada en cada modificación y bien mantenida al día permite saber el estado de la aplicación y cómo realizar las modificaciones pertinentes, pero

son pocos los que con las presiones externas de tiempo y dinero acaban actualizando la documentación. Generalmente, cuando se tiene que arreglar un programa porque algo falla o surge un cambio de alcance, nos concentramos en que funcione ya que es muy posible que haya usuarios parados y que la empresa o la organización esté perdiendo dinero, en estos casos, ni nos paramos a mirar detenidamente la documentación ni, cuando se acaba de arreglar, nos ponemos a actualizarla.

En la filosofía ágil, lo primordial es evitar estos fallos, centrar nuestro tiempo en asegurar que el software funciona y que se ha testeado exhaustivamente, e intentar reducir la creación de documentos poco útiles, de éstos que acaban no manteniéndose y ni siquiera consultándose. La regla no escrita es no producir documentos superfluos, y sólo producir aquellos que sean necesarios de forma inmediata para tomar una decisión importante durante el proceso de desarrollo. Estos documentos deben ser cortos y centrarse en lo fundamental, olvidándonos de las ambigüedades que no aportan nada a la comprensión del problema. No obstante, los documentos son soporte de documentación, permiten la transferencia del conocimiento, registran información histórica, y en muchas cuestiones legales o normativas son obligatorios, pero se resalta que son menos importantes que los productos que funcionan.

3. La colaboración con el cliente más que la negociación de un contrato.

La consultoría informática de los últimos años se ha convertido en una lucha a muerte entre el proveedor del servicio y el cliente que lo contrata. Por una parte, el cliente intenta que se hagan el mayor número de funcionalidades con el mismo dinero, por otra parte, el consultor intenta que por ese dinero sólo se realicen las funcionalidades contratadas inicialmente. En las reuniones de seguimiento de los proyectos es fácil oír frases del tipo "esta modificación no entra en el contrato" respondida generalmente por la tan típica "pues yo ya no tengo más presupuesto y así no podemos trabajar". Al final, este tipo de proyectos hacen que el consultor informático sea un híbrido entre analista y abogado, que desarrolla habilidades legales para salvaguardarse en caso de conflicto jurídico.
Sin embargo, para que un proyecto tenga éxito es fundamental la complicidad y el contacto continuo entre el cliente y el equipo de desarrollo. El cliente debe ser y sentirse parte del equipo. De esta manera, ambos entenderán las dificultades del otro y trabajarán de forma conjunta para solucionarlo.

4. Responder a los cambios más que seguir una planificación.

Una organización cambia constantemente, se adapta a las necesidades del mercado y reorganiza sus flujos de trabajo para ser más eficiente. Es difícil, pues, que en el desarrollo de un proyecto, éste no sufra ningún cambio, ya que es seguro que las necesidades de información de la empresa habrán cambiado. Y no sólo esto, sino que las posibilidades económicas de la misma pueden influir sobre nuestro proyecto, bien sea agrandándolo o reduciéndolo. Son muchos los factores que alterarán nuestra planificación inicial del proyecto.

Si no la adaptamos a estos cambios, corremos el riesgo de que, cuando acabemos, nuestro sistema no cumpla con las necesidades pre-establecidas y el cliente se haya gastado el dinero en vano. La habilidad de responder a los cambios de requisitos, de tecnología, presupuestarios o de estrategia, marcan sin duda el camino del éxito del proyecto.

Como consecuencia de estos cuatro valores, el Manifiesto ágil también enuncia los doce principios que caracterizan un proceso ágil diferenciándolo de otro tradicional donde este enfoque no se había aplicado lo suficiente, siempre se había dejado implícito pero sin hacer hincapié en ellos.

a) La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.
b) Dar la bienvenida a los cambios incluso al final del desarrollo. Los cambios le darán una ventaja competitiva a nuestro cliente.
c) Hacer entregas frecuentes de software que funcione, desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.
d) Las personas del negocio y los desarrolladores deben trabajar juntos diariamente a lo largo de todo el proyecto.
e) Construir el proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos.
f) El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.
g) El software que funciona es la principal medida del progreso.
h) Los procesos ágiles promueven un desarrollo sostenido. Los promotores, usuarios y desarrolladores deben poder mantener un ritmo de trabajo constante de forma indefinida.
i) La atención continua a la calidad técnica y al buen diseño mejoran la agilidad.
j) La simplicidad es esencial. Se ha de saber maximizar el trabajo que no se debe realizar.
k) Las mejores arquitecturas, requisitos y diseños surgen de los equipos que se han organizado ellos mismos.
l) En intervalos regulares, el equipo debe reflexionar con respecto a cómo llegar a ser más efectivo, y ajustar su comportamiento para conseguirlo.

Llegados a este punto, podemos intuir que las formas de hacer las cosas en la ingeniería del software están cambiando, adaptándose más a las personas y a las organizaciones en las que han de funcionar las aplicaciones.

Las metodologías ágiles no son la gran solución a todos los problemas del desarrollo de aplicaciones, ni tan siquiera se pueden aplicar en todos los casos, pero sí que nos aportan otro punto de vista de cómo se pueden llegar a hacer las cosas, de forma más rápida, más adaptable y sin tener que perder la rigurosidad de las metodologías clásicas.

5.2. BENEFICIOS

Durante la década pasada, las metodologías ágiles demostraron ser muy beneficiosas ayudando a los equipos de desarrollo de software a la hora de entregar en fecha software de alta calidad que compensara las necesidades de los clientes. Los equipos de software quieren trabajar con metodologías ágiles porque necesitan un proceso que pueda responder de manera eficiente a los cambios en los productos en desarrollo. Las metodologías ágiles permiten una mayor flexibilidad que las metodologías tradicionales de desarrollo, ya que éstas se bloquean y ralentizan en los detalles del proyecto y son menos capaces de ajustarse a las cambiantes necesidades de los clientes, del mercado y de los desafíos imprevistos que plantea la tecnología y el medio ambiente.

Algunos de los beneficios son los siguientes:

1. Simplificación de la sobrecarga de procesos

Los equipos que trabajan para crear productos regulados por estándares de la industria, deben demostrar el cumplimiento de esas normas. Estos equipos suelen adoptar importantes sobrecargas de trabajo para asegurarse de que cumplen con los estrictos mandatos de código. Las metodologías ágiles pueden ayudarles a cumplir los estándares de la industria con menos sobrecarga utilizando iteraciones más cortas y empaquetadas. El beneficio neto es un proceso que:

- Puede adaptarse a los cambios que, inevitablemente, surgirán
- Requiere menos sobrecarga en el proceso end-to-end
- Implica menos trabajo a medida que se acerca la fecha final

2. Calidad mejorada

Las prácticas de desarrollo ágil proporcionan la funcionalidad suficiente como para satisfacer las expectativas de los clientes con una alta calidad. Eso es, proporcionar la mínima funcionalidad con la máxima calidad. La mínima funcionalidad no implica necesariamente una pobre funcionalidad, sino que implica lo suficiente como para conseguir que el trabajo se realice. Los clientes suelen pensar que saben lo que quieren cuando especifican altos niveles de requerimientos para un producto de software. Sin embargo, la mayoría de las veces, cuando ven el producto final, éste no resuelve el problema. Simplemente, no se han imaginado de forma precisa el mismo, o el problema ha cambiado o, incluso, la tecnología no era tan buena como parecía. También puede ser que el producto no funcione realmente del modo en que las partes interesadas tenían intención, incluso aunque pensaran que habían descrito los requerimientos de manera clara. Desarrollar iteraciones en poco tiempo y demostrar a los clientes los productos pronto y con frecuencia, permite tanto a los clientes como a los equipos de desarrollo ponerse de acuerdo y coincidir en que el producto cumple con cada una de sus necesidades.

3. Mejorar la previsibilidad a través de una mejor gestión del riesgo

Cuando los equipos de desarrollo no cumplen con sus fechas de lanzamiento, a menudo se debe a muchas razones completamente justificables. Puede ser que el equipo no hubiera entendido bien lo complejo que sería utilizar la nueva tecnología, los requerimientos podían no estar del todo claros o los clientes cambiaron de opinión cuando el trabajo estaba prácticamente finalizado. En cualquier caso, los negocios demandan productos que cumplan con las fechas de entrega, de modo que los planes de negocio directamente relacionados con ellos puedan cumplirse. Hay muchas formas en las que las metodologías ágiles pueden ayudar a que los proyectos cumplan con las fechas de lanzamiento previstas.

Dar prioridad a los riegos. Las prácticas ágiles priorizan los aspectos de desarrollo de alto riesgo permitiendo así una reducción de los riesgos iniciales.

Evaluación de riesgos en paralelo. Para áreas de riesgo donde debería haber múltiples soluciones y el equipo no se pone de acuerdo a la hora de tomar el camino más adecuado, se debe tener en cuenta la posibilidad de optar por el desarrollo multi-set. Esto requiere que diferentes equipos trabajen en paralelo resolviendo el mismo problema con diferentes soluciones. La mayoría de los equipos no considerarán ni tan siquiera esta forma de trabajar, porque están convencidos de que el tiempo y el coste requeridos para hacer una evaluación paralela son demasiado grandes. De hecho, el desarrollo paralelo de alternativas es probable que traiga consigo las decisiones clave a seguir.

4. Mejor perfil de productividad

Los equipos ágiles son más productivos que aquellos que utilizan métodos tradicionales a lo largo de todo el ciclo de desarrollo. El desarrollo tradicional comienza con un ciclo de diseño de abajo arriba, moviéndose hasta una fase de prototipo para pasar luego a un ciclo de desarrollo largo, seguido por otro ciclo totalmente impredecible en el que se integran las piezas para pasar, por último, a la fase de prueba final. A medida que el proyecto progresa, los equipos tienen que trabajar juntos de manera más coherente, confiando en que todas las piezas trabajen juntas tal y como esperan. Pero lo cierto es que esto raramente ocurre, de modo que la interacción entre los equipos aumenta a medida que lo hacen los problemas de integración. Por último, la fase de pruebas lleva todo esto al límite. Trabajando juntos como un único equipo en fases verticales del producto desde el principio del ciclo de producción, se evita el ciclo de productividad tradicional. Los equipos ágiles tienden a ser muy productivos desde la primera iteración hasta su lanzamiento y su ritmo tiene que ser gestionado de modo que no se produzca agotamiento. Los equipos ágiles que mantienen este código de trabajo con cada iteración, permiten realizar pruebas de rendimiento y sistemas desde el principio, pudiendo empezar en las primeras iteraciones. De este modo, defectos críticos como problemas de integración se descubren antes, la calidad general del producto es mayor y el equipo funciona de manera más productiva durante todo el ciclo de desarrollo.

5. Capacidad para aprovechar las inversiones realizadas

Adoptar las técnicas ágiles de manera satisfactoria impone un importante soporte de herramientas. La mayoría de los equipos invierten fuertemente en buenas herramientas de software y necesitan elevar esa inversión y reducir la cantidad de cambios cuando empiezan a adoptar las metodologías ágiles. La inversión más crítica es una buena planificación y una herramienta de gestión del trabajo que proporcione una cartera de pedidos del equipo visible y que asocie el trabajo con cada elemento de trabajo en cartera. Idealmente, esa herramienta de planificación se integra con otras herramientas de desarrollo, lo que permite a los equipos mantener la trazabilidad de la cartera de pedidos en otros aspectos, incluyendo:

- Los requerimientos que la impulsan.
- La arquitectura bajo desarrollo.
- El software de la solución.
- Las pruebas que validan la solución.

Algunas de las herramientas más conocidas para la gestión y administración de proyectos ágiles son:

- Team Foundation Server (Microsoft)
- Rational (IBM)
- HP Quality Center (Hewlett Packard)

6. Realimentación continua con el cliente

De forma temprana el cliente recibe entregables de valor, lo que permite ver los constantes avances, logrando así, aportar en lo necesario para que el equipo vaya construyendo en la dirección correcta lo anterior, inmediatamente reduce de forma drástica los errores y la posibilidad de costosas correcciones, respondiendo a los cambios en requisitos de forma rápida y eficaz. El cliente es una parte más del equipo.

7. Equipo motivado

Cuando se realiza un proyecto aplicando alguna de las metodologías agiles, las personas involucradas en el mismo están más motivadas ya que pueden usar su creatividad para resolver problemas y cuando pueden decidir organizar su trabajo. Esto hace que las personas se sienten más satisfechas cuando pueden mostrar los logros que consiguen tomando sus propias decisiones.

5.3. TIPOS DE METODOLOGIAS

5.3.1. PROGRAMACION EXTREMA

La programación extrema (XP) puede que marque un antes y un después en la ingeniería del software. Ésta nueva forma de trabajo fue creada por Kent Beck, Ward Cunninghamn y Ron Jeffries a finales de los noventa. La programación extrema ha pasado de ser una simple idea para un único proyecto a inundar todas las factorías de software. Algunos la definen como un movimiento social de los analistas del software hacia los hombres y mujeres de negocios, de lo que debería ser el desarrollo de soluciones en contraposición a los legalismos de los contratos de desarrollo.

Es el más destacado de los procesos ágiles de desarrollo de software. Al igual que éstos, la programación extrema se diferencia de las metodologías tradicionales principalmente en que pone más énfasis en la adaptabilidad que en la previsibilidad. Los defensores de XP consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos. Creen que ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los requisitos.

Es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en 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.

Para alcanzar el objetivo de software como solución ágil, la metodología XP se estructura en tres capas que agrupan las doce prácticas básicas de XP:

Figura Nº 8 – Capas de la Programación Extrema.

1. Metodología de programación: diseño sencillo, testing, refactorización y codificación con estándares.
2. Metodología de equipo: propiedad colectiva del código, programación en parejas, integración continua, entregas semanales e integridad con el cliente.
3. Metodología de procesos: cliente in situ, entregas frecuentes y planificación.

Introducir la vertiente de las relaciones sociales dentro de una metodología es lo que hace de XP algo más que una guía de buenas maneras. Convierte a la programación en algo mucho más humanizado, en algo que permite a las personas relacionarse y comunicarse para encontrar soluciones, sin jerarquías ni enfrentamientos. Los analistas y programadores trabajan en equipo con el cliente final, todos están comprometidos con el mismo objetivo, que la aplicación solvente o mitigue los problemas que tiene el cliente. La vertiente social es fundamental en otras áreas del conocimiento. XP también humaniza a los desarrolladores. Un entorno agradable para el trabajo, que facilite la comunicación y los descansos adecuados, forma parte de esta metodología.

Pero donde XP centra la mayor innovación es en desmontar la preconcebida idea del coste del cambio de las metodologías en cascada, es decir, lo que cuesta cambiar alguna funcionalidad de nuestro aplicativo a medida que vamos avanzando en él. La idea generalizada es que cualquier modificación a final del proyecto es exponencialmente más costosa que al principio del mismo. Si algo no estaba especificado inicialmente, cuesta mucho introducirlo al final del proyecto.

Figura Nº 8 Extraída de [8]

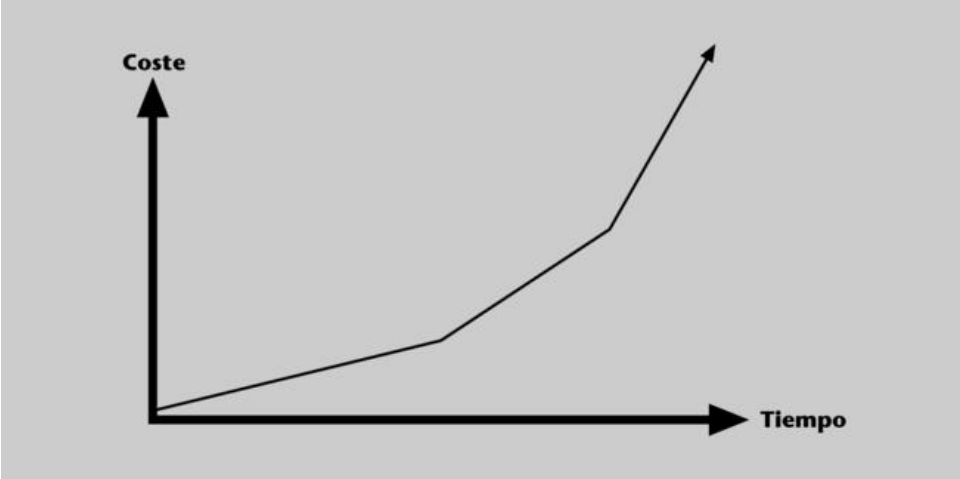

Figura Nº 9 – Curva de relación Coste/Tiempo en metodologías tradicionales.

Lo que XP mantiene es que esta curva ha perdido vigencia y que con una combinación de buenas prácticas de programación y tecnología es posible lograr que la curva no crezca siempre de forma exponencial.

XP pretende conseguir una curva de coste del cambio con crecimiento leve, que en un inicio es más costosa, pero que a lo largo del proyecto permite tomar decisiones de desarrollo lo más tarde posible sin que esa nueva decisión de cambio implique un alto coste en el proyecto.

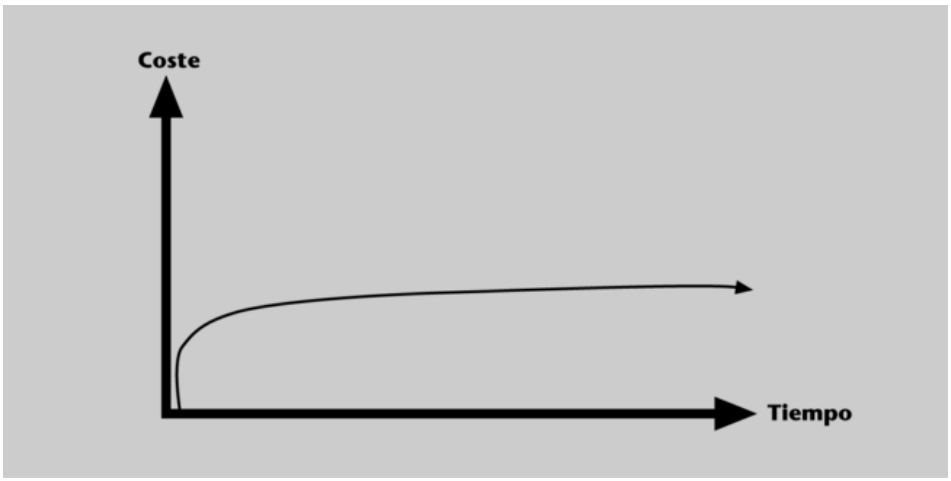

Figura Nº 10 – Curva de relación Coste/Tiempo en metodologías ágiles.

La programación extrema, propone una ecuación de equilibrio entre el coste, el tiempo de desarrollo, la calidad del software y el alcance de funcionalidades del mismo.

Figura Nº 9 Extraída de [9]
Figura Nº 10 Extraída de [10]

Las variables: coste, tiempo, calidad y alcance

El punto de partida de la metodología XP son las variables que utiliza para cada proyecto:

- **Coste** (la inversión económica y en recursos).
- **Tiempo** (el tiempo empleado, determinado por la fecha de entrega final).
- **Calidad** (del código y del aplicativo desarrollado).
- **Alcance** (Conjunto de funcionalidades).

De estas cuatro variables, tres podrán ser fijadas por el cliente y/o por el jefe de proyectos, la cuarta es responsabilidad del equipo de desarrollo y se establecerá en función de las otras tres.

Obviamente con XP no se establece el valor de todas las variables a la primera de cambio, es un proceso paulatino en el que cada uno de los responsables (cliente, jefe de proyecto y equipo de desarrollo) negocian los valores de las variables que tienen asignadas hasta conseguir una ecuación equilibrada y que satisfaga a todos.

Distinguir entre los requisitos más importantes y los que aportan menor valor al negocio, dando prioridad a los primeros, nos ayudará a conseguir que el proyecto tenga en cada instante tanta funcionalidad como sea posible. De esta manera, cuando nos acerquemos al plazo de entrega nos aseguraremos de que lo principal está implementado y que sólo quedarán los detalles de los que podemos prescindir en el caso de necesidad. El objetivo es siempre maximizar el valor de negocio.

Los valores: comunicación, simplicidad, realimentación y coraje

Los creadores de esta metodología quisieron medir su utilidad a través de cuatro valores, que representan aquellos aspectos cuyo cumplimiento nos va a garantizar el éxito en el proyecto: comunicación, simplicidad, realimentación y coraje.

- **Comunicación**: debe ser fluida entre todos los participantes en el proyecto. El entorno tiene que favorecer la comunicación espontánea, ubicando a todos los miembros en un mismo lugar. La comunicación directa nos da mucho más valor que la escrita, podemos observar los gestos del cliente, o la expresión de cansancio de nuestro compañero.

- **Simplicidad:** cuanto más sencilla sea la solución, más fácilmente podremos adaptarla a los cambios. Las complejidades aumentan el coste del cambio y disminuyen la calidad del software. Sólo se utiliza lo que en ese momento nos da valor, y lo haremos de la forma más sencilla posible. Esto podría dar a pensar que va en contra de toda la filosofía de diseño y utilización de patrones. Nada más alejado de la realidad. En un proyecto XP, el uso de patrones nos va a ayudar a reducir el tiempo de implantación, pero lo que no vamos a hacer es dedicar tiempo a la implementación de patrones que no vayamos a utilizar en este proyecto, sólo haremos los que sean necesarios para éste, no utilizaremos tiempo del proyecto para beneficiar a otro proyecto futuro que quizás no llegue nunca. Por otro lado, nada nos impide

desarrollar un proyecto que únicamente se dedique a desarrollar patrones que más tarde se utilicen en proyecto XP.

- **Realimentación**: el usuario debe utilizar y probar el software desarrollado desde la primera entrega, dándonos sus impresiones y sus necesidades no satisfechas, de manera que estas cosas encontradas vuelvan a formar parte de los requisitos del sistema y así poder atacarlas con tiempo.

- **Coraje**: con XP debemos tocar continuamente cosas que ya funcionan, para mejorarlas, optimizarlas o agregar funcionalidad. Es por eso que el coraje es parte del proyecto también, ya que el miedo a tocar o modificar cosas que ya funcionan perfectamente, a veces puede ser difícil.

Las doce prácticas básicas de XP

Kent Beck, Ward Cunninghamn y Ron Jeffries tenían muy claro las prácticas que les habían dado mejores resultados en sus proyectos. Así que intentaron aplicarlas todas juntas dando una vuelta de tuerca. Ésa fue la semilla de la metodología de Programación Extrema. Crearon las doce prácticas que se refuerzan entre sí para obtener los mejores resultados. Las relaciones entre ellas las podemos ver en el siguiente gráfico:

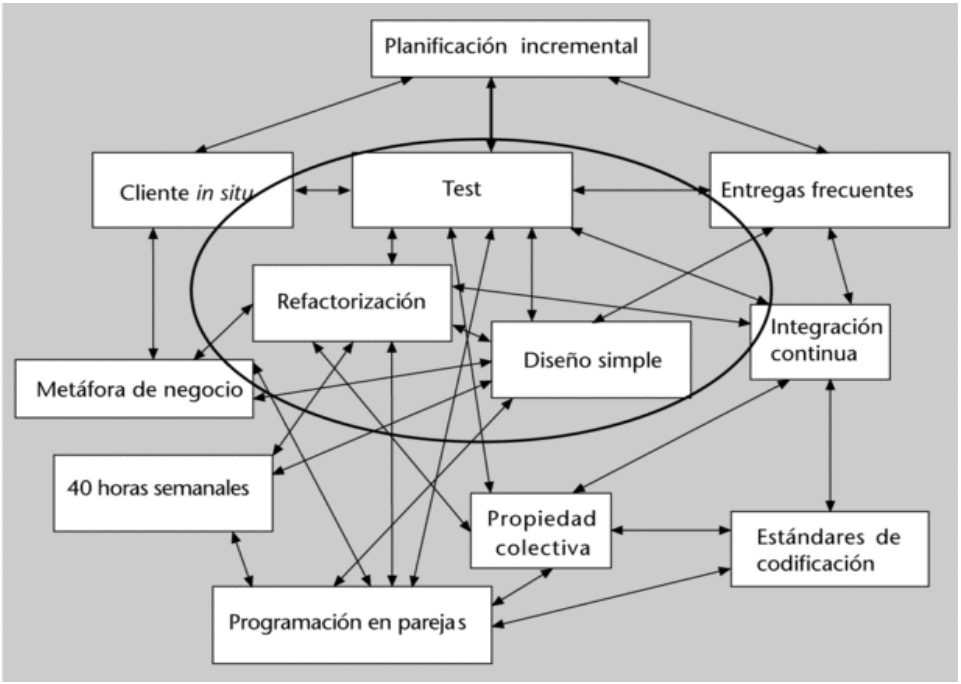

Figura Nº 11 – Prácticas de la Programación Extrema.

Figura Nº 11 Extraída de [11]

En el centro están situadas las prácticas que más resultados nos pueden dar al adaptarlas (diseño simple, el test y la refactorización).

Incluso si no queremos tomar la totalidad de las prácticas de XP adoptando estas tres a nuestra metodología habitual, podemos tener una sustancial mejora en los resultados obtenidos.

1. **Diseño simple**: si nuestro diseño es simple, cuando alguien lo vea lo entenderá, si es complejo, probablemente no pueda descifrarlo.

   El principio es utilizar el diseño más sencillo que consiga que todo funcione, de esta manera facilitaremos el mantenimiento y minimizaremos los riesgos de modificaciones que se hagan sin necesidad de entender el código en su máximo nivel.

   XP define diseño simple aquel que:

   - No tiene código redundante, ni duplicado.
   - Supera todas las pruebas de funcionalidad, integridad y aceptación.
   - No utiliza sintaxis complejas, es decir, que queda clara la intención de los programadores en cada línea de código.
   - Contiene el menor número posible de clases y métodos.

   Refactorización: inicialmente, nuestro sistema, son todas líneas de código bien ordenadas y comentadas, pero a medida que vamos introduciendo cambios, el orden se va perdiendo hasta que aquello deriva en una serie de líneas de código caóticas, llenas de código comentado de las diferentes pruebas y con comentarios obsoletos que no se corresponden con la realidad de la funcionalidad.

   El excesivo coste de las modificaciones de las metodologías tradicionales se debe en gran medida a este deterioro progresivo del código, tras la acumulación de modificaciones.

   Para mantener la curva del coste de cambio tan plana como sea posible, en la metodología XP se aplica la refactorización, que no es otra cosa que modificar el código para dejarlo en buen estado, volviendo a escribir las partes que sean necesarias pero siempre desde un punto de vista global a la funcionalidad, independientemente del cambio que hagamos.

   El código final debe conservar la claridad y sencillez del original. Los comentarios son muy importantes para mantener esta claridad y sencillez.

2. **Test**: es el pilar fundamental de las prácticas de esta metodología, sin los test, XP sería un fracaso total, es el punto de anclaje que le da la base metodológica a la flexibilidad de XP.

   Si una funcionalidad no se ha testeado, sólo funciona en apariencia, los tests han de ser aplicados tras cada cambio, y han de ser automatizados en la medida que estos sean posibles. Si no lo hacemos, podemos incurrir en fallos humanos a la hora de testearlos y eso puede resultar fatal.

El objetivo de los tests no es detectar errores, sino evitarlos, no se trata de corregir errores, sino de prevenirlos. Para ello, los tests siempre se escriben antes que el código a testear, nunca después. De esta manera estamos obligados a pensar por adelantado cuáles son los problemas que podemos encontrar cuando usen nuestro código, a ponernos en el lugar del usuario final, y este hecho de pensar por adelantado evita muchos problemas, previniéndonos sobre ellos en lugar de dejar que aparezcan y luego responder sobre la marcha.

Cuando escribimos el código, ya sabemos a qué se ha de enfrentar y de esta manera los errores se minimizan. El objetivo de nuestro software ya no es cumplir unas funcionalidades sino pasar una serie de tests definidos previamente. Es por esto por lo que el usuario final debe ayudar al programador a desarrollar los tests unitarios de forma conjunta, ya que en éstos estará implícita la funcionalidad que queremos que tenga. Otro factor clave es que debe ser el mismo programador el que los desarrolle, si no es así, perdemos la ventaja de minimización de errores.

Los tests han de ser de tres tipos: de aceptación, unitarios y de integridad; y siempre han de estar automatizados en su mayor medida posible.

- **Test de aceptación**: es creado conjuntamente con el cliente final y debe reflejar las necesidades funcionales del primero.
- **Test unitario**: es creado por el programador para ver que todos los métodos de la clase funcionan correctamente.
- **Test de integridad**: es creado por el equipo de desarrollo para probar que todo el conjunto funciona correctamente con la nueva modificación.
  El uso de los tests nos facilita la refactorización, permitiéndonos comprobar de forma sencilla que los cambios originados por la refactorización no han cambiado el comportamiento del código.

3. **Estándares de codificación**: el equipo de desarrollo debe tener unas normas de codificación en común, unas nomenclaturas propias que todos los miembros del equipo puedan entender.

   El hecho de utilizar una nomenclatura común permite que cualquier persona del equipo entienda con mayor facilidad el código desarrollado por otro miembro, de esta manera, facilitamos las modificaciones y la refactorización.

   Por ejemplo, si decidimos que a la hora de poner nombre a nuestras variables de una función seguiremos las siguientes normas:

   - La primera letra ha de ser una "v" si es variable local o una "p" si nos viene por parámetro de entrada o una "o" si es parámetro de entrada y salida.
   - La segunda letra ha de indicar el tipo de datos "n" si es numérico, "s" si es una cadena de caracteres, "d" si es una fecha y "b" si es booleano.
   - El resto del nombre ha de ser descriptivo con su contenido lógico, separando significados con el "_".

   Si yo ahora en el código veo la siguiente sentencia:

If (vbPeriodo_no_calculable) {odFinal_Periodo = pdInicio_Periodo;} tengo mucha más información que si hubiese visto la siguiente: ***If (pnc) { fp = ip;}***
El funcionamiento es el mismo y ambas sentencias están bien codificadas, pero la primera nos da el valor añadido de la comprensión de la codificación estándar.

4. **Propiedad colectiva del código**: para poder aplicar la refactorización y para asegurarnos de que el diseño es simple y que se codifican según los estándares, tenemos que eliminar otra de las ideas que están muy arraigadas en el mundo del desarrollo de aplicaciones que es la "*propiedad individual del código"*. Frases como "*que lo modifique quien lo hizo que seguro que lo entiende mejor*" o *"¿quién ha tocado mi función*?" dejan de tener sentido en XP, ya que el diseño simple nos garantiza que será fácilmente entendible, la refactorización permite que cualquier miembro del equipo rehaga el código, los test automatizados nos garantizan que no hemos modificado el comportamiento esperado del código con nuestra modificación, y la codificación con estándares nos da ese grado de comprensión adicional.

   En XP el código es propiedad de todo el equipo y cualquier miembro tiene el derecho y la obligación de modificarlo, para hacerlo más eficiente o comprensible, sin que nadie se tenga por qué sentir ofendido o con miedo de tocar algo.

5. **Programación por parejas**: la programación siempre se ha visto como algo solitario, y tener dos personas delante de un solo teclado y de un solo monitor sorprende en un inicio. Pero las ventajas son muchas.

   Nos es muy complicado pensar a nivel abstracto y luego pasar a pensar a nivel concreto, y si lo hacemos de forma continua, acabamos desconcentrados y cometemos errores. Es por esto por lo que, cuando programamos, primero pensamos la estrategia de codificación que vamos a seguir y luego nos ponemos a codificar cada una de las partes, sin pensar de nuevo a nivel estratégico hasta que no finalizamos alguna de las partes o a veces la totalidad del código, y entonces vemos los errores o las cosas que nos hemos dejado en la estrategia de codificación original.

   Pensar a lo grande y en detalle a la vez nos es imposible con un solo cerebro. En la programación en parejas uno de los miembros debe estar pensando a nivel táctico y el otro a nivel estratégico de manera que esos dos procesos siempre estén activos reduciendo así los errores y mejorando la calidad del programa. Obviamente, estos dos roles deben intercambiarse cada poco tiempo entre los miembros de la pareja para abarcar todas las posibilidades tácticas y estratégicas.

   El nivel de los miembros de la pareja ha de ser equivalente, no sirve que uno sepa mucho y otro no tenga ni idea, deben de estar equilibrados y obviamente llevarse bien para que tenga éxito.

   También la rotación ha de ser muy importante, cada miembro del equipo ha de ser capaz de trabajar en cualquier área de la aplicación que se esté desarrollando. De esta manera no provocaremos cuellos de botella cuando asignemos las tareas.

El hecho de que asignemos las tareas por parejas hace que el tiempo de aprendizaje se reduzca casi a cero, simplemente sustituyendo uno de los miembros por otro nuevo. Así pues, la rotación de áreas y de parejas nos garantiza que podremos hacer un reparto más equitativo del trabajo sin tener que depender de una sola persona para un trabajo específico.

Otro efecto que produce la programación en parejas es el psicológico, disminuye la frustración de la programación en solitario, tienes a alguien que entiende el problema justo al lado, y con el que compartir el problema. Además, muchas veces los problemas vienen por falta de concentración y se tiene la solución delante de nosotros y no se ve, y se pierde mucho tiempo.

6. **Integración continúa**: en XP no esperamos a que todas las partes estén desarrolladas para integrarlas en el sistema, sino que a medida que se van creando las primeras funcionalidades ya se ensamblan en el sistema, de manera que éste puede ser construido varias veces durante un mismo día. Esto se hace para que las pruebas de integración vayan detectando los errores desde el primer momento y no al final de todo y hace que el impacto sea mucho menor. Es responsabilidad de cada equipo publicar lo antes posible cada funcionalidad o cada modificación. La idea es que todos los miembros del equipo trabajen con la última versión del código, para así evitar pisar versiones desarrolladas por otros compañeros y también poder interactuar nuestro código con la última versión vigente y así evitar posibles errores de compatibilidad.

7. **40 horas semanales**: no se pueden trabajar durante 14 horas seguidas y hacerlo con calidad. Las semanas de 70 horas de trabajo son contraproducentes. Al final de la semana se tiene que llegar cansado pero satisfecho, nunca exhausto ni desmotivado. Trabajar horas extras disminuye la moral y el espíritu del equipo. Si durante dos semanas hay que hacer horas extras, entonces es que el proyecto va mal y se debe replantear alguna de las cuatro variables.

8. **Metáfora del negocio**: para que dos o más personas se puedan comunicar de forma eficiente, deben tener el mismo vocabulario y compartir el mismo significado. El modelo de negocio que entiende el usuario final seguramente no se corresponderá con el que cree entender el programador. Es por esto por lo que en los equipos de XP se debe crear una "*metáfora*" con la que el usuario final se encuentre cómodo y que le sirva al equipo de desarrollo a la hora de modelar las clases y métodos del sistema. La metáfora, es un requerimiento común compartido por el usuario y el equipo de desarrollo que describe cómo deben comportarse las diferentes partes del sistema que se quiere implementar, en un alto nivel de abstracción.

9. **Cliente in situ**: en más de una ocasión, cuando estamos programando o analizando el sistema, nos surge una duda y pensamos en que cuando veamos al usuario final se la preguntaremos. Posiblemente tengamos que seguir trabajando sin resolver esa duda y, si nuestra suposición

ha sido errónea, mucho del trabajo realizado puede ser en vano y así haber aumentado el coste de nuestro proyecto.

XP necesita que el cliente final forme parte del equipo de desarrollo y esté ubicado físicamente en el mismo sitio para que así se agilice el tiempo de respuesta y se puedan validar todas las funcionalidades lo antes posible.

En XP, el cliente siempre tiene que estar disponible para el resto del equipo, formando parte de él y fomentando la comunicación cara a cara, que es la más eficiente de las comunicaciones posibles.

10. **Entregas frecuentes**: Se deben desarrollar lo antes posible versiones pequeñas del sistema, que aunque no tengan toda la funcionalidad, nos den una idea de cómo ha de ser la entrega final y que nos sirvan para que el usuario final se vaya familiarizando con el entorno y para que el equipo de desarrollo pueda ejecutar las pruebas de integridad.

    Las nuevas versiones tienen que ser tan pequeñas como sean posibles, pero tienen que aportar un nuevo valor agregado del negocio para el cliente. Dar valor al negocio es lo que hará que la visión final del cliente sea la mejor posible.

11. **Planificación incremental**: la planificación nunca será perfecta, variará en función de cómo varíen las necesidades del negocio y en cada ciclo de re planificación se volverán a establecer las cuatro variables de la metodología XP.

    Asumir una planificación estática no corresponde con la agilidad que queremos dar, ya que las necesidades del negocio pueden cambiar drásticamente mientras estamos desarrollando el aplicativo. En XP la planificación se va revisando continuamente, de forma incremental, priorizando aquellas necesidades de negocio que nos aporten mayor valor.

    La planificación se plantea como un permanente diálogo entre los responsables de la perspectiva empresarial y de la perspectiva técnica del proyecto.

Ciclo de vida de la Programación Extrema

Las doce prácticas mencionadas anteriormente habían dado por separado muy buenos resultados a Kent Beck y compañía, pero unificadas en un mismo ciclo de vida es lo que dio origen a la metodología XP.

Una de las características más importantes de este modelo es la comunicación y la mejor forma de hacerlo es presencialmente. Cuando tenemos algo importante que decir, no hay mejor manera que hacerlo cara a cara, para que no haya equívocos. La expresión, la entonación y las miradas son muy importantes en este proceso.

El ciclo de vida de XP se organiza como si fuese una conversación cliente- desarrollador.

1) Tengo una necesidad.

2) ¿Puedo ayudarte? ¿Qué necesitas?

3) Lo que necesito es esto.

4) Creo que lo entiendo. ¿Si construyo esto te sirve?

5) Pues sí. ¿Podrías hacerlo?

6) Ahora mismo me pongo a trabajar.

7) ¿Has acabado?

8) Estoy en ello.

9) Necesitaría también esto.

10) ¿Es crucial? Tardaría mucho tiempo.

11) Entonces no lo pongas.

12) Ya tengo esto. ¿Te sirve?

13) No del todo.

14) ¿Y esta nueva versión?

15) Si, es justo lo que necesito. Gracias.

16) De nada. Si tienes problemas, avísame.

Figura Nº 12 – Comunicación Cliente - Desarrollador.

Figura Nº 12 Extraída de [12]

Éste sería el desarrollo ideal de un proyecto XP. Para acercarnos a esto, se establece un ciclo de vida dividido en seis fases.

1. Fase de exploración
2. Fase de planificación
3. Fase de iteraciones
4. Fase de producción
5. Fase de mantenimiento
6. Fase de muerte del proyecto

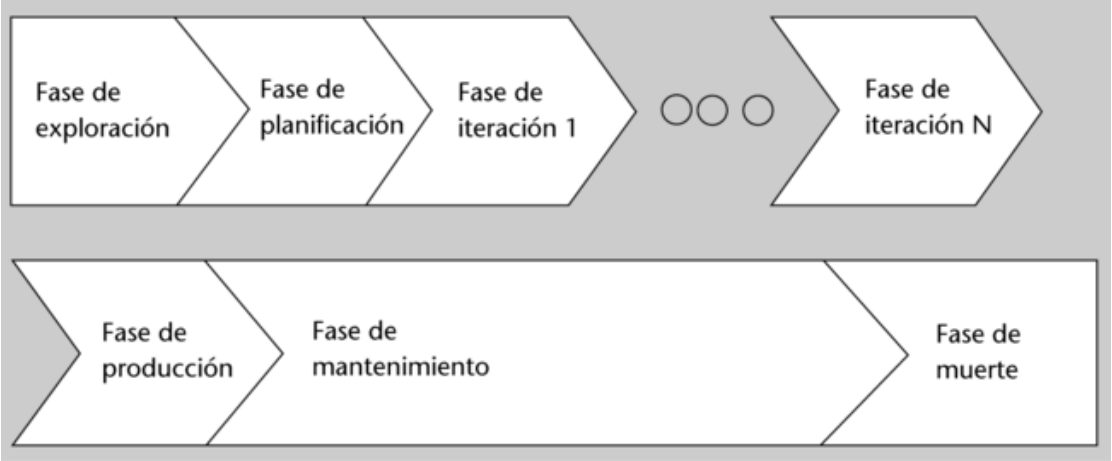

Figura Nº 13 – Fases del ciclo de vida de la Programación Extrema.

1. **La fase de exploración**: La fase de exploración es la primera fase del ciclo de vida de la metodología XP. En ella se desarrollan tres procesos:
    - Las historias del usuario.
    - El spike arquitectónico.
    - La metáfora del negocio.

Todo comienza con las “historias del usuario" (users stories). En esta fase los usuarios plantean a grandes rasgos las funcionalidades y requerimientos que desean obtener del aplicativo. Las historias de usuario tienen el mismo propósito que los casos de uso, salvo en un punto crucial, las escriben los usuarios y no el analista. Han de ser descripciones cortas y escritas en el lenguaje del usuario sin terminología técnica.

Estas historias son las que guiarán la creación de los tests de aceptación que han de garantizar que dichas historias se han comprendido y se han implementado correctamente.

Figura Nº 13 Extraída de [13]

No debemos confundir las historias de usuario con el análisis de requisitos, la principal diferencia está en la profundidad de análisis, con los requisitos queremos llegar al último detalle para no quedar mal frente al cliente, pero en XP el cliente forma parte del equipo y le podemos preguntar más cosas durante la implementación, así que el nivel de detalle en las historias de usuario ha de ser el mínimo imprescindible para que nos hagamos una idea general de la funcionalidad.

Las historias de usuario han de ser:

- Escritas por el cliente final, en su lenguaje y sin tecnicismos.
- Descripciones cortas de la usabilidad y funcionalidad que se espera del sistema.

Paralela y conjuntamente se empieza con el spike arquitectónico, en el que el equipo de desarrollo empieza a familiarizarse con la metodología, herramientas, lenguaje y codificaciones que se van a usar en el proyecto.

En el spike arquitectónico el equipo de desarrollo:

- Prueba la tecnología.
- Se familiariza con la metodología.
- Se familiariza con las posibilidades de la arquitectura.
- Realiza un prototipo que demuestre que la arquitectura es válida para el proyecto.

Una vez finalizadas las historias de usuario y el spike arquitectónico, se pasa a desarrollar conjuntamente la metáfora del negocio.

La metáfora del negocio:

- Es una historia común compartida por el usuario y el equipo de desarrollo.
- Debe servir para que el usuario se sienta a gusto refiriéndose al sistema en los términos de ella.
- Debe servir a los desarrolladores para implementar las clases y objetos del sistema.

2. **La fase de planificación**: el resultado ha de ser una planificación, de manera flexible, del proyecto.
   El procedimiento es el siguiente:
   - El cliente entrega al equipo de desarrollo las historias de usuario que ha confeccionado, pero priorizándolas de mayor a menor importancia.
   - El equipo de desarrollo las estudia y estima el coste de implementarlas.
   - Si el equipo de desarrollo considera que la historia de usuario es demasiado compleja, entonces el usuario final debe descomponerla en varias historias independientes más sencillas.
   - Si el equipo de desarrollo no ve claro cómo implementar una parte de la historia, el usuario puede realizar un spike tecnológico para ver cómo se podría implantar y así poder evaluar el coste.

- Una vez tenemos la lista de historias priorizadas junto con su coste de implementación, pasamos a convocar la reunión del plan de entregas.

El plan de entregas se compone de una serie de planes de iteración en el que se especifica qué funcionalidades se van a implementar en cada vuelta de la fase de iteraciones.

Participan de esta reunión tanto los usuarios como el equipo técnico y cada uno debe aportar su visión del negocio de manera que se obtengan más rápidamente aquellas funcionalidades que den el mayor beneficio para el negocio posible.

A cada iteración se le asigna un tiempo intentando que todas sean lo más parecido posible.

Se determina el alcance del proyecto.

3. **La fase de iteraciones**: como hemos dividido el proyecto en iteraciones, esta fase se repetirá tantas veces como iteraciones tengamos. Generalmente, cada iteración suele ser de dos a tres semanas.

El plan de iteración se trata de la siguiente manera:

- Se recogen las historias de usuario asignadas a esta iteración.
- Se detallan las tareas a realizar por cada historia de usuario.
- Las tareas deben ser de uno o tres días de desarrollo. Si son más grandes, se debería intentar dividir en varias más sencillas.
- Se estima el coste de cada tarea. Si el total es superior al tiempo de iteración, se deberá prescindir de alguna historia de usuario que se pasaría a la siguiente iteración. Si son muchas las historias de usuario desechadas, entonces hay que volver a estimar las cuatro variables de la metodología y volver a planificar el proyecto.
- Si el tiempo total estimado de las tareas es inferior al tiempo de iteración, se puede asumir una historia de usuario que correspondiese a la siguiente iteración.
- Se priorizan las tareas que más valor darán al negocio, intentando que se finalicen historias de usuario lo antes posible.
- Se reparten las primeras tareas al equipo de desarrollo y el resto se deja en una cola de tareas sin asignar de dónde se irán tomando a medida que se vayan finalizando las anteriores.

Se convocan reuniones de seguimiento diarias para ver si nos vamos retrasando en las estimaciones o nos vamos adelantando a ellas y así poder desechar o incorporar historias de usuario.

Lo más importante es que en cada momento de cada iteración estemos realizando la tarea que más valor posible da al negocio de entre las que tenemos pendientes, de manera que, si tenemos que reducir el alcance del proyecto, sólo afecte a las funcionalidades secundarias de nuestro aplicativo.

4. **La fase de producción**: llegamos a esta fase al alcanzar la primera versión que el usuario final decida que puede ponerse en producción.

   Pasaremos el aplicativo a producción cuando alcance las funcionalidades mínimas que aporten un valor real al negocio y una operativa arquitectónica estable.

   Es decir, no esperamos a tener todas las funcionalidades implementadas, sino que en cuanto tenemos algo que los usuarios pueden utilizar y que ayuda al negocio, pasamos la primera versión a producción.

   Paralelamente, se sigue con las iteraciones finales de proyecto. De esta manera, antes de que finalice el proyecto, ya estamos dando valor a la organización, el ROI (retorno de la inversión) del proyecto empieza a generarse antes de que éste finalice su versión final.

   En la etapa de producción se realizan también iteraciones como en la anterior etapa, pero el ritmo de éstas ya no es de dos a tres semanas, sino mensuales.

   Esta fase se mantiene hasta que realizamos la última entrega, con la que finalizamos el ámbito del aplicativo y pasamos al mantenimiento del mismo.

   Durante la fase de producción, el ritmo de desarrollo decae debido a que el equipo debe solventar las incidencias de los usuarios. Es por esto por lo que a veces es necesario incorporar nuevo personal al equipo.

5. **La fase de mantenimiento**: una vez el alcance del proyecto se ha conseguido, y tenemos todas las funcionalidades en producción, se revisan con el usuario aquellas nuevas historias de usuario que se han producido tras la puesta en producción del proyecto.

   Estas nuevas funcionalidades se van incorporando según su valor de negocio y el presupuesto adicional del que se disponga.

   El equipo de desarrollo se reduce a la mínima expresión, dejando algunos miembros para el mantenimiento.

6. **La fase de muerte del proyecto**: Cuando no existen más historias de usuario para introducir en nuestro sistema o cuando se reduce progresivamente valor de las historias de usuario implementadas en él, el proyecto entra en la fase de muerte.

   Se irá desinvirtiendo en él hasta abandonarlo totalmente cuando no aporte valor al negocio o cuando sus historias de usuario hayan sido absorbidas por otro sistema de información.

Los diferentes roles dentro de la Programación Extrema

Cada rol tiene unas funciones claras dentro de la metodología. Cada persona del equipo puede ejecutar uno o varios roles, o incluso cambiar de rol durante las diferentes fases del proyecto.

1. Programador:
    - Escribe las pruebas unitarias.
    - Produce el código del programa.

2. Cliente:
    - Escribe las historias de usuario.
    - Diseña las pruebas de aceptación.
    - Prioriza las historias de usuario.
    - Aporta la dimensión de negocio al equipo de desarrollo.
    - Representa al colectivo de usuarios finales.
    - Está siempre disponible para consultas.

3. Encargado de pruebas (tester):
    - Ejecuta las pruebas de aceptación.
    - Ayuda al cliente a diseñar pruebas de aceptación.
    - Ejecuta las pruebas de integración.
    - Difunde los resultados entre el equipo de desarrollo y el cliente.
    - Es el responsable automatizar los casos de prueba.
    - Encargado de seguimiento de los errores y sus diferentes estados.
    - Se encarga de realimentar todo el proceso de XP, midiendo las desviaciones con respecto a las estimaciones y comunicando los resultados para mejorar las siguientes estimaciones.
    - Realiza el seguimiento de cada iteración del proceso de XP tanto en la etapa de iteraciones como en la de producción.
    - Revalúa la posibilidad de incorporar o eliminar historias de usuario.

4. Líder Técnico
    - Se encarga del proceso global.
    - Garantiza que se sigue la filosofía de XP.
    - Conoce a fondo la metodología.
    - Provee guías y ayudas a los miembros del equipo a la hora de aplicar las prácticas básicas de XP.

5. Consultor:
    - No forma parte del equipo.
    - Tiene un conocimiento específico de un área en concreto.
    - Ayuda a resolver un problema puntual, ya sea de spike tecnológico o de valor de negocio.

6. PM (Project Manager):
    - Es el máximo responsable del proyecto.
    - Hace de enlace con los clientes.
    - Se encarga de coordinar y de garantizar las condiciones necesarias para el desarrollo del trabajo.

5.3.2. SCRUM

La metodología de trabajo de Scrum, tiene sus principios fundamentales en la década de 1980, la cual fue desarrollada por su necesidad en procesos de reingeniería por Goldratt, Takeuchi y Nonaka.

El concepto de Scrum tiene su origen sobre los nuevos procesos de desarrollo utilizados en productos exitosos en Japón y los Estados. Los equipos que desarrollaron estos productos partían de requisitos muy generales, así como novedosos, y debían salir al mercado en mucho menos del tiempo del que se tardó en lanzar productos anteriores. Estos equipos seguían patrones de ejecución de proyecto muy similares. En este estudio se comparaba la forma de trabajo de estos equipos altamente productivos y multidisciplinares con la colaboración entre los jugadores de Rugby y su formación de Scrum, de la cual se tomó su nombre.

Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos.

Podríamos decir que SCRUM se basa en cierto caos controlado pero establece ciertos mecanismos para controlar esta indeterminación, manipular lo impredecible y controlar la flexibilidad.

En Scrum se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto. Por ello, Scrum está especialmente indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos, donde la innovación, la competitividad, la flexibilidad y la productividad son fundamentales.

Scrum también se utiliza para resolver situaciones en que no se está entregando al cliente lo que necesita, cuando las entregas se alargan demasiado, los costes se disparan o la calidad no es aceptable, cuando se necesita capacidad de reacción ante la competencia, cuando la moral de los equipos es baja y la rotación alta, cuando es necesario identificar y solucionar ineficiencias sistemáticamente o cuando se quiere trabajar utilizando un proceso especializado en el desarrollo de producto.

Fundamentos de Scrum

Scrum se basa en los siguientes puntos:

- El desarrollo incremental de los requisitos del proyecto en bloques temporales cortos y fijos (iteraciones de un mes natural y hasta de dos semanas, si así se necesita).
  Las iteraciones se pueden entender como mini proyectos, en todas las iteraciones se repite un proceso de trabajo similar (de ahí el nombre "iterativo") para proporcionar un resultado completo sobre el producto final, de manera que el cliente pueda obtener los beneficios del proyecto de forma incremental. Para ello, cada requisito se debe completar en una única iteración. El equipo debe realizar todas las tareas necesarias para completarlo (incluyendo pruebas y documentación) y que esté preparado para ser entregado al cliente con el mínimo esfuerzo necesario. De esta manera no se deja para el final del proyecto ninguna actividad arriesgada relacionada con la entrega de requisitos.

- La priorización de los requisitos por valor para el cliente y coste de desarrollo en cada iteración. Para que un proyecto proporcione el mejor resultado posible, y como soporte fundamental al control empírico del proyecto, es necesario repriorizar los requisitos de manera regular, en cada iteración, según el valor que proporcionan al cliente en ese momento y su coste estimado de desarrollo. Como resultado de esta re priorización se actualiza la lista de requisitos priorizada (Product Backlog).

- El control empírico del proyecto. Por un lado, al final de cada iteración se demuestra al cliente el resultado real obtenido, de manera que pueda tomar las decisiones necesarias en función de lo que observa y del contexto del proyecto en ese momento. Por otro lado, el equipo se sincroniza diariamente y realiza las adaptaciones necesarias.

- La potenciación del equipo, que se compromete a entregar unos requisitos y para ello se le otorga la autoridad necesaria para organizar su trabajo.

- La sistematización de la colaboración y la comunicación tanto entre el equipo y como con el cliente.

- El timeboxing de las actividades del proyecto, para ayudar a la toma de decisiones y conseguir resultados. La técnica del timebox consiste en fijar el tiempo máximo para conseguir ciertos objetivos, tomar una decisión o realizar unas tareas, y hacer lo mejor que podamos en ese

intervalo. De esta manera, en lugar de ponerse a trabajar en algo hasta que esté hecho, de antemano se acuerda que sólo se dedica un tiempo limitado. La consciencia de esta limitación temporal favorece la priorización de objetivos/tareas y fuerza la toma de decisiones.

Requisitos para poder utilizar Scrum

Los siguientes puntos son de especial importancia para la implantación de una gestión ágil de proyectos como Scrum:

- Cultura de empresa basada en trabajo en equipo, delegación, creatividad y mejora continua.
- Compromiso del cliente en la dirección de los resultados del proyecto, gestión del ROI y disponibilidad para poder colaborar.
- Compromiso de la Dirección de la organización para resolver problemas frecuentes y realizar cambios organizativos, formando equipos autogestionados y multidisciplinares y fomentando una cultura de gestión basada en la colaboración y en la facilitación llevada a cabo por líderes al servicio del equipo.
- Compromiso conjunto y colaboración de los miembros del equipo.
- Relación entre proveedor y cliente basada en la colaboración y transparencia.
- Facilidad para realizar cambios en el proyecto.
- Tamaño de cada equipo entre 5 y 9 personas (con técnicas específicas de planificación y coordinación cuando varios equipos trabajan en el mismo proyecto).
- Equipo trabajando en un mismo espacio común para maximizar la comunicación.
- Dedicación del equipo a tiempo completo.
- Estabilidad de los miembros del equipo.

A continuación se realiza un detalle más específico sobre los requisitos de Scrum:

1. Cultura de empresa

La cultura de la empresa proveedora del proyecto debe estar alineada con la filosofía de una gestión ágil de proyectos como Scrum y debe fomentar:

- El trabajo en equipo y la colaboración entre todas las personas implicadas en un proyecto.
- Equipos autogestionados a los que se ha delegado la responsabilidad y autoridad para hacer su trabajo.
- La creatividad del equipo.
- La transparencia y la mejora continua, tanto del contexto de la organización y del proyecto y como de las herramientas del equipo.

Scrum sistematiza la identificación de obstáculos que pueden impedir el correcto progreso del proyecto. Los problemas previamente existentes en la organización (procesos, personas,

herramientas, etc.) y que atentan contra la productividad se harán más evidentes cuando se aplique Scrum y será necesario ir solucionándolos en cada iteración.

2. Compromiso del cliente

Scrum exige del cliente una alta implicación y una dedicación regular:

- El cliente tiene la responsabilidad de dirigir los resultados del producto o proyecto.
- El cliente debe disponer de una visión de alto nivel del producto o proyecto y tener reflejadas sus expectativas en forma de lista de requisitos priorizada donde ha indicado el valor que le aportará cada uno. A partir de los costes de desarrollo que le proporcione el equipo, priorizará los requisitos en función del Retorno de la Inversión (ROI) más rápido.
- El cliente re planifica el proyecto en cada iteración para maximizar este ROI de manera continua.
- Al tratarse de un proyecto que va entregando resultados en iteraciones regulares, el cliente debe colaborar participando en el inicio de cada iteración (reunión de planificación) y en el fin de cada iteración (demostración), y debe estar disponible durante la ejecución de cada iteración para resolver dudas.

3. Compromiso de la Dirección

La Dirección debe estar comprometida y apoyar el uso de la metodología:

- Se harán muy evidentes los obstáculos ya existentes y por venir que impiden el correcto desarrollo de los proyectos (a nivel de expectativas del cliente, productividad, calidad, etc.), sean organizativos, técnicos, procesos, relaciones entre personas/departamentos, habilidades de los equipos y demás.
- Será necesario tomar decisiones, realizar cambios organizativos, alinear a personas y proporcionar recursos para hacer la transición. Gestores y equipos deberán desaprender formas de trabajar y de relacionarse a las que estaban habituados y aprender nuevas dinámicas.
- Un proyecto ya no consistirá en que cada Departamento/Área realice su parte del trabajo y se la pase al siguiente. Será necesario formar equipos autogestionados y multidisciplinares capaces de conseguir un objetivo por ellos mismos.
- Habrá gestores que tendrán que cambiar sus roles para ser Facilitadores o Clientes, en una jerarquía de equipos organizada.
- Se tendrá que gestionar aquellas conductas personales que no permiten que otras personas puedan aportar ideas sobre el qué y el cómo de un proyecto, que defienden a toda costa su parcela de responsabilidad, que les cuesta mucho cederla al equipo y dejar de controlarlo, que no son capaces de delegar tareas o de colaborar con otras personas en la resolución de problemas.

4. Compromiso del equipo

Scrum se basa en el compromiso conjunto y la colaboración entre los miembros del equipo. La transparencia entre todos es fundamental para poder inspeccionar la situación real del proyecto y así poder hacer las mejores adaptaciones que permitan conseguir el objetivo común. Por ello, será difícil trabajar utilizando Scrum para las personas que:

- No confían en los demás, no permiten que otras personas puedan aportar ideas sobre el qué y el cómo, no son capaces de colaborar en la resolución de problemas ni de delegar tareas.
- No son transparentes respecto a su trabajo personal, sea por que defienden a toda costa su parcela de responsabilidad o por inseguridad para comunicarse o colaborar, cosa que no permite que sean ayudados.
- Su modo de relación se basa en la generación de conflicto o bien evitan entrar en conflictos sanos en que ambas partes ganen y terminar sin adquirir un compromiso real con el equipo.
- Priorizan su ego, sus intereses personales, de carrera o de departamento, por encima de los intereses del equipo.
- No son capaces de cambiar sus hábitos y salir de su zona de confort, tienen miedo al cambio, prefieren que se les diga qué tienen que hacer.
- Quieren seguir siendo los héroes que solucionan los proyectos y/o las personas de las que depende la empresa.

5. Relación entre proveedor y cliente

La relación entre el cliente y el proveedor del proyecto debe estar basada en el principio de obtener el mayor beneficio posible en todos los puntos del proyecto. En lugar de estar ligados por un contrato férreo de alcance, tiempo y coste, las dos partes asumen que habrá cambios para que cliente pueda obtener lo que realmente necesita, no lo que está escrito en un documento inicial o seguir un plan inicial que vaya perdiendo su sentido. La relación contractual se aproxima a un contrato de un equipo por meses, donde el cliente dirige mes a mes los resultados que el proyecto debe ir proporcionando. Debe existir transparencia en la ejecución del proyecto para facilitar esta relación. Esta transparencia la facilita de manera regular el propio proceso de Scrum, especialmente en la actividad de demostración de los requisitos completados al final de cada iteración.

6. Facilidad para realizar cambios en el proyecto

Para poder utilizar Scrum se debe poder ir incorporando requisitos de manera incremental en el producto del proyecto y realizar cambios de forma controlada sin un coste prohibitivo para el cliente. Para ello es necesario:

- Disponer de técnicas y herramientas que faciliten el crecimiento incremental y la introducción de cambios.

- Mantener la simplicidad y calidad interna del producto que se está creando. Para cubrir los requisitos actuales del cliente no hay que realizar más esfuerzo del que sea necesario y, a la vez, se debe vigilar de no hacer nada en contra de futuros requisitos.

Dado que los requisitos se desarrollan priorizados por su valor, es más improbable que ocurran cambios sustanciales en los primeros requisitos desarrollados, cuando se construye la base del producto. Se fomenta que los cambios que suelen aparecen cuando el proyecto ya está avanzado se realicen sobre requisitos que no son tan importantes.

La arquitectura emerge conforme se va necesitando, iteración a iteración, con lo que se asegura que todo lo que se diseña se utiliza y se prueba.

7. Tamaño del equipo

El tamaño de un equipo debe estar conformado entre 5 y 9 personas. Por debajo de 5 personas cualquier imprevisto o interrupción sobre un miembro del equipo compromete seriamente el compromiso que han adquirido y, por tanto, el resultado que se va a entregar al cliente al finalizar la iteración. Por encima de 9 personas, la comunicación y colaboración entre todos los miembros se hace más difícil y se forman subgrupos.

8. Equipo trabajando en un mismo espacio común

Todos los miembros del equipo deben trabajar en la misma localización física, para poder maximizar la comunicación entre ellos mediante conversaciones cara a cara, diagramas en pizarras, tarjetas en el tablón de tareas, etc. De esta manera se minimizan otros canales de comunicación menos eficientes (llamadas telefónicas, correos electrónicos, documentos), que hacen que las tareas se transformen en un pasa manos o que hacen perder el tiempo en el establecimiento de la comunicación.

9. Dedicación del equipo a tiempo completo

Los miembros del equipo deben dedicarse al proyecto a tiempo completo para de esta manera:

- Evitar dañar su productividad, que se vería afectada si tuviesen que ir cambiando de tarea para diferentes proyectos o duplicando el número de reuniones para estos diferentes proyectos. Si el equipo está dedicado a un único proyecto es más sencillo mantener el compromiso que adquiere en cada iteración. Como ayuda adicional para conseguir la máxima productividad, el Facilitador se encarga de proteger al equipo de interrupciones externas.
- Facilitar la gestión de recursos humanos de la organización. Esta gestión se simplifica si en la organización las personas se reservan a un proyecto por iteraciones completas.
- Por otro lado, el cliente y el facilitador deben estar dedicados al proyecto el tiempo necesario para cumplir con sus responsabilidades.

10. Estabilidad del equipo

El equipo debe ser estable durante el proyecto, sus miembros deben cambiar lo mínimo posible, para poder aprovechar el esfuerzo que les ha costado construir sus relaciones interpersonales, conectarse y establecer su organización del trabajo.

El proceso de aplicación de Scrum

En Scrum un proyecto se ejecuta en bloques temporales cortos y fijos (iteraciones de 2 a 4 semanas como máximo). Cada iteración tiene que proporcionar un resultado completo, un incremento de producto final que sea susceptible de ser entregado con el mínimo esfuerzo al cliente cuando lo solicite.

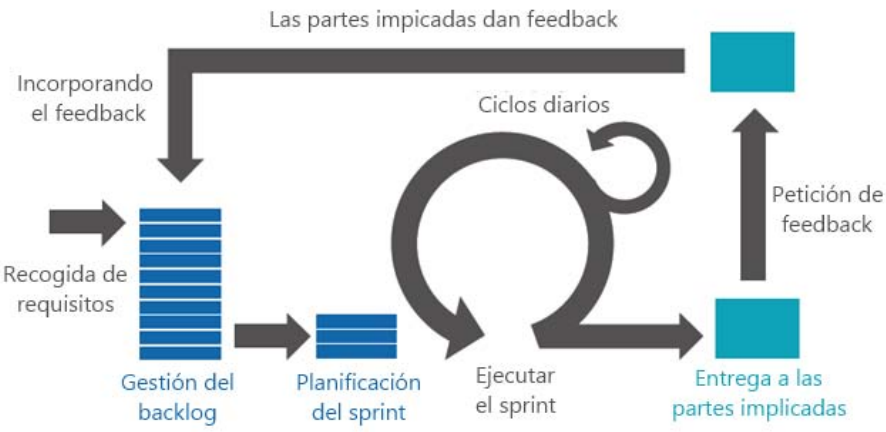

Figura Nº 14 – Proceso de Scrum.

El proceso parte de la lista de objetivos/requisitos priorizada del producto (Product Backlog), que actúa como plan del proyecto. En esta lista el cliente prioriza los objetivos balanceando el valor que le aportan respecto a su coste y quedan repartidos en iteraciones y entregas. De manera regular el cliente puede maximizar la utilidad de lo que se desarrolla y el retorno de inversión mediante la replanificación de objetivos del producto, que realiza durante la iteración con vista a las siguientes iteraciones.

Figura Nº 14 Extraída de [14]

Las actividades que se llevan a cabo en Scrum son las siguientes:

1. Planificación de la iteración (Sprint Planning)

La planificación de las tareas a realizar en la iteración se divide en dos partes:

- Primera parte de la reunión. Se realiza en un timebox promedio de 1 a 2 horas.
    - El cliente presenta al equipo la lista de requisitos priorizada (Product Backlog) del producto o proyecto, pone nombre a la meta de la iteración (de manera que ayude a tomar decisiones durante su ejecución) y propone los requisitos más prioritarios a desarrollar en ella.
    - El equipo examina la lista, pregunta al cliente las dudas que le surgen, añade más condiciones de satisfacción y selecciona los objetivos/requisitos más prioritarios que se compromete a completar en la iteración, de manera que puedan ser entregados si el cliente lo solicita.
- Segunda parte de la reunión. Se realiza en un timebox promedio de 1 a 2 horas. El equipo planifica la iteración, elabora la táctica que le permitirá conseguir el mejor resultado posible con el mínimo esfuerzo. Esta actividad la realiza el equipo dado que ha adquirido un compromiso, es el responsable de organizar su trabajo y es quien mejor conoce cómo realizarlo.
    - Define las tareas necesarias para poder completar cada objetivo/requisito, creando la lista de tareas de la iteración (Sprint Backlog).
    - Se realiza una estimación conjunta del esfuerzo necesario para realizar cada tarea (Pocker Planning).
    - Cada miembro del equipo se autoasigna a las tareas que puede realizar.

Beneficios

- Productividad mediante comunicación y creación de sinergias. Todos los miembros del equipo tienen una misma visión del objetivo y se ha utilizado los conocimientos y las experiencias de todos para elaborar la mejor solución entregable en el mínimo tiempo y con el mínimo esfuerzo, eliminando tareas innecesarias, detectando conflictos y dependencias entre tareas.
- Potenciación del compromiso del equipo sobre el objetivo común de la iteración:
    - Es todo el equipo quien asume la responsabilidad de completar en la iteración los requisitos que selecciona. Facilita la ayuda de cualquier miembro si se detecta algún impedimento que bloquea el progreso de la iteración, especialmente si cuando se está llegando al final de la iteración es necesaria la participación de todos para poder completar los objetivos comprometidos.

- Es cada una de las personas la que se responsabiliza de realizar sus tareas (a las que se autoasignó) en los tiempos que proporcionó. Si existe falta de compromiso con respecto al resto de miembros del equipo se hará muy evidente en las reuniones diarias de sincronización del equipo (Daily meeting).

- Una estimación conjunta es más fiable, dado que tiene en cuenta los diferentes conocimientos, experiencia y habilidades de los integrantes del equipo.

2. Ejecución de la iteración (Sprint)

En Scrum un proyecto se ejecuta en bloques temporales de tiempos cortos y fijos (iteraciones de 2 a 4 semanas). Cada iteración tiene que proporcionar un resultado completo, un incremento de producto que sea potencialmente entregable, de manera que cuando el cliente (Product Owner) lo solicite sólo sea necesario un esfuerzo mínimo para que el producto esté disponible para ser utilizado. Para ello, durante la iteración el equipo colabora estrechamente y se llevan a cabo las siguientes dinámicas:

- Cada día el equipo realiza una reunión de sincronización (Daily meeting), donde cada miembro inspecciona el trabajo de los otros para poder hacer las adaptaciones necesarias, comunica cuales son los impedimentos con que se encuentra, actualiza el estado de la lista de tareas de la iteración (Sprint Backlog) y los gráficos de trabajo pendiente (Burndown charts).
- El Facilitador (Scrum Master) se encarga de que el equipo pueda cumplir con su compromiso y de que no se reduzca su productividad. También se encarga de eliminar los obstáculos que el equipo no puede resolver por sí mismo y protege al equipo de interrupciones externas que puedan afectar su compromiso o su productividad.

Para poder completar el máximo de requisitos en la iteración, se debe minimizar el número de objetivos/requisitos en que el equipo trabaja simultáneamente, completando primero los que den más valor al cliente. Esta forma de trabajar, que se ve facilitada por la propia estructura de la lista de tareas de la iteración, permite tener más capacidad de reacción frente a cambios o situaciones inesperadas.

3. Reunión diaria de sincronización del equipo (Daily meeting)

El objetivo de esta reunión es facilitar la transferencia de información y la colaboración entre los miembros del equipo para aumentar su productividad, al poner de manifiesto puntos en que se pueden ayudar unos a otros.

Cada miembro del equipo inspecciona el trabajo que el resto está realizando (dependencias entre tareas, progreso hacia el objetivo de la iteración, obstáculos que pueden impedir este objetivo) para al finalizar la reunión poder hacer las adaptaciones necesarias que permitan cumplir con el compromiso conjunto que el equipo adquirió para la iteración (en la reunión de planificación de la iteración).

En esta reunión, la cual es dirigida por el Scrum Master, cada miembro del equipo debe responder las siguientes preguntas en un timebox de cómo máximo 15 minutos:

- ¿Qué he hecho desde la última reunión de sincronización? ¿Pude hacer todo lo que tenía planeado? ¿Cuál fue el problema?
- ¿Qué voy a hacer a partir de este momento?
- ¿Qué impedimentos tengo o voy a tener para cumplir mis compromisos en esta iteración y en el proyecto?

Como apoyo a la reunión, el equipo cuenta con la lista de tareas de la iteración, donde se actualiza el estado y el esfuerzo pendiente para cada tarea, así como con el gráfico de horas pendientes en la iteración.

Recomendaciones

- Realizar la reunión diaria de sincronización de pie, para que los miembros del equipo no se relajen ni se extiendan en más detalles de los necesarios.
- Realizar las reuniones de colaboración entre miembros del equipo justo después de la de sincronización.

4. Demostración de requisitos completados (Sprint Review)

Es una reunión informal donde el equipo presenta al cliente los requisitos completados en la iteración, en forma de incremento de producto preparado para ser entregado con el mínimo esfuerzo, haciendo un recorrido por ellos lo más real y cercano posible al objetivo que se pretende cubrir.

En función de los resultados mostrados y de los cambios que haya habido en el contexto del proyecto, el cliente realiza las adaptaciones necesarias de manera objetiva, ya desde la primera iteración, re planificando el proyecto.

Se realiza en un timebox de 1 a 2 horas promedio.

Beneficios

- El cliente puede ver de manera objetiva cómo han sido desarrollados los requisitos que proporcionó, ver si se cumplen sus expectativas, entender más qué es lo que necesita y tomar mejores decisiones respecto al proyecto.
- El equipo puede ver si realmente entendió cuáles eran los requisitos que solicitó el cliente y ver en qué puntos hay que mejorar la comunicación entre ambos.
- El equipo se siente más satisfecho cuando puede ir mostrando los resultados que va obteniendo. No está meses trabajando sin poder exhibir su obra.

Restricciones

- Sólo se pueden mostrar los requisitos completados, para que el cliente no se haga falsas expectativas y pueda tomar decisiones correctas y objetivas en función de la velocidad de desarrollo y el resultado realmente completado. Un requisito no completado quedará como un requisito más a re planificar.

5. Retrospectiva (Sprint Retrospective)

Con el objetivo de mejorar de manera continua su productividad y la calidad del producto que se está desarrollando, el equipo analiza cómo ha sido su manera de trabajar durante la iteración, por qué está consiguiendo o no los objetivos a los que se comprometió al inicio de la iteración y por qué el incremento de producto que acaba de demostrar al cliente era lo que él esperaba o no.

- Que se hizo bien.
- Que hay que mejorar.
- Qué he aprendido.
- Cuáles son los problemas que podrían impedirle progresar adecuadamente.

El Facilitador se encargará de ir eliminando los obstáculos identificados que el propio equipo no pueda resolver por sí mismo.

Notar que esta reunión se realiza después de la reunión de demostración al cliente de los objetivos conseguidos en la iteración, para poder incorporar su feedback y cumplimiento de expectativas como parte de los temas a tratar en la reunión de retrospectiva. Se realiza en un timebox de cómo máximo 2 horas.

Beneficios

- Incrementa la productividad en el proyecto, la calidad del producto (cosa que permite hacerlo crecer de manera sostenida) y potencia el aprendizaje del equipo de manera sistemática, iteración a iteración, con resultados a corto plazo.
- Aumenta la motivación del equipo dado que participa en la mejora de proceso, se siente escuchado, toma decisiones consensuadas (y más sostenibles) para ir eliminando lo que molesta e impide que sea más productivo.

6. Replanificación del proyecto (Product Backlog Refinement)

En las reuniones de planificación de entregas y durante el transcurso de una iteración (en el Product Backlog Refinement), el cliente va trabajando en la lista de objetivos/requisitos priorizada del producto o proyecto, añadiendo requisitos, modificándolos, eliminándolos, re priorizándolos, cambiando el contenido de iteraciones y definiendo un calendario de entregas que se ajuste mejor a sus nuevas necesidades.

Los cambios en la lista de requisitos pueden ser debidos a:

- Modificaciones que el cliente solicita tras la demostración que el equipo realiza al final de cada iteración sobre los resultados obtenidos, ahora que el cliente entiende mejor el producto o proyecto.
- Cambios en el contexto del proyecto (sacar al mercado un producto antes que su competidor, hacer frente a urgencias o nuevas peticiones de clientes, etc).
- Nuevos requisitos o tareas como resultado de nuevos riesgos en el proyecto.

Para realizar esta tarea, el cliente colabora con el equipo:

- Para cada nuevo objetivo/requisito, conjuntamente se hace una identificación inicial de sus condiciones de satisfacción (que se detallarán en la reunión de planificación de la iteración).
- El cliente obtiene del equipo la re estimación de costes de desarrollo para completar los objetivos/requisitos siguientes. El equipo ajusta el factor de complejidad, el coste para completar los requisitos y su velocidad de desarrollo en función de la experiencia adquirida hasta ese momento en el proyecto.
- El cliente re prioriza la lista de objetivos/requisitos en función de estas nuevas estimaciones.

Roles y responsabilidades de las personas involucradas en la metodología

1. Cliente (Product Owner)

Las responsabilidades del Cliente son:

- Ser el representante de todas las personas interesadas en los resultados del proyecto (internas o externas a la organización, promotores del proyecto y usuarios finales o consumidores finales del producto) y actuar como interlocutor único ante el equipo, con autoridad para tomar decisiones.
- Definir los objetivos del producto o proyecto.
- Dirigir los resultados del proyecto y maximizar su ROI.
- Es el propietario de la planificación del proyecto. Crea y mantiene la lista priorizada con los requisitos necesarios para cubrir los objetivos del producto o proyecto, conoce el valor que aportará cada requisito y calcula el ROI a partir del coste de cada requisito que le proporciona el equipo.
- Reparte los objetivos/requisitos en iteraciones y establece un calendario de entregas.
- Antes de iniciar cada iteración re planifica el proyecto en función de los requisitos que aportan más valor en ese momento, de los requisitos completados en la iteración anterior y del contexto del proyecto en ese momento
- Colaborar con el equipo para planificar, revisar y dar detalle a los objetivos de cada iteración.

- Participar en la reunión de planificación de iteración, proponiendo los requisitos más prioritarios a desarrollar, respondiendo a las dudas del equipo y detallando los requisitos que el equipo se comprometer a hacer.
- Estar disponible durante el curso de la iteración para responder a las preguntas o dudas que puedan surgir.
- No cambiar los requisitos que se están desarrollando en una iteración, una vez ésta esta iniciada.

2. Facilitador (Scrum Master)

Se encarga de liderar al equipo llevando a cabo las siguientes responsabilidades:

- Controlar que todos los participantes del proyecto sigan los valores y principios ágiles, las reglas y proceso de Scrum y guiar la colaboración dentro del equipo y con el cliente de manera que las sinergias sean máximas.
- Asegurar que exista una lista de requisitos priorizada y que esté preparada antes de la siguiente iteración.
- Facilitar las reuniones de Scrum (planificación de la iteración, reuniones diarias de sincronización del equipo, demostración, retrospectiva), de manera que sean productivas y consigan sus objetivos.
- Enseñar al equipo a auto gestionarse. No da respuestas, si no que guía al equipo con preguntas para que descubra por sí mismo una solución.
- Quitar los impedimentos que el equipo tiene en su camino para conseguir el objetivo de cada iteración, proporcionar un resultado útil al cliente de la manera más efectiva y así poder finalizar el proyecto con éxito. Estos obstáculos se identifican de manera sistemática en las reuniones diarias de sincronización del equipo y en las reuniones de retrospectiva.
- Proteger y aislar al equipo de interrupciones externas durante la ejecución de la iteración (introducción de nuevos requisitos, desvinculación no prevista de un miembro del equipo, etc.). De esta manera, el equipo puede mantener su productividad y el compromiso que adquirió sobre los requisitos que completaría en la iteración.

3. Equipo (Team)

Es el grupo de personas que de manera conjunta desarrollan el producto del proyecto. Tienen un objetivo común y comparten la responsabilidad del trabajo que realizan.

El tamaño del equipo está entre 5 y 9 personas. Por debajo de 5 personas cualquier imprevisto o interrupción sobre un miembro del equipo compromete seriamente el compromiso que han adquirido y, por tanto, el resultado que se va a entregar al cliente al finalizar la iteración. Por encima de 9 personas, la comunicación y colaboración entre todos los miembros se hace más difícil y se forman subgrupos.

Este equipo está constituido por los siguientes perfiles:

- Desarrollador
- Analista funcional
- Analista Técnico
- Tester
- Arquitecto
- Proyect Manager
- Team Leader

Es un equipo auto-organizado, que comparte información y cuyos miembros confían entre ellos. Realiza de manera conjunta las siguientes actividades:

- Seleccionar los requisitos que se compromete a completar en una iteración, de forma que estén preparados para ser entregados al cliente.
- Estimar la complejidad de cada requisito en la lista de requisitos priorizada del producto o proyecto.
- En la reunión de planificación de la iteración decide cómo va a realizar su trabajo:
    - Seleccionar los requisitos que pueden completar en cada iteración, realizando al cliente las preguntas necesarias.
    - Identificar todas las tareas necesarias para completar cada requisito.
    - Estimar el esfuerzo necesario para realizar cada tarea.
    - Cada miembro del equipo se auto asigna a las tareas.
- Durante la iteración, trabajar de manera conjunta para conseguir los objetivos de la iteración. Cada especialista lidera el trabajo en su área y el resto colaboran si es necesario para poder completar un requisito.
- Al finalizar la iteración:
    - Demostrar al cliente los requisitos completados en cada iteración.
    - Hacer una retrospectiva la final de cada iteración para mejorar de forma continua su manera de trabajar.

Herramientas que se utilizan en Scrum

- *Lista de objetivos / requisitos priorizada (Product Backlog)*

La lista de objetivos/requisitos priorizada representa la visión y expectativas del cliente respecto a los objetivos y entregas del producto o proyecto. El cliente es el responsable de crear y gestionar la lista con la ayuda del Facilitador y del equipo, quien proporciona el coste estimado de completar cada requisito.

- *Lista de tareas de la iteración (Sprint Backlog)*

Es una lista de tareas que el equipo elabora en la reunión de planificación de la iteración (Sprint planning) como plan para completar los objetivos/requisitos seleccionados para la iteración y que se compromete a demostrar al cliente al finalizar la iteración, en forma de incremento de producto preparado para ser entregado. A cada tarea se la suele llamar Historia (History)

- *Gráficos de trabajo pendiente (Burndown charts)*

Consiste en un gráfico que indica el trabajo pendiente a lo largo del tiempo y muestra la velocidad a la que se estan completando los objetivos/requisitos. Permite predecir si el equipo podrá completar el trabajo en el tiempo estimado.

Ventajas de utilizar Scrum

- Entregas parciales a corto plazo de resultados
- Gestión regular de las expectativas del cliente y basada en resultados tangibles.
- Resultados anticipados.
- Flexibilidad y adaptación respecto a las necesidades del cliente, cambios en el mercado, etc.
- Gestión sistemática del Retorno de Inversión (ROI).
- Mitigación sistemática de los riesgos del proyecto.
- Productividad y calidad.
- Alineamiento entre el cliente y el equipo de desarrollo.
- Equipo motivado.

5.3.3. CRYSTAL

Crystal no es solo una metodología de desarrollo de software ágil, ya que se la considera una familia de metodologías, debido a que se subdivide en varios tipos en función a la cantidad de persona involucradas en el proyecto. Esta nueva serie de metodologías fue creada por el antropólogo Alistair Cockburn, el cuál tomo como base el análisis de distintos proyectos de desarrollo de software y su propia experiencia, lo cual fusionando ambos aspectos dio lugar este nuevo método de trabajo. Estas metodologías 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 Extreme Programming.

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 UI (Interface de Usuario) y para participar en la definición de los requerimientos funcionales y no funcionales del software.

La familia Crystal dispone de un código de color para marcar la complejidad de una metodología, ya que cuanto más oscuro es el color, más pesado es el método y cuanto más crítico es un sistema, más rigor se requiere. El código cromático se aplica a una forma tabular que se usa para situar el rango de complejidad al cual se aplica una metodología.

El nombre Crystal deriva de la caracterización de los proyectos según 2 dimensiones, tamaño y complejidad.

En la siguiente imagen se puede apreciar la escala de colores dependiendo de la complejidad del sistema:

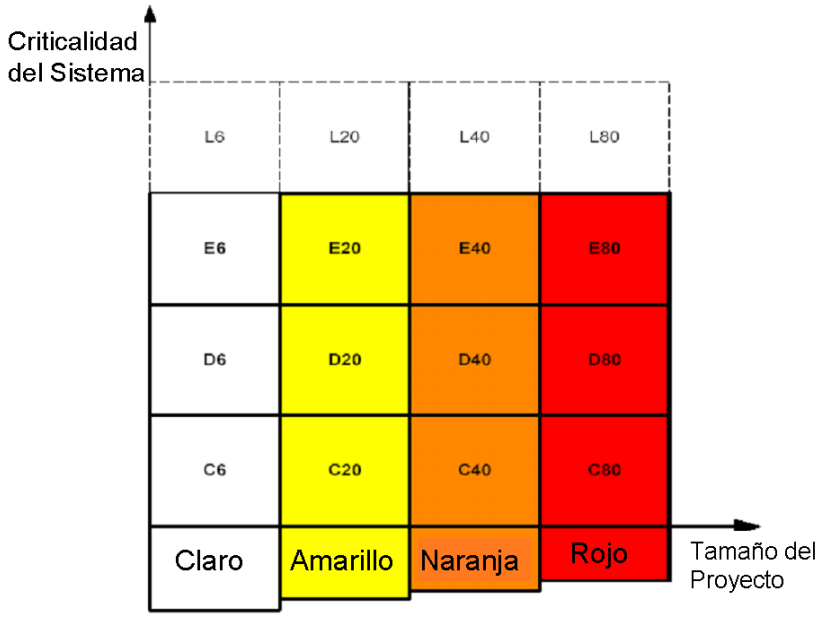

Figura Nº 15 – Familia de Metodologías Crystal.

- Claro (Clear) es para equipos de hasta 8 personas o menos.
- Amarillo para equipos entre 10 a 20 personas.
- Naranja para equipos entre 20 a 50 persona.
- Roja para equipos de más de 50 personas.

Se trata de un conjunto de metodologías para el desarrollo de software caracterizadas por estar centradas en las personas que componen el equipo y la reducción al máximo del número de artefactos producidos. El desarrollo de software se considera un juego cooperativo de invención y comunicación, limitado por los recursos a utilizar. El equipo de desarrollo es un factor clave, por lo que se deben invertir esfuerzos en mejorar sus habilidades y destrezas, así como tener políticas de trabajo en equipo o definidas.

Figura Nº 15 Extraída de [15]

Propiedades de Crystal Clear:

- 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 o mensual.
- Comunicación osmótica: todos juntos en el mismo cuarto. Una variante especial es disponer en la sala de un experto diseñador senior y discutir respecto del tema que se trate.
- Mejora reflexiva: tomarse un pequeño tiempo diariamente para pensar bien qué se está haciendo, cotejar notas, reflexionar, discutir.
- Seguridad personal: hablar con los compañeros cuando algo molesta dentro del grupo.
- Foco: saber lo que se está haciendo y tener la tranquilidad y el tiempo para hacerlo.
- Fácil acceso a usuarios expertos: tener alguna comunicación con expertos desarrolladores.

Roles:

- Patrocinador: produce la Declaración de Misión con Prioridades de Compromiso (Tradeoff). Consigue los recursos y define la totalidad del proyecto.
- Usuario Experto: junto con el Experto en Negocios produce la Lista de Actores-Objetivos y el Archivo de Casos de Uso y Requerimientos. Debe familiarizarse con el uso del sistema, sugerir atajos de teclado, modos de operación, información a visualizar simultáneamente, navegación.
- Diseñador Principal: produce la Descripción Arquitectónica. Se supone que debe ser al menos un profesional de Nivel 3. En Metodologías Ágiles se definen tres niveles de experiencia:
    - Nivel 1 es capaz de seguir los procedimientos.
    - Nivel 2 es capaz de apartarse de los procedimientos específicos y encontrar otros distintos.
    - Nivel 3 es capaz de manejar con fluidez, mezclar e inventar procedimientos.

  El Diseñador Principal tiene roles de coordinador, arquitecto, mentor y programador más experto.
- Diseñador/Programador: produce, junto con el Diseñador Principal, los Borradores de Pantallas, el Modelo Común de Dominio, las Notas y Diagramas de Diseño, el Código Fuente, el Código de Migración, las Pruebas y el Sistema Empaquetado.
- Experto en Negocios: junto con el Usuario Experto produce la Lista de Actores/Objetivos y el Archivo de Casos de Uso y Requerimientos. Debe conocer las reglas y políticas del negocio.
- Coordinador: con la ayuda del equipo, produce el Mapa de Proyecto, el Plan de Entrega, el Estado del Proyecto, la Lista de Riesgos, el Plan y Estado de Iteración y la Agenda de Visualización.
- Verificador: produce el Reporte de errores. Puede ser un programador en tiempo parcial, o un equipo de varias personas.

- Escritor: produce el Manual de Usuario. El Equipo como Grupo es responsable de producir la Estructura y Convenciones del Equipo y los Resultados del Taller de Reflexión.

La guía de trabajo que presenta Crystal Clear es altamente recomendable para equipos pequeños. Da flexibilidad y prioriza la parte humana, apuntando a lograr eficiencia, habitabilidad y confianza en los miembros del equipo.

Presta especial importancia a la ubicación física del grupo, donde la comunicación cumple el principal rol.

La entrega frecuente de código confiable mantiene el foco y evita distracciones.

5.3.4. Kanban

Esta metodología tiene como base de su origen la aplicación de los procesos de producción JIT (Just in Time) ideados por la empresa automotriz Toyota, en la cual utilizaban tarjetas visuales para identificar necesidades de material en la cadena de producción.

Kanban se basa en la idea de que el trabajo en curso debería limitarse, y sólo deberíamos empezar con algo nuevo cuando un bloque de trabajo anterior haya sido entregado o ha pasado a otra función posterior de la cadena.

La metodología Kanban utiliza un mecanismo de control visual para hacer seguimiento del trabajo conforme este viaja a través del flujo de valor. Normalmente, se emplea un panel o pizarra con notas adhesivas o un panel electrónico de tarjetas para gestionar el flujo de trabajo y las asignaciones. Las mejores prácticas apuntan al uso de ambos métodos.

El Kanban (o tarjeta visual) implica que se genera una señal visual para indicar que hay nuevos bloques de trabajo que pueden ser comenzados porque el trabajo en curso actual no alcanza el máximo acordado.

El aporte principal de Kanban a las metodologías agiles es que a través de la implementación de tarjetas visuales, proporciona transparencia al proceso, ya que su flujo de trabajo expone los cuellos de botella, colas, variabilidad y desperdicios a lo largo del tiempo y todas las cosas que impactan al rendimiento de la organización en términos de la cantidad de trabajo entregado y el ciclo de tiempo requerido para entregarlo. Proporciona a los miembros del equipo y a las partes interesadas visibilidad sobre los efectos de sus acciones o falta de acción. De esta forma, cambia el comportamiento y motiva a una mayor colaboración en el trabajo. La visibilidad de los cuellos de botella, desperdicios y variabilidades y su impacto también promueve la discusión sobre las posibles mejoras, y hace que los equipos comiencen rápidamente a implementar mejoras en su proceso.

Como resultado, Kanban propicia la evolución incremental de los procesos existentes, una evolución que generalmente está alineada con los valores de las metodologías agiles. Kanban no genera una revolución radical de la forma en la que las personas trabajan, sino que sugiere un cambio gradual. Es un cambio que surge del entendimiento y del consenso de entre todos los participantes del proyecto.

Las principales ventajas de esta metodología es que es muy fácil de aplicar, actualizar y asumir por parte del equipo. Además, se destaca por utilizar una técnica de gestión de las tareas muy visual y práctica, lo que permite ver a simple vista el estado de los proyectos, así como también pautar el desarrollo del trabajo de manera efectiva.

Principios del método Kanban

- Calidad garantizada: todo lo que se hace debe salir bien en la primera instancia, no hay margen de error. Para lograr esto no se premia la rapidez, sino la calidad final de las tareas realizadas. Esto se basa en el hecho que muchas veces cuesta más arreglarlo después que hacerlo bien de entrada.
- Reducción del desperdicio: se basa en hacer solamente lo justo y necesario, para garantizar que se haga bien. Esto supone la reducción de todo aquello que es superficial o secundario.
- Mejora continua: Se acuerda perseguir el cambio incremental y evolutivo. No es simplemente un método de gestión, sino también un sistema de mejora en el desarrollo de proyectos, según los objetivos a alcanzar.
  El equipo debe estar de acuerdo que el cambio continuo, gradual y evolutivo es la manera de hacer mejoras en el sistema y debe apegarse a ello. Los cambios radicales pueden parecer más eficaces, pero tienen una mayor tasa de fracaso debido a la resistencia y el miedo en la organización.
- Flexibilidad: es necesario poder priorizar aquellas tareas entrantes según las necesidades del momento y tener la capacidad de dar respuesta a estas tareas imprevistas.

Aplicación del método Kanban

La aplicación del método Kanban implica la generación de un tablero de tareas que permitirá mejorar el flujo de trabajo y alcanzar un ritmo sostenible. Para implantar esta metodología, deberemos tener claro los siguientes aspectos:

1. Definir el flujo de trabajo de los proyectos: para ello, simplemente deberemos crear nuestro propio tablero, que deberá ser visible y accesible por parte de todos los miembros del equipo. Cada una de las columnas corresponderá a un estado concreto del flujo de tareas, que nos servirá para saber en qué situación se encuentra cada proyecto. El tablero debe tener tantas columnas como estados por los que pasa una tarea, desde que se inicia hasta que finaliza. A diferencia de SCRUM, una de las peculiaridades del tablero es que este es continuo. Esto significa que no se compone de tarjetas que se van desplazando hasta que la actividad queda realizada por completo. En este caso, a medida que se avanza, las nuevas tareas (mejoras, incidencias o nuevas funcionalidades) se acumulan en la sección inicial, de manera que en las reuniones periódicas con el cliente se priorizan y se colocan dentro de la sección que se estima oportuna.

Dicho tablero puede ser específico para un proyecto en concreto o bien genérico. No hay unas fases del ciclo de producción establecidas sino que se definirán según el caso en cuestión, o se establecerá un modelo aplicable genéricamente para cualquier proyecto de la organización.

2. Visualizar las fases del ciclo de producción. Al igual que Scrum, Kanban se basa en el principio de desarrollo incremental, dividiendo el trabajo en distintas partes. Esto significa que no hablamos de la tarea en sí, sino que lo dividimos en distintos pasos para agilizar el proceso de producción.
   Normalmente cada una de esas partes se escribe en un post-it y se pega en el tablero, en la fase que corresponda. Dicho post-it contiene, normalmente, la descripción de la tarea con la estimación de horas, la información básica para que el equipo sepa rápidamente la carga total de trabajo que supone. Además, se pueden emplear fotos para asignar responsables así como también usar tarjetas con distintas formas para poner observaciones o indicar bloqueos (cuando una tarea no puede hacerse porqué depende de otra).
   Al final, el objetivo de la visualización es clarificar al máximo el trabajo a realizar, las tareas asignadas a cada equipo de trabajo (o departamento), así como también las prioridades y la meta asignada.

3. Stop Starting, start finishing. Este es el lema principal de la metodología Kanban. De esta manera, se prioriza el trabajo que está en curso en vez de empezar nuevas tareas. Precisamente, una de las principales aportaciones del Kanban es que el trabajo en curso debe estar limitado y, por tanto, existe un número máximo de tareas a realizar en cada fase; no se puede abrir una nueva tarea sin finalizar otra.
   De esta manera, se pretende dar respuesta al problema habitual de muchas empresas de tener muchas tareas abiertas pero con un promedio de finalización muy bajo. Aquí lo importante es que las tareas que se abran se cierren antes de empezar con la siguiente.

4. Control del Flujo. A diferencia de SCRUM, la metodología Kanban no se aplica a un único proyecto, sino que mezcla tareas y proyectos. Se trata de mantener a los trabajadores con un flujo de trabajo constante, las tareas más importantes en cola para ser desarrolladas y un seguimiento pasivo para no tener que interrumpir al trabajador en cada momento.
   Asimismo, dicha metodología de trabajo nos permite hacer un seguimiento del trabajo realizado, almacenando la información que nos proporcionan las tarjetas.

Las tres reglas de Kanban

1. Mostrar el proceso
2. Limitar el trabajo en curso
3. Optimizar el flujo de trabajo

1. Mostrar el proceso

Consiste en la visualización de todo el proceso de desarrollo, mediante un tablero físico, públicamente asequible. El objetivo de mostrar el proceso, consiste en:

- Entender mejor el proceso de trabajo actual.
- Conocer los problemas que puedan surgir y tomar decisiones.
- Mejorar la comunicación entre todos los interesados/participantes del proyecto.
- Hacer los futuros procesos más predecibles.

Un tablero Kanban, se divide en columnas las cuales representan un proceso de trabajo. Un ejemplo clásico de columnas para dividir un tablero Kanban, sería el siguiente:

Cola de entrada | Análisis | Desarrollo | Test | Deploy

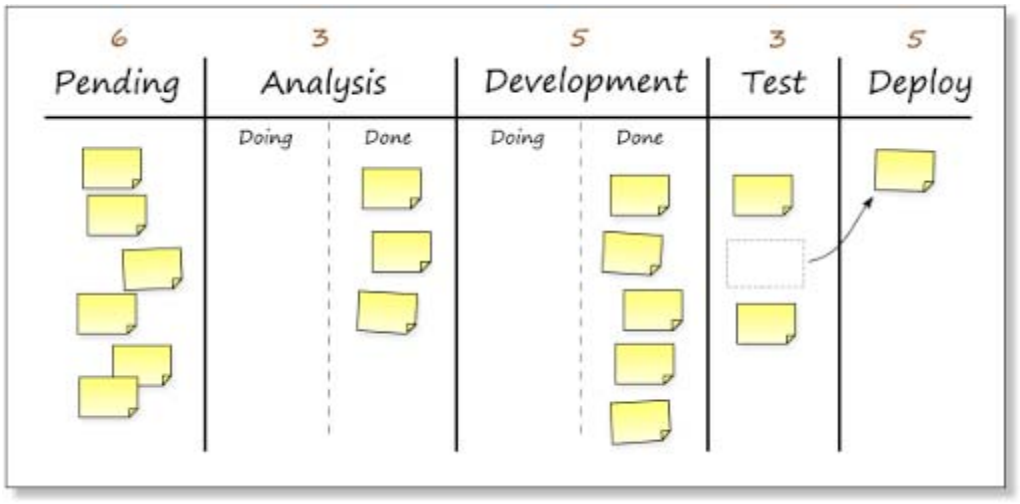

Figura Nº 16 – Tablero Kanban.

La cantidad y nombre de las columnas, varía de acuerdo a las necesidades de cada equipo y en la mayoría de los casos, éstas, son subdivididas en dos columnas: cola de espera y en curso.

2. Limitar el trabajo en curso

Los límites del trabajo en curso consisten en acordar anticipadamente, la cantidad de ítems que pueden abordarse por cada proceso (es decir, por columnas del tablero). El principal objetivo de establecer estos límites, es el de detectar cuellos de botella que representan el estancamiento de un proceso determinado.

Figura Nº 16 Extraída de [16]

Es un valor a tener en cuenta, que la resolución de cuellos de botella, la mayoría de las veces, motiva la colaboración del equipo entre los diferentes procesos. Pues mientras existen procesos colapsados, existen a la vez, procesos libres para aceptar nuevos ítems. El cuello de botella ha generado un estancamiento, y los procesos libres, pueden ayudar a destrabar a los procesos colapsados.

3. Optimizar el flujo de trabajo

El objetivo es generar una producción estable, continua y previsible. Midiendo el tiempo que el ciclo completo de ejecución del proyecto demanda (por ejemplo, cantidad de días desde el inicio del análisis hasta el fin de la implementación), se obtiene el CycleTime.

Al dividir, el CycleTime por el WIP (trabajo en curso), se obtiene el rendimiento de trabajo, denominado Throughput, es decir, la cantidad de ítems que un equipo puede terminar en un determinado período de tiempo.

Con estos valores, la optimización del flujo de trabajo consistirá en la búsqueda de:

- Minimizar el CycleTime
- Maximizar el Throughput
- Lograr una variabilidad mínima entre CycleTime y Throughput

Dicho todo esto, se puede decidir que implementando Kanban se consigue aumentar la eficiencia en los procesos, evitar retrasos y no desaprovechar recursos, reducción de tiempos muertos en y entre procesos, mejor mantenimiento, información más rápida y precisa, minimización de entregas con errores y evitar sobrecarga de trabajo.

5.3.5. FEATURE DRIVEN DEVELOPMENT (FDD)

La metodología ágil FDD, con sus siglas en inglés Feature Driven Development, fue impulsada por Jeff de Luca y Meter Coad en los años 80.

Como las otras metodologías adaptables, se enfoca en iteraciones cortas que entregan funcionalidad tangible. Dicho enfoque no hace énfasis en la obtención de los requerimientos sino en cómo se realizan las fases de diseño y construcción. Sin embargo, fue diseñado para trabajar con otras actividades de desarrollo de software y no requiere la utilización de ningún modelo de proceso específico. Hace énfasis en aspectos de calidad durante todo el proceso e incluye un monitoreo permanente del avance del proyecto. Al contrario de otras metodologías, FDD promete ser conveniente para el desarrollo de sistemas críticos y está orientada a equipos de trabajo más grandes, con más personas que aquellos a los que normalmente se aplican otras metodologías como Scrum.

Define claramente entregas tangibles y formas de evaluación del progreso del proyecto. No hace énfasis en la obtención de los requerimientos sino en cómo se realizan las fases de diseño y construcción.

FDD se basa en un ciclo muy corto de iteración, nunca superior a dos semanas, y en el que el análisis y los desarrollos están orientados a cumplir una lista de Features que tiene que tener el software a desarrollar.

Un Feature debe cumplir las siguientes características:

- Debe ser simple y poco costosa de desarrollar, de entre uno y diez días.
- Debe aportar valor al cliente y ser relevante para su negocio.
- Debe poderse expresar en términos de acción, resultado y objeto.

La metodología sigue cinco fases iterativas:

1. Desarrollo/Modificación de un Modelo Global
2. Creación/Modificación de la lista de Features
3. Planificación
4. Diseño
5. Implementación

Ventajas

- El equipo de desarrollo no malgasta el tiempo y dinero del cliente desarrollando soluciones innecesariamente generales y complejas que en realidad no son un requisito del cliente.
- Cada componente del producto final ha sido probado y satisface los requerimientos.
- Rápida respuesta a cambios de requisitos a lo largo del desarrollo.
- Entrega continua y en plazos cortos de software funcional.
- Trabajo conjunto entre el cliente y el equipo de desarrollo.
- Minimiza los costos frente a cambios.
- Importancia de la simplicidad, al eliminar el trabajo innecesario.
- Atención continua a la excelencia técnica y al buen diseño.
- Mejora continua de los procesos y el equipo de desarrollo.
- Evita malentendidos de requerimientos entre el cliente y el equipo.

Desventajas

- Falta de documentación del diseño. El código no puede tomarse como una documentación. En sistemas de tamaño grande se necesita leer los cientos o miles de páginas del listado de código fuente.
- Problemas derivados de la comunicación oral. Este tipo de comunicación resulta difícil de preservar cuando pasa el tiempo y está sujeta a muchas ambigüedades.
- Fuerte dependencia de las personas. Como se evita en lo posible la documentación y los diseños convencionales, los proyectos ágiles dependen críticamente de las personas.
- Falta de reusabilidad. La falta de documentación hacen difícil que pueda reutilizarse el código ágil.

5.3.6. ADAPTIVE SOFTWARE DEVELOPMENT (ASD)

Esta metodología parte de la idea de que las necesidades del cliente son siempre cambiantes durante el desarrollo del proyecto y posteriormente a su entrega.

Esta técnica fue desarrollada por Jim Highsmith y Sam Bayer a comienzos de los 90. La novedad de esta metodología es que en realidad no es una metodología de desarrollo de software, sino un método/técnica, a través del cual inculcar una cultura adaptativa a la empresa para adaptarse al cambio y no luchar contra él. En ella no hay un ciclo de vida estático (planear-diseñar-construir), si no que ofrece un ciclo de vida iterativo, donde cada ciclo puede ser modificado al tiempo que otro es ejecutado (especular colaborar-aprender).

Los objetivos de esta metodología son:

1) Concienciar a la organización de que debe esperar cambio e incertidumbre y no orden y estabilidad.
2) Desarrollar procesos iterativos de gestión del cambio.
3) Facilitar la colaboración y la interacción de las personas a nivel interpersonal, cultural y estructural.
4) Marcar una estrategia de desarrollo rápido de aplicaciones pero con rigor y disciplina.

Sus principales características son:

1) Iterativo.
2) Orientado a los componentes software más que a las tareas.
3) Tolerante a los cambios.
4) Guiado por los riesgos.
5) La revisión de los componentes sirve para aprender de los errores y volver a iniciar el ciclo de desarrollo.

El ciclo utilizado por ASD es conocido como: especular-colaborar-aprender, el cual está dedicado a un constante aprendizaje y colaboración entre desarrollador y cliente.

Ciclo de vida ASD

Especulación

1) Inicio, para determinar la misión del proyecto.
2) Fijación del marco temporal del proyecto.
3) Determinación de número de iteraciones y la duración de cada una.
4) Definición de objetivo de cada iteración.
5) Asignación de funcionalidad de cada iteración.

Una primera fase de iniciación para establecer los principales objetivos y metas del proyecto en su conjunto y comprender las limitaciones (zonas de riesgo) con las que operará el proyecto. En ASD se realizan estimaciones de tiempo sabiendo que pueden sufrir desviaciones. Sin embargo, estas son necesarias para la correcta atención de los trabajadores que se mueven dentro de plazos de forma que puedan priorizar sus tareas. Se decide el número de iteraciones para consumir el proyecto, prestando atención a las características que pueden ser utilizadas por el cliente al final de la iteración. Son por tanto necesarios, marcar objetivos prioritarios dentro de las mismas iteraciones. Estos pasos se puede volver a examinar varias veces antes de que el equipo y los clientes están satisfechos con el resultado.

Colaborar

Es la fase donde se centra la mayor parte del desarrollo manteniendo una componente cíclica. Un trabajo importante es la coordinación que asegure que lo aprendido por un equipo se transmite al resto y no tenga que volver a ser aprendido por los otros equipos.

Aprender

En cada iteración se revisa:

- Calidad del producto desde el punto de vista del cliente.
- Calidad del producto desde el punto de vista de los desarrolladores.
- Funcionalidad desarrollada.
- Estado del proyecto

Esta última etapa termina con una serie de ciclos de colaboración, su trabajo consiste en capturar lo que se ha aprendido, tanto positivo como negativo. Es un elemento crítico para la eficacia de los equipos.

Se identifican cuatro tipos de aprendizaje en esta etapa:

- Calidad del producto desde un punto de vista del cliente: es la única medida legítima de éxito, pero además, dentro de las metodologías ágiles, los clientes tienen un valor importante.
- Calidad del producto desde un punto de vista de los desarrolladores: se trata de la evaluación de la calidad de los productos desde un punto de vista técnico. Ejemplos de esto incluyen la adhesión a las normas y objetivos conforme a la arquitectura.
- La gestión del rendimiento: este es un proceso de evaluación para ver lo que se ha aprendido mediante el empleo de los procesos utilizados por el equipo.
- Situación del proyecto: como paso previo a la planificación de la siguiente iteración del proyecto, es el punto de partida para la construcción de la siguiente serie de características.

Ventajas

- Se utiliza para poder aprender de los errores e iniciar nuevamente el ciclo de desarrollo.
- Utiliza información disponible acerca de todos los cambios para poder mejorar el comportamiento del software.
- Promulga la colaboración y la interacción de personas.
- Apunta hacia el Rapid Application Development (RAD), el cual enfatiza velocidad de desarrollo para crear un producto de alta calidad, bajo mantenimiento involucrando al usuario lo más posible.

Desventajas

- Los errores y cambios que no son detectados con anterioridad afectan la calidad del producto y su costo total.
- Ya que esta es una metodología ágil, no permite realizar procesos que son requeridos en las metodologías tradicionales.

5.3.7.LEAN DEVELOPMENT (LD) Y LEAN SOFTWARE DEVELOPMENT (LSD)

El desarrollo Lean es una adaptación a los entornos de desarrollo de software, del método de producción que desarrolló Toyota, para equipos pequeños de programadores. Se fundamenta principalmente en constituir un equipo fuerte y altamente preparado capaz de llevar a cabo cualquier tarea en poco tiempo, legando todo a la eficacia y la cohesión de los componentes del equipo y obviando los procesos y la burocracia que conlleva normalmente el tener un sistema de producción preestablecido.

La filosofía Lean consiste en tener un equipo muy preparado, altamente motivado y muy unido. Los activos más importantes a tener en cuenta cuando se está desarrollando un proyecto bajo Lean Development no son el tiempo o el dinero que se está invirtiendo sino el grado de compromiso y, sobre todo, cuánto está aprendiendo el equipo. Se considera que cuanto más hayan aprendido los miembros del equipo y más unidos se sientan, la cantidad de tiempo y dinero necesario para llevar a cabo los desarrollos será cada vez menor.

Comprendiendo esta práctica desde un punto de vista empresarial, se deberá hacer una inversión fuerte al principio para sostener a un equipo poco fogueado y trabajar en mejorar su compromiso con la empresa y dotarles de experiencia, pero a medio plazo estos costes se reducirán y la productividad subirá, previendo a la larga un escenario en el que los costes de producción se mantienen estables y la productividad del equipo es extraordinaria.

Las ventajas que se derivan de tener un equipo muy sólido son entonces evidentes y muy útiles en el mundo del desarrollo de software. Se dispone de programadores que son capaces de analizar la situación, tomar decisiones correctas y llevarlas a cabo a una velocidad fuera de lo normal. Se puede comenzar a desarrollar teniendo una vaga idea de cuál es el objetivo final del proyecto e ir variando el

rumbo a la vez que se programa, posponiendo las decisiones importantes lo más posible a medida que se va disponiendo de datos estadísticos sobre la aceptación del producto que se está desarrollando.

Es un método de desarrollo ágil muy eficaz para proyectos a medio plazo: se concibe una idea, se programa y se lanza un prototipo que se ofrecen a un conjunto de personas para que lo prueben y poder analizar su comportamiento. Una vez analizado, se toman decisiones, se varía el rumbo, se desarrolla rápidamente y se repite el análisis con un nuevo prototipo. Después de una serie de iteraciones, se obtendrá un producto muy definido y que ha sido diseñado específicamente para cumplir el objetivo con el que fue concebido en función de las opiniones de los propios clientes finales.

Principios Lean

1. Eliminar los desperdicios
2. Ampliar el aprendizaje
3. Decidir lo más tarde posible
4. Reaccionar tan rápido como sea posible
5. Potenciar el equipo
6. Crear la integridad
7. Véase todo el conjunto

Eliminar los desperdicios

Todo lo que no añade valor al cliente se considera un desperdicio: código y funcionalidades innecesarias. Requisitos poco claros, burocracia, comunicación interna lenta.

Con el fin de poder eliminar los desperdicios deberíamos ser capaces de reconocerlos y encontrarlos. Si alguna actividad podría ser excluida o el mismo resultado podría ser logrado sin ella, esta actividad es considerada un desperdicio. Los procesos y funcionalidades extra que no son usados por el cliente son desperdicios. Los gastos de gestión que no producen valor real son desperdicios. Se utiliza una técnica llamada Value Stream Mapping (o mapa de flujo de valor) para distinguir y reconocer los desperdicios. El segundo paso consiste en señalar las fuentes de los desperdicios y eliminarlos. Lo mismo debe hacerse iterativamente hasta que incluso los procesos y procedimientos que parecían esenciales sean eliminados.

Ampliar el aprendizaje

El desarrollo de software es un proceso de aprendizaje continuo, a ello se le suman los retos de los equipos de desarrollo y el tamaño del producto final. El mejor enfoque para encarar una mejora en el ambiente de desarrollo de software es ampliar el aprendizaje. La acumulación de defectos debe evitarse ejecutando las pruebas tan pronto como el código está escrito en lugar de añadir más

documentación o planificación detallada. Las distintas ideas podrían ser probadas escribiendo código e integrándolo. El proceso de recopilación de requisitos de usuarios podría simplificarse mediante la presentación de las pantallas de los usuarios finales para que estos puedan hacer sus aportes. El proceso de aprendizaje es acelerado, con el uso de iteraciones cortas, cada una de ellas acompañada de refactorización y sus pruebas de integración.

Incrementando la retroalimentación mediante reuniones cortas con los clientes se ayuda a determinar la fase actual de desarrollo y se ajustan los esfuerzos para introducir mejoras en el futuro.

Durante las reuniones, tanto los clientes como el equipo de desarrollo, logran aprender sobre el alcance del problema y buscan posibles soluciones para un mejor desarrollo. Por lo tanto, los clientes comprenden mejor sus necesidades basándose en el resultado de los esfuerzos del desarrollo y los desarrolladores aprenden a satisfacer mejor estas necesidades.

Otra idea para ampliar el aprendizaje es a través de la integración del cliente en el ambiente de desarrollo para concentrar la comunicación en las soluciones futuras y no en las soluciones posibles, promoviendo así el nacimiento de la solución a través del diálogo con el cliente.

Decidir lo más tarde posible

El desarrollo de software está siempre asociado con cierto grado de incertidumbre, los mejores resultados se alcanzan con un enfoque basado en opciones por lo que se pueden retrasar las decisiones tanto como sea posible hasta que éstas se basen en hechos y no en suposiciones y pronósticos inciertos. Cuanto más complejo es un proyecto, más capacidad para el cambio debe incluirse en éste, así que debe permitirse el retraso de los compromisos importantes y cruciales. El enfoque iterativo promueve este principio: la capacidad de adaptarse a los cambios y corregir los errores, ya que un error podría ser muy costoso si se descubre después de la liberación del sistema.

Un enfoque de desarrollo de software ágil puede llevarles opciones rápidamente a los clientes, lo que implica, retrasar algunas decisiones cruciales hasta que los clientes hayan reconocido mejor sus necesidades. Esto también permite la adaptación tardía a los cambios y previene las costosas decisiones delimitadas por la tecnología. Esto no significa que no haya planificación involucrada en el proceso, por el contrario, las actividades de planificación deben centrarse en las diferentes opciones y se les adapta a la situación actual; así como, se deben clarificar las situaciones confusas estableciendo las pautas para una acción rápida.

Reaccionar tan rápido como sea posible

Cuanto antes se entrega el producto final sin defectos considerables más pronto se pueden recibir comentarios y se incorporan en la siguiente iteración. Cuanto más cortas sean las iteraciones, mejor es el aprendizaje y la comunicación dentro del equipo. Sin velocidad, las decisiones no pueden ser postergadas. La velocidad asegura el cumplimiento de las necesidades actuales del cliente y no lo que éste requería para ayer. Esto les da la oportunidad de demorarse pensando lo que realmente

necesitan, hasta que adquieran un mejor conocimiento. Los clientes valoran la entrega rápida de un producto de calidad.

La ideología de producción Just In Time podría aplicarse a programas de desarrollo, reconociendo sus necesidades específicas y el ambiente. Lo anterior se logra mediante la presentación de resultados, la necesidad de dejar que el equipo se organizarse y dividiendo las tareas para lograr el resultado necesario para una iteración específica.

Al principio, el cliente dispone los requisitos necesarios. Esto podría ser simplemente presentar los requisitos en pequeñas fichas o historias y los desarrolladores estimarán el tiempo necesario para la aplicación de cada tarjeta. Así, la organización del trabajo cambia en sistema autopropulsado: cada mañana durante una reunión inicial cada miembro del equipo evalúa lo que se ha hecho ayer, lo que hay que hacer hoy y mañana y pregunta por cualquier nueva entrada necesaria de parte de sus colegas o del cliente. Esto requiere la transparencia del proceso, que es también beneficioso para la comunicación del equipo.

Potenciar el equipo

Una creencia errónea es la de considerar a las personas como recursos. Las personas podrían ser los recursos desde el punto de vista de una hoja de datos estadísticos, pero en el desarrollo de software, así como cualquier organización de negocios, las personas necesitan algo más que la lista de tareas y la seguridad de que no será alterada durante la realización de las tareas. Las personas necesitan motivación y un propósito superior para el cual trabajar, un objetivo alcanzable dentro de la realidad con la garantía de que el equipo puede elegir sus propios compromisos. Los desarrolladores deberían tener acceso a los clientes; el jefe de equipo debe proporcionar apoyo y ayuda en situaciones difíciles, así como asegurarse de que el escepticismo no arruine el espíritu de equipo.

Crear la integridad

Una de las maneras más saludables hacia una arquitectura integrante es la refactorización.

Cuantas más funcionalidades se añaden a las del sistema, más se pierde del código base para futuras mejoras. Así como en la Programación extrema (XP), la refactorización es mantener la sencillez, la claridad, la cantidad mínima de funcionalidades en el código. Las repeticiones en el código son signo de un mal diseño de código y deben evitarse. El completo y automatizado proceso de construcción debe ir acompañada de una suite completa y automatizada de pruebas, tanto para desarrolladores y clientes que tengan la misma versión, sincronización y semántica que el sistema actual. Al final, la integridad debe ser verificada con una prueba global, garantizando que el sistema hace lo que el cliente espera que haga. Las pruebas automatizadas también son consideradas como parte del proceso de producción y, por tanto, si no agregan valor deben considerarse residuos. Las pruebas automatizadas no deberían ser un objetivo, sino, un medio para un fin; específicamente para la reducción de defectos.

Véase todo el conjunto

Los sistemas de software hoy en día no son simplemente la suma de sus partes, sino también el producto de sus interacciones. Los defectos en el software tienden a acumularse durante el proceso de desarrollo por medio de la descomposición de las grandes tareas en pequeñas tareas y de la normalización de las diferentes etapas de desarrollo. Las causas reales de los defectos deben ser encontradas y eliminadas. Cuanto más grande sea el sistema, más serán las organizaciones que participan en su desarrollo y más partes son las desarrolladas por diferentes equipos y mayor es la importancia de tener bien definidas las relaciones entre los diferentes proveedores con el fin de producir una buena interacción entre los componentes del sistema.

Como se puede ver, Lean va más allá de ser un conjunto de técnicas, hoy por hoy es una filosofía de gestión empresarial por sí misma, que promueve la comunicación y la motivación como base de su aprendizaje para llevar a cabo proyectos chicos-medianos de manera exitosa.

5.3.8.PROCESO UNIFICADO DE DESARROLLO SOFTWARE

Proceso Unificado de Desarrollo (RUP) es una metodología de desarrollo de software que está basado en componentes e interfaces bien definidas, y junto con el Lenguaje Unificado de Modelado (UML), constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos.

Es un proceso que puede especializarse para una gran variedad de sistemas de software, en diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyecto.

RUP no es un sistema con pasos firmemente establecidos, sino un conjunto de metodologías adaptables al contexto y necesidades de cada organización.

Es el resultado de varios años de desarrollo y uso práctico en el que se han unificado técnicas de desarrollo, a través del UML, y trabajo de muchas metodologías utilizadas por los clientes. La versión que se ha estandarizado vio la luz en 1998 y se conoció en sus inicios como Proceso Unificado de Rational 5.0, de ahí las siglas con las que se identifica a este proceso de desarrollo.

Principales Elementos

Como RUP es un proceso, en su modelación define como sus principales elementos:

- Trabajadores: define el comportamiento y responsabilidades (rol) de un individuo, grupo de individuos, sistema automatizado o máquina, que trabajan en conjunto como un equipo. Ellos realizan las actividades y son propietarios de elementos.
- Actividades: es una tarea que tiene un propósito claro, es realizada por un trabajador y manipula elementos.

- Artefactos: productos tangibles del proyecto que son producidos, modificados y usados por las actividades. Pueden ser modelos, elementos dentro del modelo, código fuente y ejecutables.
- Flujo de actividades: secuencia de actividades realizadas por trabajadores y que produce un resultado de valor observable.

Características Principales de RUP

- Unifica los mejores elementos de metodologías anteriores.
- Preparado para desarrollar grandes y complejos proyectos.
- Orientado a Objetos.
- Utiliza el UML como lenguaje de representación visual.

Principales ventajas:

- Coste del riesgo a un solo incremento.
- Reduce el riesgo de no sacar el producto en el calendario previsto.
- Acelera el ritmo de desarrollo.
- Se adapta mejor a las necesidades del cliente.

Características del ciclo de vida de RUP

Dirigido por casos de uso

Los casos de uso reflejan lo que los usuarios futuros necesitan y desean, lo cual se capta cuando se modela el negocio y se representa a través de los requerimientos. A partir de aquí los casos de uso guían el proceso de desarrollo ya que los modelos que se obtienen, como resultado de los diferentes flujos de trabajo, representan la realización de los casos de uso (cómo se llevan a cabo).

Centrado en la arquitectura

La arquitectura muestra la visión común del sistema completo en la que el equipo de proyecto y los usuarios deben estar de acuerdo, por lo que describe los elementos del modelo que son más importantes para su construcción, los cimientos del sistema que son necesarios como base para comprenderlo, desarrollarlo y producirlo económicamente. RUP se desarrolla mediante iteraciones, comenzando por los casos de uso relevantes desde el punto de vista de la arquitectura. El modelo de arquitectura se representa a través de vistas en las que se incluyen los diagramas de UML.

Iterativo e Incremental

Una iteración involucra actividades de todos los flujos de trabajo, aunque desarrolla fundamentalmente algunos más que otros.

Por ejemplo, una iteración de elaboración centra su atención en el análisis y diseño, aunque refina los requerimientos y obtiene un producto con un determinado nivel, pero que irá creciendo incrementalmente en cada iteración.

Es práctico dividir el trabajo en partes más pequeñas o miniproyectos. Cada miniproyecto es una iteración que resulta en un incremento. Las iteraciones hacen referencia a pasos en los flujos de trabajo, y los incrementos, al crecimiento del producto. Cada iteración se realiza de forma planificada es por eso que se dice que son miniproyectos.

Flujo y fases de trabajo de RUP

Cada fase representa un ciclo de desarrollo en la vida de un producto de software.

La fase de concepción o inicio tiene por finalidad definir la visión, los objetivos y el alcance del proyecto, tanto desde el punto de vista funcional como del técnico, obteniéndose como uno de los principales resultados una lista de los casos de uso y una lista de los factores de riesgo del proyecto.

El principal esfuerzo está radicado en el Modelamiento del Negocio y el Análisis de Requerimientos. Es la única fase que no necesariamente culmina con una versión ejecutable.

La fase de elaboración tiene como principal finalidad completar el análisis de los casos de uso y definir la arquitectura del sistema, además se obtiene una aplicación ejecutable que responde a los casos de uso que la comprometen. A pesar de que se desarrolla a profundidad una parte del sistema, las decisiones sobre la arquitectura se hacen sobre la base de la comprensión del sistema completo y los requerimientos (funcionales y no funcionales) identificados de acuerdo al alcance definido. La fase de construcción está compuesta por un ciclo de varias iteraciones, en las cuales se van incorporando sucesivamente los casos de uso, de acuerdo a los factores de riesgo del proyecto.

Este enfoque permite por ejemplo contar en forma temprana con versiones el sistema que satisfacen los principales casos de uso. Los cambios en los requerimientos no se incorporan hasta el inicio de la próxima iteración.

La fase de transición se inicia con una versión de prueba (beta) del sistema y culmina con el sistema en fase de producción.

En RUP se han agrupado las actividades en grupos lógicos definiéndose 9 flujos de trabajo principales, los 6 primeros son conocidos como flujos de ingeniería y los tres últimos como flujos de apoyo.

- Modelo del Negocio: describe los procesos de negocio, identificando quiénes participan y las actividades que requieren automatización.
- Requerimiento: define qué es lo que el sistema debe hacer, para lo cual se identifican las funcionalidades requeridas y las restricciones que se imponen.

- Análisis y Diseño: describe cómo el sistema será realizado a partir de la funcionalidad prevista y las restricciones impuestas (requerimientos), por lo que indica con precisión lo que se debe programar.
- Implementación: define cómo se organizan las clases y objetos en componentes, cuáles nodos se utilizarán y la ubicación en ellos de los componentes y la estructura de capas de la aplicación.
- Prueba (Testing): busca los defectos a lo largo del ciclo de vida.
- Instalación: el sistema se pone en marcha en producción (se libera al cliente y usuario final) y se realizan actividades (empaque, instalación, asistencia a usuarios, etc.) para entregar el software a los usuarios finales.
- Administración del proyecto: involucra actividades con las que se busca producir un producto que satisfaga las necesidades de los clientes.
- Administración de configuración y cambios: describe cómo controlar los elementos producidos por todos los integrantes del equipo de proyecto en cuanto a utilización/actualización concurrente de elementos, control de versiones, etc.
- Ambiente: contiene actividades que describen los procesos y herramientas que soportarán el equipo de trabajo del proyecto; así como el procedimiento para implementar el proceso en una organización.

Diferencias de RUP con las demás metodologías

Algunos aspectos que diferencian a RUP de las demás metodologías y lo que lo hace único es que en RUP, los casos de uso no son sólo una herramienta para especificar los requisitos del sistema, sino que también guían su diseño, implementación y prueba. Los casos de uso constituyen un elemento integrador y una guía del trabajo.

Además de utilizar los casos de uso para guiar el proceso, se presta especial atención al establecimiento temprano de una buena arquitectura que no se vea fuertemente impactada ante cambios posteriores durante la construcción y el mantenimiento. También este propone que cada fase se desarrolle en iteraciones.

6. ESTUDIO Y ANALISIS DE CASOS REALES

6.1. CASOS DE ÉXITO

1. Tantus Technologies, Inc.

¿Cómo hace una organización para implementar algo que nunca ha desarrollado antes, para un cliente que nunca lo ha utilizado anteriormente, para miembros de un equipo sin experiencia alguna y para usuarios finales que nunca la habían sentido nombrar?. Esa es la pregunta que se planteó Scott Granieri, proyect manager de la empresa Tantus Technologies, cuando se dispusieron a explorar como hacerle frente a un determinado componente del plan estratégico de un cliente federal. El siguiente documento detalla la experiencia interna de cómo se implementó Agile, sin contratar personal capacitado en esta nueva metodología y como se capacito al personal, clientes y usuarios finales debido a las limitaciones presupuestarias.

El cliente, en 2011, decidió incorporar Agile como metodología de trabajo debido a las tendencias de la industria y del sector, tanto público como privado, en ese momento.

Las empresas de productos, así como los proveedores de servicios estaban adoptando esta nueva metodología de trabajo de manera exponencial, incluso las empresas de desarrollo de software más tradicionales, habían reconocido este movimiento metodológico que sacudía la industria del software.

Para Tantus Technologies, el desafío de implementar un enfoque ágil sobre el programa actual, era complejo. Se enfrentaban a numerosos obstáculos al intentar ayudar al cliente a alcanzar el objetivo estratégico de incorporar un desarrollo ágil de software.

La implementación de prácticas agiles y la creación de una cultura ágil fue un gran desafío para la organización; sin embargo se identificaron las tres categorías específicas de los desafíos que tuvieron que superar para lograr el éxito: organización, cultural y personal.

Retos organizacionales

El programa federal, es una organización altamente matricial, soportando más de 23 sistemas, con más de 80 miembros entre los equipos, realizando ambas operaciones y trabajos de mantenimiento a lo largo de mejoras proyectadas. El número de proyectos por año, oscilaba históricamente entre 50 y 70. Debido al gran número de proyectos y al conjunto de habilidades diversas necesarias para apoyar a todos los distintos sistemas, se identificaron de 5 a 7 miembros por equipo multifuncional. Este reto, era importante, porque uno de los factores claves para el éxito de la implementación de una metodologia ágil era tener todos los miembros de los equipos dedicados completamente a este nuevo trabajo. Equipos estables y dedicados ayudan a construir cohesión conjuntamente como a su vez también, aprenden a entender el producto, su metodologia y a sus compañeros de trabajo. Esta estabilidad conduce a realizar una estimación del esfuerzo mucho más precisa. La estimación de la productividad del equipo, medida en puntos de la historia, se volvía más confiables con cada iteración.

Retos culturales

Se puso en marcha la implementación de una metodologia ágil sobre una organización con una larga historia de desarrollo en casada. El programa había madurado los procesos a lo largo de los años, tomando un gran esfuerzo para lograr CMMI y la impresa en el equipo de una cultura de éxito a través de la conformidad con los procesos. Muchas de las lecciones aprendidas de las victorias del pasando, deberían ser desafiadas y/o reemplazadas por un nuevo entorno de trabajo ágil que incluiría lo siguiente:

- La adopción de técnicas de programación y el costo de estimación ágil

En un entorno ágil, el triángulo tradicional de prioridades está a la inversa. Lo que una vez fue fijo (alcance) se convierte en una variable, y las que eran tradicionalmente variables (recursos y tiempo) se convierten en fijos. Cuando se empezó a desarrollar la capacidad ágil, no se sabía cómo el equipo líder de programación iba a reaccionar a este nuevo enfoque de estimación, especialmente después de poner tanto esfuerzo en crear sistemas para identificar y gestionar los horarios y los sobre costos.

- Gestión vs liderazgo

Gran parte de la cultura y la estructura organizativa del sistema federal fue construido sobre cimientos de la visión tradicional de cascada para la gestión de proyectos. Al igual que al reto sobre la gestión de los horarios y al costo de estimación, no se sabía cómo esta cultura altamente estructurada y tradicional se adaptaría a la naturaleza más dinámica de requerimientos, diseños y productos de trabajo final producidos en un ambiente ágil. Equipos autodirigidos y un entorno de constante cambio eran conceptos extraños y potencialmente riesgosos para la jerarquía de la organización.

Desafíos del personal

Tanto el personal de trabajo de la compañía como el personal del cliente, mostraban una escasez de conocimientos en un entorno ágil, lo que dificultaba el éxito a tiempo del proyecto. Debido a esto, en su momento se pensó en contratar personal capacitado para poder enfrentar estos problemas de experiencia, pero debido a limitaciones presupuestarias, se descartaron dichas opciones.

***CMMI**: Capability Maturity Model Integration es un modelo para la mejora y evaluación de procesos para el desarrollo, mantenimiento y operación de sistemas de software.*

La solución

En apoyo para el objetivo estratégico de una implementación ágil, se desarrolló y ejecuto un plan de acción ágil. Los componentes claves fueron los siguientes:

- Evaluación y selección de un entorno de trabajo ágil. Luego de este proceso, Tantus Technologies se decidió por implementar Scrum.
- Se envió parte del personal a capacitar para formar Scrum Masters certificados.
- Se incorporó un recurso externo con conocimiento en Scrum para realizar capacitaciones y demostrar al cliente la metodologia ágil y sus beneficios
- El desarrollo de una visión general de formación para los nuevos clientes para educarlos acerca de su papel como dueños del producto (product owner)
- Creación de un ciclo de vida de desarrollo de software ágil incluyendo herramientas, plantillas y técnicas para el desarrollo y la entrega.
- La identificación del conjunto de Sprints a ejecutar durante el ciclo de vida del proyecto

Luego de un largo y duro año de aprendizaje, el proyecto se pudo llevar a cabo con éxito.

Lecciones aprendidas

Los principales resultados y factores claves de éxito derivados de la experiencia que tuvo Tantus Technologies son los siguientes:

- Un Product Owner autorizado y disponible es clave: cuando un equipo de desarrollo puede contar y trabajar con un cliente interno totalmente dedicado, centrado y confiado en su equipo es parte de la clave del éxito.
- Un personal dedicado en el tiempo es de suma importancia: cuando las personas de un proyecto entran y salen de los equipos, genera un efecto negativo en el mismo y puede afectar drásticamente al proyecto y las demás áreas. Es por esto que es altamente recomendable que los equipos de trabajo se mantengan estáticos en el tiempo para poder aprovechar al máximo las experiencias logradas.
- Respetar los tiempos: es normal de cualquier proyecto, que se quieran agregar más funcionalidades en iguales tiempos o incluso más cortos. En cambio, mas Sprints, significan entregas más pequeñas, reduciendo el riesgo y permitiendo al equipo a aprender más rápidamente y lograr los objetivos con un porcentaje muy bajo de fracaso. Para esto es necesario resistirse a la tentación de sobrecargar los sprints con más user stories.
- Desarrollar constantemente el Product Backlog: en Scrum se recomienda que el 5 o 10 por ciento de cada sprint se dedicase a la refinación del product backlog ya que es un componente sumamente importante para la recolección de requisitos e información.
- Herramientas automatizadas para la gestión del proyecto: Tantus Technologies se decidió por adoptar la herramienta de gestión JIRA. Esta permite la gestión de las user stories, la carga de errores, estimaciones, priorización, asignación de tareas y seguimiento del trabajo.

Es cierto que el punto principal de aplicar una metodologia ágil consiste en eliminar las herramientas burocráticas de trabajo e implementar la comunicación entre las personas del equipo pero, sin embargo, siempre es necesario una herramienta de gestión de este tipo.

2. Spotify

Existen muchísimas compañías que desarrollan software de una forma ágil y exitosa. Google, por ejemplo, cuenta con 15.000 desarrolladores trabajando en una rama del código. Lanzan cambios varias veces al día y realizan 75 millones de tests automáticos diariamente.

Una empresa que ha sabido adaptarse perfectamente a las metodologías ágiles es Spotify, haciendo especial hincapié en la figura del Scrum Master. Muchas veces contratan un Agile Coach externo con una gran experiencia en el campo para liderar los proyectos. Vemos aquí la importancia de contar con roles especializados que conozcan las metodologías ágiles para llevar un proyecto de este tipo al éxito. Ya no solo el Scrum Master, sino también otros roles como el Product Owner, responsable de entender al cliente y al usuario para saber trasladar en tiempo y forma la información adecuada al equipo de desarrollo.

Spotify es consciente de la metodología de trabajo de su competencia (Google o Apple por ejemplo), por lo que decidieron acercarse al Scrum de forma muy sistemática. Compitiendo contra semejantes corporaciones, sabían que en cualquier momento podrían ser derrotados a menos que fuesen más rápidos, más baratos y mejores.

En Spotify los equipos se organizan por escuadrones (squads), pequeños equipos de Scrum con la habilidad de implementar el software desarrollado al final de cada sprint, sin romper ningún otro equipo. Una característica curiosa del funcionamiento de Spotify es que cada uno de estos pequeños grupos tiene una parte del producto que es totalmente suyo. Después crean tribus (tribes) agregando distintos escuadrones.

Aun así, Spotify necesita implementar, cambiar y actualizar su código constantemente sin romper nada más. Para ello es necesaria una buena coordinación central de la compañía.

Para ser rápido también es necesario deshacerse de todas aquellas partes del proceso que entorpezcan el avance. En Spotify, por ejemplo, contaban con un equipo de operaciones que se encargada de las implementaciones, pero el funcionamiento era demasiado lento. Por eso decidieron eliminar esta fase y hacer que los propios desarrolladores implementasen sus trabajos.

Cada vez son más las empresas que operan en el mundo digital y necesitan optimizar sus procesos al máximo para asegurarse la primera posición en el mercado. Es por ello que las metodologías ágiles como el Scrum están adquiriendo una importancia especial dentro de las organizaciones y los profesionales especializados en la gestión de este tipo de proyectos son cada vez más demandados.

6.2. CASOS DE FRACASO

1. Healthcare

Healthcare es un proyecto del gobierno americano diseñado para ofrecer toda la información y transparencia sobre el mercado de los seguros sanitarios, para que los consumidores puedan asegurarse de obtener el mejor valor. Jeff Sutherland, cocreador de Scrum, lo cita como ejemplo de mala gestión de un proyecto Scrum.

Las principales causas del fracaso en el desarrollo de Healthcare fueron la falta de coordinación entre el Front End y el Back End, la falta de liderazgo en un proyecto con más de 20 consultoras implicadas y no haber lanzado el proyecto fase a fase sin testeo ni aprendizaje de por medio, haciendo que fuese imposible detectar las fases que sí funcionaban y las que no.

Según Jeff Sutherlands, el fracaso de este proyecto no debería sorprendernos tanto ya que se trata de un proyecto de desarrollo en cascada, y afirma que el 86% de este tipo de proyectos suelen acabar en fracaso.

Tras trabajar en el desarrollo durante varios años, sólo se tuvieron seis días para testear la aplicación debido a las presiones para el inminente lanzamiento. Es de entender que con tan corto periodo de testing el proyecto no saliese bien lo que provoco que, aunque el equipo del Back End realmente trabajó como un proyecto ágil y en unos meses ya habían terminado su trabajo, el error estuvo en no respetar el segundo principio del manifiesto Agile de aceptación del cambio: “Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.” Por más que se haya logrado el desarrollo, no alcanzó para su funcionamiento.

7. CONCLUSION

Hoy en día la comunicación, la tecnología y los sistemas de información avanzan y evolucionan a una velocidad exponencial generando consigo que la gestión de proyectos informáticos esté a la altura de la velocidad de los cambios ocasionados por esta evolución.

El mundo del desarrollo del software ha cambiado drásticamente desde la aparición de internet y la gran cantidad de herramientas que nos permiten crear ecosistemas de trabajo mucho más colaborativos en los que el flujo de información es mucho más rápido que con las estructuras tradicionales.

En esta nueva generación, las metodologías tradicionales de desarrollo de software fueron quedado obsoletas en determinados sectores, en los que la propia demanda de los usuarios es más rápida que la capacidad de producción de las empresas ancladas a las viejas metodologías de gestión de proyectos de sistemas informáticos.

Este gran impacto en las tecnologías, ha generado la necesidad de encontrar y crear nuevas metodologías de trabajo y gestión, que aseguren la entrega en tiempo y forma del producto. Esta necesidad de calidad, eficiencia, flexibilidad y rapidez en la entrega de un producto informático se volvió prioridad y en conjunto con su necesidad se crearon las nombradas Metodologías Agiles.

El mundo en general y la vida de las personas, día a día se vuelve más ágil en todos sus aspectos, siendo prácticamente inevitable la evolución en los sistemas de información para poder atacar ésta demanda.

Los métodos ágiles y los tradicionales no son competidores directos. Cada uno de ellos tiene su propio segmento de aplicación o terreno en base a las necesidades del proyecto y de bien saber distinguir e identificar cual es la más adecuada en base a las características de nuestro proyecto, necesidades y recursos.

Algunos aspectos del desarrollo de software se beneficiarán del enfoque agilista mientras otros obtendrán beneficios de un enfoque tradicional-predictivo menos ágil y más estructurado.

Ambos metodologías, pueden fracasar si son mal implementadas, gestionadas y administradas.

No podemos decir que exista una metodología mejor que otra sino que dependerá de la naturaleza de la empresa y la forma de organización de sus procesos internos y de la capacidad de los líderes del proyecto de poder identificar la metodología que más se adecua e implementarla de manera eficiente.

Sin embargo, la tendencia natural actual indica que las metodologías agiles están ganando terreno muy rápidamente lo que en algunos años podrían generar la extinción definitiva de las metodologías tradicionales.

8. REFERENCIAS BIBLIOGRAFICAS

- Amador Duran Toro, Beatriz Bernrdez Jiménez, **"Metodología para el análisis de de requisitos de sistemas de software versión 2.2",** Universidad de Sevilla, departamento de Lenguajes y Sistemas informáticos, escuela Técnica superior de Ingeniería Informática, Diciembre de 2001.
- Roger S.Pressman. **"Ingeniería del Software: un enfoque práctico",** de segunda edición, Editorial McGraw Hill. 1990.
- Manuel Díaz Rodríguez, Antonio Moña Gómez, **"Ingeniería de Software Especificación",** Departamento de Lenguajes de Ciencias de la computación, Universidad de Málaga.
- Bruce I. Blum, "***Software Engineering: A Holistic View***".
- Dorothy Graham, Erik Van Veenendaal, Isabel Evans y Rex Black, *"**Foundations of Software Testing - ISTQB® Certification**".* 2007.
- Duvall, Paul M., "***Continuous Integration. Improving Software Quality and Reducing Risk***". 2007.
- Hans Van Vliet, "***Software Engineering. Principles and Practice***". Tercera edición. 2002.
- Ian Sommerville, "***Software Engineering***". Sexta Edición. 2001.
- Ivar Jacobson, Grady Booch y James Rumbaugh, "***The Unified Software Development Process***". 1999.
- Kent Beck, "***Test-Driven Development by Example***".
- Kent Beck, Martin Fowler, "***Planning Extreme Programming***". 2000.
- Ken Schwaber, Mike Beedle, "***Agile Software Development with SCRUM***". 2008.
- Lawrence-Pfleeger y Shari, *"**Software Engineering: Theory and Practice**".* 1998.
- Mitchel H. Levine, "***Analyzing the Deliverables Produced in the Software Development Life Cycle***". 2000.
- Pierre Bourque y Robert Dupuis, "***Guide to the Software Engineering Body of Knowledge***". 2004.
- Robert C. Martin, "***Agile Software Development, Principles, Patterns, and Practices***".
- Roger S. Pressman, "***Software Engineering. A practitioner's Approach***". Quinta Edición. 2001.
- Ron Burback, "***Software Engineering Methodology***". 1998.
- Tong Ka lok, Kent, "***Essential Skills for Agile Development***".
- Beck, K. "**Extreme Programming Explained: Embrace Change**". Boston: Addison-Wesley. 2000.
- Beck, K. Martin F. "**Planning Extreme Programming**". Boston: Addison-Wesley. 2001.
- Canós, J. H.; Letelier, P.; Penadés, M. C. "**Metodologías ágiles en el desarrollo del software**". Valencia: Universidad de Valencia. 2004.
- Cockburn, A. "**Agile Software Development**". Boston: Addison-Wesley. 2001.
- Cronin, G. "**Extreme Solo: A Case Study in Single Developer extreme Programming**". University of Auckland. 2003.
- Highsmith, J. "**Agile Software Development Ecosystems**". Boston: Addison-Wesley. 2002.
- Reynoso, C. "**Métodos heterodoxos en desarrollo del software**". Buenos Aires: Universidad de Buenos Aires. 2004.

- Salo, J. H.; Abrahamson, P.; Ronkainen, J.; Warsta, J. "**Agile Software Development**". 2002.
- Schwaber, K.; Beedle, M. "**Agile Software Development with Scrum**". Nueva Jersey: Prentice Hall. 2002.
- Wake, W. C. "**Extreme Programming Explored**". Boston: Addison-Wesley. 2002.
- Boehm, Barry W., "**A Spiral Model of Software Development and Enhancement**", IEEE Computer, Vol.21, No 15, pp.61-72. Mayo 1988.
- Martin, J., "**Rapid Application Development**", Macmillan Inc., New York. 1991.
- Beck, Kent, "**Extreme Programming Explained**", Addison-Wesley the XP Series, 2000.
- Alistair Cockburn. "**Balancing Lightness with Sufficiency**". Septiembre 2000.
- Pérez S. Jesús, "**Metodologías Ágiles: La ventaja competitiva de estar preparado para tomar decisiones lo más tarde posible y cambiarlas en cualquier momento**". Artículo de Agile Spain. Grupo ISSI, Metodologías Ágiles en el Desarrollo de Software, Artículo de Grupo ISSI Noviembre 2003.
- Acebal F.César, Cueva L. Juan, "**EXtreme Programming (XP): un nuevo método de desarrollo de software, Articulo de Novática**".
- Letelier P., Penadés C., "**Metodologías ágiles para el desarrollo de Software: eXtreme Programming (XP)**", Universidad Politécnica de Valencia.
- Shenone M. Hernán, "**Diseño de una metodología Ágil de Desarrollo de Software**", Tesis de Grado en Ingeniería en Informática, Universidad de Buenos Aires.
- Página de Microsoft: MSDN en Español, "**Métodos Heterodoxos en Desarrollo de Software**", Artículo de Carlos Reynoso – Universidad de Buenos Aires, Abril del 2004.

Sitios web

- http://www.agile-spain.com
- http://www.agilealliance.org
- http://www.ieee.org/portal/site
- http://www.agilemanifesto.org
- Sitio web de la Organización Internacional para la Estandarización www.iso.org
- http://www.swebok.org
- http://www.rena.edu.ve/cuartaEtapa/Informatica/Tema11.html
- http://www.inf-cr.uclm.es/www/mpolo/asig/0708/phd/apuntesDoctorado.pdf
- http://audiemangt.blogspot.mx/2010/04/metodologia-clasica-en-cascada.html
- http://www.agiles.org/agiles-parana
- http://www.proyectosagiles.org/
- https://www.softeng.es/es-es/empresa/metodologias-de-trabajo/metodologia-scrum.html
- http://leansoftwareengineering.com/
- http://www.projectperfect.com.au/downloads/Info/info_lean_development.pdf
- http://www.poppendieck.com/
- http://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software
- http://www.agileshift.cl/Tutorial/DesarrolloAgilParte2.pdf
- http://www.willydev.net/descargas/prev/TodoAgil.pdf
- http://ingenieriadesoftware.mex.tl/61162_FDD.html
- https://agilidaddeldesarrollo.wordpress.com/2012/12/02/adaptive-software-development-asd/
- http://www.microsoft.com/spanish/msdn/arquitectura/roadmap_arq/heterodox.asp#12
- http://www.enterprisexp.org
- http://www.dsdm.org
- https://www.scrumalliance.org/community/articles/2013/november/success-story-sometimes-it-just-may-take-a-waterfa
- http://comunidad.iebschool.com/iebs/agile-scrum/exitos-y-fracasos-en-proyectos-scrum-spotify-vs-healthcare/

9. APENDICE

Figura Nº 1 Extraída de:
[http://www.itlalaguna.edu.mx/academico/carreras/sistemas/ingsofware1/Unidad1.pdf]

Figura Nº 2 Extraída de:
[http://www.itlalaguna.edu.mx/academico/carreras/sistemas/ingsofware1/Unidad1.pdf]

Figura Nº 3 Extraída de:
[https://www.academia.edu/13247296/MONOGRAFIA_SOBRE_LA_METODOLOGIA_DE_DESARROLLO_DE_SOFTWARE_RUP]

Figura Nº 4 Extraída de: [http://cflores334.blogspot.es/1192848180/]

Figura Nº 5 Extraída de: [http://andres-modelosdedesarrollo.blogspot.com.ar/]

Figura Nº 6 Extraída de: [http://xherrera334.blogspot.es/]

Figura Nº 7 Extraída de: [http://www.testingexcellence.com/rapid-application-development-rad/]

Figura Nº 8 Extraída de:
[https://www.exabyteinformatica.com/uoc/Informatica/Tecnicas_avanzadas_de_ingenieria_de_software/Tecnicas_avanzadas_de_ingenieria_de_software_(Modulo_3).pdf]

Figura Nº 9 Extraída de:
[https://www.exabyteinformatica.com/uoc/Informatica/Tecnicas_avanzadas_de_ingenieria_de_software/Tecnicas_avanzadas_de_ingenieria_de_software_(Modulo_3).pdf]

Figura Nº 10 Extraída de:
[https://www.exabyteinformatica.com/uoc/Informatica/Tecnicas_avanzadas_de_ingenieria_de_software/Tecnicas_avanzadas_de_ingenieria_de_software_(Modulo_3).pdf]

Figura Nº 11 Extraída de:
[https://www.exabyteinformatica.com/uoc/Informatica/Tecnicas_avanzadas_de_ingenieria_de_software/Tecnicas_avanzadas_de_ingenieria_de_software_(Modulo_3).pdf]

Figura Nº 12 Extraída de:
[https://www.exabyteinformatica.com/uoc/Informatica/Tecnicas_avanzadas_de_ingenieria_de_software/Tecnicas_avanzadas_de_ingenieria_de_software_(Modulo_3).pdf]

Figura Nº 13 Extraída de: [https://www.exabyteinformatica.com/uoc/Informatica/Tecnicas_avanzadas_de_ingenieria_de_software/Tecnicas_avanzadas_de_ingenieria_de_software_(Modulo_3).pdf]

Figura Nº 14 Extraída de: [http://bytelchus.com/tableros-kanban/]

Figura Nº 15 Extraída de: [http://paradigmas14.blogspot.com.ar/p/metodologias-agiles.html]

Figura Nº 16 Extraída de: [http://www.michalwolski.pl/2014/09/kanban-i-wykrywanie-waskich-gardel/].