GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, TURISMO Y COMERCIO
SECRETARÍA DE ESTADO DE TELECOMUNICACIONES Y PARA LA SOCIEDAD DE LA INFORMACIÓN

INTECO
Instituto Nacional de Tecnologías de la Comunicación

INGENIERÍA DEL SOFTWARE: METODOLOGÍAS Y CICLOS DE VIDA

Laboratorio Nacional de Calidad del Software

Marzo 2009

NOTA DE EDICIÓN

Esta guía ha sido desarrollada por el Laboratorio Nacional de Calidad del Software de INTECO. Esta primera versión ha sido editada en Marzo de 2009.

AVISO LEGAL

- CMMI® es una marca registrada en la Oficina de Marcas y Patentes de EEUU por la Universidad Carnegie Mellon
- Las distintas normas ISO mencionadas han sido desarrolladas por la International Organization for Standardization.

Todas las demás marcas registradas que se mencionan, usan o citan en la presente guía son propiedad de los respectivos titulares.

INTECO cita estas marcas porque se consideran referentes en los temas que se tratan, buscando únicamente fines puramente divulgativos. En ningún momento INTECO busca con su mención el uso interesado de estas marcas ni manifestar cualquier participación y/o autoría de las mismas.

Nada de lo contenido en este documento debe ser entendido como concesión, por implicación o de otra forma, y cualquier licencia o derecho para las Marcas Registradas deben tener una autorización escrita de los terceros propietarios de la marca.

Por otro lado, INTECO renuncia expresamente a asumir cualquier responsabilidad relacionada con la publicación de las Marcas Registradas en este documento en cuanto al uso de ninguna en particular y se eximen de la responsabilidad de la utilización de dichas Marcas por terceros.

El carácter de todas las guías editadas por INTECO es únicamente formativo, buscando en todo momento facilitar a los lectores la comprensión, adaptación y divulgación de las disciplinas, metodologías, estándares y normas presentes en el ámbito de la calidad del software.

ÍNDICE

ÍNDICE DE FIGURAS

1. DESCRIPCIÓN DE LA GUÍA

Esta guía pretende ser una introducción a la ingeniería del software y a las distintas metodologías y ciclos de vida de desarrollo que existen, haciendo especial hincapié en el desarrollo ágil.

La primera parte de la guía se va a centrar en la contextualización de la ingeniería del software. Va a permitir al lector entender las características, componentes y tipos de software, así como los objetivos y componentes de la ingeniería del software.

En la segunda parte de la guía nos centraremos en la definición del concepto de ciclo de vida y en la explicación de distintos modelos de ciclo de vida existentes.

La tercera parte de la guía se centrará en la explicación del concepto de metodología, donde se hablará tanto de modelos tradicionales como de métodos ágiles y se desarrollarán algunas de las metodologías más utilizadas.

La última parte se centrará en la explicación del desarrollo ágil, donde se tratarán tanto modelos como prácticas de desarrollo ágil.

2. INTRODUCCIÓN A LA INGENIERÍA DEL SOFTWARE

Cuando un software se desarrolla con éxito, cuando satisface las necesidades de las personas que lo utilizan; cuando funciona de forma impecable durante mucho tiempo; cuando es fácil de modificar o incluso es más fácil de utilizar, puede cambiar todas las cosas y de hecho cambiar para mejor. Ahora bien, cuando un software falla, cuando los usuarios no quedan satisfechos, cuando es propenso a errores, cuando es difícil de cambiar e incluso más difícil de utilizar, pueden ocurrir y de hecho ocurren verdaderos desastres. Todos queremos desarrollar un software que haga bien las cosas, evitando que esas cosas malas aparezcan. Para tener éxito al diseñar y construir un software necesitaremos disciplina. Es decir, necesitaremos un enfoque de ingeniería.

2.1. SOFTWARE

En primer lugar se va a tratar un concepto tan importante como es el software. Es importante entender este concepto para poder pasar a definir a continuación lo que es la ingeniería del software.

Algunas definiciones de **software**:

- IEEE Std. 610 define el software como “programas, procedimientos y documentación y datos asociados, relacionados con la operación de un sistema informático”
- Según el Webster’s New Collegiate Dictionary (1975), “software es un conjunto de programas, procedimientos y documentación relacionada asociados con un sistema, especialmente un sistema informático”.

El software se puede definir como el conjunto de **tres componentes**:

- Programas (instrucciones): este componente proporciona la funcionalidad deseada y el rendimiento cuando se ejecute.
- Datos: este componente incluye los datos necesarios para manejar y probar los programas y las estructuras requeridas para mantener y manipular estos datos.
- Documentos: este componente describe la operación y uso del programa.

Figura 1. Componentes del software

2.1.1. Componentes del software

Es importante contar con una definición exhaustiva del software ya que de otra manera se podrían olvidar algunos componentes. Una percepción común es que el software sólo consiste en programas. Sin embargo, los programas no son los únicos componentes del software.

Programas

Los programas son conjuntos de instrucciones que proporcionan la funcionalidad deseada cuando son ejecutadas por el ordenador. Están escritos usando lenguajes específicos que los ordenadores pueden leer y ejecutar, tales como lenguaje ensamblador, Basic, FORTRAN, COBOL, C… Los programas también pueden ser generados usando generadores de programas.

Datos

Los programas proporcionan la funcionalidad requerida manipulando datos. Usan datos para ejercer el control apropiado en lo que hacen. El mantenimiento y las pruebas de los programas también necesitan datos. El diseño del programa asume la disponibilidad de las estructuras de datos tales como bases de datos y archivos que contienen datos.

Documentos

Además de los programas y los datos, los usuarios necesitan también una explicación de cómo usar el programa.

Documentos como manuales de usuario y de operación son necesarios para permitir a los usuarios operar con el sistema.

Los documentos también son requeridos por las personas encargadas de mantener el software para entender el interior del software y modificarlo, en el caso en que sea necesario.

2.1.2. Características del software

A lo largo de los años, se han evolucionado muchas formas de producir bienes de mejor calidad en el sector de las manufacturas. Este conocimiento puede extenderse a la construcción de productos software de mejor calidad si los profesionales del software entienden las características propias del software.

Para poder comprender lo que es el software (y consecuentemente la ingeniería del software), es importante examinar las características del software que lo diferencian de otras cosas que el hombre puede construir.

El software es esencialmente un conjunto de instrucciones (programas) que proporcionan la funcionalidad requerida, los datos relacionados y documentos. Por lo tanto, el software es un elemento lógico y se diferencia del hardware, un elemento físico, en sus características.

El software se **desarrolla**, no se fabrica en el sentido clásico. Aunque existen similitudes entre el desarrollo del software y la construcción del hardware, ambas actividades son fundamentalmente distintas.

Cada producto software es diferente porque se construye para cumplir los requisitos únicos de un cliente. Cada software necesita, por lo tanto, ser construido usando un enfoque de ingeniería.

Construir un producto software implica entender qué es necesario, diseñar el producto para que cumpla los requisitos, implementar el diseño usando un lenguaje de programación y

comprobar que el producto cumple con los requisitos. Todas estas actividades se llevan a cabo mediante la ejecución de un proyecto software y requiere un equipo trabajando de una forma coordinada.

El proceso usado para construir software es diferente de la fabricación del hardware, donde las máquinas se usan para producir partes y cada trabajador sólo necesita realizar la tarea asignada o usar una máquina.

En el software, el recurso principal son las personas. No es siempre posible acelerar la construcción de software añadiendo personas porque la construcción de software requiere un esfuerzo en equipo. El equipo tiene que trabajar de forma coordinada y compartir un objetivo de proyecto común. Se necesita comunicación efectiva dentro del equipo.

Un nuevo miembro del equipo no es inmediatamente productivo y necesita la iniciación adecuada al equipo y la formación para realizar el trabajo. Esto requiere una inversión de tiempo y esfuerzo por parte de los miembros del equipo existentes y les puede distraer de su propio trabajo.

Otra característica del software es que **no se estropea**. Los defectos no detectados harán que falle el programa durante las primeras etapas de su vida. Sin embargo, una vez que se corrigen (suponiendo que no se introducen nuevos errores) los fallos disminuyen.

El software no se estropea, pero se deteriora. Durante su vida, el software sufre cambios (mantenimiento). Conforme se hacen los cambios, es bastante probable que se introduzcan nuevos defectos, lo que hace que el software se vaya deteriorando debido a los cambios.

Otro aspecto del software es que, debido a que la industria del software es nueva, el software se diferencia del hardware en el aspecto de uso de componentes. Aunque la mayoría de la industria tiende a ensamblar componentes, la mayoría del software **se construye a medida**.

Los componentes reutilizables se han creado para que el ingeniero pueda concentrarse en elementos verdaderamente innovadores de un diseño. En el mundo del software es algo que sólo ha comenzado a lograrse en productos lanzados a gran escala.

El componente software debería diseñarse e implementarse para que pueda volver a ser reutilizado en muchos programas diferentes. Hoy en día, se ha extendido la visión de la reutilización para abarcar tanto algoritmos como estructuras de datos, permitiendo al ingeniero del software crear nuevas aplicaciones a partir de las partes reutilizables.

El hardware usa componentes estándar con funciones e interfaces bien definidas. El uso de estos componentes ayuda a evitar reinventar la rueda. La fase de diseño en el ciclo de vida de un producto hardware implica seleccionar los componentes disponibles más adecuados y decidir el enfoque para montarlos. Los componentes de hardware estándar son útiles porque conducen a:

- Reducir el coste y el tiempo de lanzamiento al mercado
- Buena calidad
- Ingeniería rápida

- Fácil mantenimiento
- Fácil mejora

El software se crea normalmente desde cero. Con frecuencia se construye de acuerdo a los requisitos específicos de un cliente y no se crea por la unión de componentes existentes.

Como la industria del hardware, la industria del software está intentando adoptar el mecanismo de reutilizar para hacer más fácil y más rápida la construcción. Las ventajas de la reutilización de software están siendo entendidas y apreciadas. Existen algunos elementos reutilizables a través de librerías de funciones y objetos reutilizables que combinan funciones y datos.

Mientras que la reutilización y el montaje basado en componentes se están incrementando, la mayoría del software continua siendo construido de forma personalizada, y los niveles de reutilización actuales están lejos de los que deberían ser. Además, la tarea de identificar componentes reutilizables potenciales es difícil porque cada producto software es único y distinto.

La industria del software tiene procesos bien definidos para la reutilización de componentes. Esto incluye procesos para la construcción de componentes, almacenamiento de los mismos en librerías de donde se pueden extraer para su reutilización y entonces incorporarlos.

A lo largo de los años, la industria del software espera crear componentes reutilizables específicos a dominios de aplicación particulares.

2.1.3. Tipos de software

El software puede dividirse en dos grandes **categorías**:

- Software de aplicaciones: se usan para proveer servicios a clientes y ejecutar negocios de forma más eficiente. El software de aplicaciones puede ser un sistema pequeño o uno grande integrado. Como ejemplos de este tipo de software están: un sistema de cuentas, un sistema de planificación de recursos…
- Software de sistemas: el software de sistemas se usa para operar y mantener un sistema informático. Permite a los usuarios usar los recursos del ordenador directamente y a través de otro software. Algunos ejemplos de este tipo de software son: sistemas operativos, compiladores y otras utilidades del sistema.

2.1.4. Aplicaciones del software

El software puede aplicarse en cualquier situación en la que se haya definido previamente un conjunto específico de pasos procedimentales (es decir, un algoritmo) (excepciones notables a esta regla son el software de los sistemas expertos y de redes neuronales). El contenido y determinismo de la información son factores importantes a considerar para determinar la naturaleza de una aplicación software. El contenido se refiere al significado y a la forma de la información de entrada y salida. Por ejemplo, muchas aplicaciones bancarias usan unos datos de entrada muy estructurados (una base de datos) y producen informes con determinados formatos. El software que controla una máquina automática (por ejemplo:

un control numérico) acepta elementos discretos con una estructura limitada y produce órdenes concretas para la máquina en rápida sucesión.

El determinismo de la información se refiere a la predictibilidad del orden y del tiempo de llegada de los datos. Un programa de análisis de ingeniería acepta datos que están en un orden predefinido, ejecuta algoritmos de análisis sin interrupción y produce los datos resultantes en un informe o formato gráfico. Un sistema operativo multiusuario, por otra parte, acepta entradas que tienen un contenido variado y que se producen en instantes arbitrarios, ejecuta algoritmos que pueden ser interrumpidos en condiciones externas y produce una salida que depende de una función del entorno y del tiempo. Las aplicaciones con estas características se dice que son indeterminadas.

Algunas veces es difícil establecer categorías genéricas para las aplicaciones del software que sean significativas. Conforme aumenta la complejidad del software, es más difícil establecer compartimentos nítidamente separados. Las siguientes áreas del software indican la amplitud de las aplicaciones potenciales:

- **Software de sistemas**: el software de sistemas es un conjunto de programas que han sido escritos para servir a otros programas. Algunos programas de sistemas (por ejemplo: compiladores, editores y utilidades de gestión de archivos) procesan estructuras de información complejas pero determinadas. Otras aplicaciones de sistemas (por ejemplo: ciertos componentes del sistema operativo, utilidades de manejo de periféricos, procesadores de telecomunicaciones) procesan datos en gran medida indeterminados. En cualquier caso, el área del software de sistemas se caracteriza por una fuerte interacción con el hardware de la computadora; una gran utilización por múltiples usuarios; una operación concurrente que requiere una planificación, una compartición de recursos y una sofisticada gestión de procesos; unas estructuras de datos complejas y múltiples interfaces externas.
- **Software de tiempo real**: el software que coordina/analiza/controla sucesos del mundo real conforme ocurren. Entre los elementos del software de tiempo real se incluyen: un componente de adquisición de datos que recolecta y da formato a la información recibida del entorno externo, un componente de análisis que transforma la información según lo requiera la aplicación, un componente de control/salida que responda al entorno externo y un componente de monitorización que coordina todos los demás componentes, de forma que pueda mantenerse el respuesta en tiempo real.
- **Software de gestión**: el proceso de la información comercial constituye la mayor de las áreas de aplicación del software. Los sistemas discretos (por ejemplo: nóminas, cuentas de haberes-débitos, inventarios, etc.) han evolucionado hacia el software de sistemas de información de gestión (SIG) que accede a una o más bases de datos que contienen información comercial. Las aplicaciones en esta área reestructuran los datos existentes para facilitar las operaciones comerciales o gestionar la toma de decisiones. Además de las tareas convencionales de procesamiento de datos, las aplicaciones de software de gestión también realizan cálculo interactivo (por ejemplo: el procesamiento de transacciones en puntos de venta).

- **Software de ingeniería y científico**: este tipo de software está caracterizado por los algoritmos de manejo de números. Las aplicaciones van desde la astronomía a la vulcanología, desde el análisis de la presión de los automotores a la dinámica orbital de las lanzaderas espaciales y desde la biología molecular a la fabricación automática. Sin embargo las nuevas aplicaciones del área de ingeniería/ciencia se han alejado de los algoritmos convencionales numéricos. El diseño asistido por computadora (CAD), la simulación de sistemas y otras aplicaciones interactivas, han comenzado a coger características del software de tiempo real e incluso de software de sistemas.
- **Software empotrado**: los productos inteligentes se han convertido en algo común en casi todos los mercados de consumo e industriales. El software empotrado reside en memoria de sólo lectura y se utiliza para controlar productos y sistemas de los mercados industriales y de consumo. El software empotrado puede ejecutar funciones muy limitadas y curiosas (por ejemplo: el control de las teclas de un horno microondas) o suministrar una función significativa y con capacidad de control (por ejemplo: funciones digitales en un automóvil, tales como control de la gasolina, indicadores en el salpicadero, sistemas de frenado, etc.)
- **Software de computadoras personales**: el mercado del software de computadoras personales ha germinado en las pasadas décadas. El procesamiento de textos, las hojas de cálculo, los gráficos por computadora, multimedia, entretenimiento, gestión de bases de datos, aplicaciones financieras, de negocios y personales y redes o acceso a bases de datos externas son algunas de los cientos de aplicaciones.
- **Software basado en web**: las páginas web buscadas por un explorador son software que incorpora instrucciones ejecutables y datos.
- **Software de inteligencia artificial**: el software de inteligencia artificial hace uso de algoritmos no numéricos para resolver problemas complejos para los que no son adecuados el cálculo o el análisis directo. Los sistemas expertos, también llamados sistemas basados en el conocimiento, reconocimiento de patrones (imágenes y voz), redes neuronales artificiales, prueba de teoremas y los juegos son representativos de las aplicaciones de esta categoría.

2.2. INGENIERÍA DEL SOFTWARE

El término ingeniería de software se define en el DRAE (Diccionario de la Real Academia Española) como:

1. Conjunto de conocimientos y técnicas que permiten aplicar el saber científico a la utilización de la materia y de las fuentes de energía.
2. Profesión y ejercicio del ingeniero

Y el término ingeniero se define como:

1. Persona que profesa o ejerce la ingeniería.

De igual modo, la Real Academia de Ciencias Exactas, Físicas y Naturales de España define el término Ingeniería como: conjunto de conocimientos y técnicas cuya aplicación permite la utilización racional de los materiales y de los recursos naturales, mediante invenciones, construcciones u otras realizaciones provechosas para el hombre.

Otras definiciones:

1. Ingeniería del Software es el estudio de los principios y metodologías para el desarrollo y mantenimiento de sistemas de software (Zelkovitz, 1978)
2. Ingeniería del 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 (funcionar) y mantenerlos. Se conoce también como desarrollo de software o producción de software (Bohem, 1976)
3. Ingeniería del 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)
4. La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación (funcionamiento) y mantenimiento del software; es decir, la aplicación de ingeniería al software. (IEEE, 1993)

La definición de IEEE describe la ingeniería del software como un enfoque sistemático cubriendo los aspectos del desarrollo, operación y mantenimiento. Este enfoque es disciplinado y cuantificable.

2.2.1. Historia

El término ingeniería del software apareció por primera vez en la conferencia de ingeniería de software de la OTAN en 1968 y fue mencionado para provocar el pensamiento sobre la crisis de software del momento. Desde entonces, ha continuado como una profesión y campo de estudio dedicado a la creación de software de alta calidad, barato, con capacidad de mantenimiento y rápido de construir. Debido a que el campo es todavía relativamente joven comparado con otros campos de la ingeniería, hay todavía mucho trabajo y debate sobre qué es realmente la ingeniería del software, y si se merece el título de ingeniería. Ha crecido orgánicamente fuera de las limitaciones de ver el software sólo como programación.

Mientras que el término ingeniería del software fue acuñado en una conferencia en 1968, los problemas que intentaba tratar empezaron mucho antes. La historia de la ingeniería del software está entrelazada con las historias contrapuestas de hardware y software.

Cuando el ordenador digital moderno apareció por primera vez en 1941, las instrucciones para hacerlo funcionar estaban conectadas dentro de la maquina. Las personas relacionadas con la ingeniería rápidamente se dieron cuenta de que este diseño no era flexible e idearon la arquitectura de programa almacenado o arquitectura von Neumann. De esta forma la primera división entre “hardware” y “software” empezó con la abstracción usada para tratar la complejidad de la computación.

Los lenguajes de programación empezaron a aparecer en la década de 1950 y este fue otro paso importante en la abstracción. Los lenguajes principales como Fortran, Algol y Cobol se lanzaron a finales de los 50s para tratar con problemas científicos, algorítmicos y de negocio respectivamente. Dijsktra escribió "Go to Statement Considered Harmful" en 1968 y David Parnas introdujo el concepto clave de la modularidad y encapsulación en 1972 para ayudar a los programadores a tratar con la complejidad de los sistemas de software. Un sistema software para gestionar el hardware, denominado sistema operativo también se introdujo, más notablemente por Unix en 1969. En 1967, el lenguaje Simula introdujo el paradigma de la programación orientada a objetos.

Estos avances en software se encontraron con más avances en el hardware. A mediados de los 70s, la microcomputadora fue introducida, haciendo económico a los aficionados a obtener una computadora y escribir software para él. Esto sucesivamente condujo al famoso ordenador personal o PC y Microsoft Windows. El ciclo de vida de desarrollo de software o SDLC también empezó a aparecer como un consenso para la construcción centralizada de software a mediados de los 80s. A finales de los 70s y principios de los 80 se vieron varios lenguajes de programación orientados a objetos inspirados en Simula, incluyendo C++, Smalltalk y Objective C.

El software open source empezó a aparecer a principios de los 90s en la forma de Linux y otros software introduciendo el "bazaar" o el estilo descentralizado de construcción de software. Después aparecieron Internet y la World Wide Web a mediados de los 90s cambiando de nuevo la ingeniería del software. Los sistemas distribuidos ganaron dominio como forma de diseñar sistemas y el lenguaje de programación Java se introdujo como otro paso en la abstracción, teniendo su propia máquina virtual. Varios programadores colaboraron y escribieron el manifiesto ágil que favoreció procesos más ligeros para crear software más barato y en menos tiempo.

2.2.2. Etapas

La ingeniería de software requiere llevar a cabo numerosas tareas, dentro de etapas como las siguientes:

Análisis de requisitos

Extraer los requisitos de un producto software es la primera etapa para crearlo. Mientras que los clientes piensan que ellos saben lo que el software tiene que hacer, se requiere habilidad y experiencia en la ingeniería del software para reconocer requisitos incompletos, ambiguos o contradictorios. El resultado del análisis de requisitos con el cliente se plasma en el documento Especificación de Requisitos. Asimismo, se define un diagrama de entidad/relación, en el que se plasman las principales entidades que participarán en el desarrollo de software.

La captura, análisis y especificación de requisitos (incluso pruebas de ellos), es una parte crucial; de esta etapa depende en gran medida el logro de los objetivos finales. Se han ideado modelos y diversos procesos de trabajo para estos fines. Aunque aún no está formalizada, se habla de la Ingeniería de Requisitos.

La IEEE Std. 830-1998 normaliza la creación de las especificaciones de requisitos software.

Especificación

Es la tarea de escribir detalladamente el software a ser desarrollado, en una forma matemáticamente rigurosa. En la realidad, la mayoría de las buenas especificaciones han sido escritas para entender y afinar aplicaciones que ya estaban desarrolladas. Las especificaciones son más importantes para las interfaces externas, que deben permanecer estables.

Diseño y arquitectura

Se refiere a determinar cómo funcionará el software de forma general sin entrar en detalles. Consisten en incorporar consideraciones de la implementación tecnológica, como el hardware, la red, etc. Se definen los casos de uso para cubrir las funciones que realizará el sistema, y se transformarán las entidades definidas en el análisis de requisitos en clases de diseño, obteniendo un modelo cercano a la programación orientada a objetos.

Programación

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

Prueba

Consiste en comprobar que el software realice correctamente las tareas indicadas en la especificación del problema. Una técnica de prueba es probar por separado cada módulo del software y luego probarlo de forma integral, para así llegar al objetivo. Se considera una buena práctica que las pruebas sean efectuadas por alguien distinto al desarrollador que la programó.

Mantenimiento

Mantener y mejorar el software para solventar errores descubiertos y tratar con nuevos requisitos. El mantenimiento puede ser de cuatro tipos: perfectivo (mejorar la calidad interna de los sistemas), evolutivo (incorporaciones, modificaciones y eliminaciones necesarias en un producto software para cubrir la expansión o cambio en las necesidades del usuario), adaptativo (modificaciones que afectan a los entornos en los que el sistema opera, por ejemplo, cambios de configuración del hardware, software de base, gestores de base de datos, comunicaciones) y correctivo (corrección de errores).

2.2.3. Objetivo primario de la ingeniería del software

Hemos visto que todas las definiciones de la ingeniería del software se centran en el uso de un enfoque sistemático para la construcción de software.

El objetivo primario de la ingeniería del software es construir un producto de alta calidad de una manera oportuna. Trata de conseguir este objetivo primario usando un enfoque de ingeniería.

La ingeniería implica un conjunto de principios fundamentales que deberían seguirse siempre. 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. Si se siguen los principios básicos, esto resulta en productos de alta calidad.

Para conseguir el objetivo de construir productos de alta calidad dentro de la planificación, la ingeniería del software emplea 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 ingeniería del software 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.

2.2.4. Relevancia de la ingeniería del software

Hoy en día, los productos 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.

Los profesionales del software, por lo tanto, no deberían ser dogmáticos sobre la ingeniería del software. No es una religión y no hay verdades absolutas.

2.2.5. Principios de la ingeniería del software

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.

2.2.6. Capas

El enfoque de ingeniería del software cuenta con un compromiso organizacional con la calidad porque no es posible incorporar la ingeniería del software en una organización que no está centrada en conseguir calidad.

La ingeniería del software es una tecnología multicapa. Se puede ver como un conjunto de componentes estratificados, que reposan sobre ese enfoque de calidad.

Figura 2. Capas de la ingeniería del software

Estos **componentes** que forman parte de la ingeniería del software son:

- Procesos: un marco de trabajo que ayuda al jefe de proyecto a controlar la gestión del proyecto y las actividades de ingeniería.
- Métodos: las actividades técnicas requeridas para la creación de productos de trabajo.
- Herramientas: la ayuda automatizada para los procesos y métodos.

2.2.6.1. Procesos

El fundamento de la ingeniería del software es la capa de proceso. El **proceso** define un marco de trabajo para un conjunto de áreas clave de proceso que se deben establecer para la entrega efectiva de la tecnología de la ingeniería del software.

La capa de proceso define el proceso que se usará para construir el software y las actividades y tareas que un jefe de proyecto tiene que gestionar. Por lo tanto, las áreas claves del proceso forman la base del control de gestión de proyectos del software y establecen el contexto en el que se aplican los métodos técnicos, se obtienen productos de trabajo (modelos, documentos, datos, informes, formularios, etc.), se establecen hitos, se asegura la calidad y el cambio se gestiona adecuadamente. El proceso de la ingeniería del software es la unión que mantiene juntas las capas de tecnologías y que permite un desarrollo racional y oportuno de la ingeniería del software.

La capa de proceso:

- Permite al jefe de proyecto planificar una ejecución exitosa del proyecto. La capa de proceso proporciona una hoja de ruta del trabajo de ingeniería del software. Ayuda al jefe de proyecto en la creación de un plan de trabajo viable que aísle tareas de trabajo, responsabilidades, los productos de trabajo producidos, y los mecanismos usados para asegurar calidad en dichos productos de trabajos. Permite la ejecución de proyectos software dentro de un marco de tiempo razonable.
- Proporciona a las personas involucradas el contexto de su trabajo. La capa de proceso guía a las personas involucradas proporcionando el marco de trabajo en el que entienden el contexto de las tareas a realizar.

Se pueden ver todas las actividades, incluyendo las actividades técnicas, como parte del proceso. Además, cualquier recurso, incluyendo herramientas usadas para construir el software también encajan en el proceso. La capa de proceso es, por lo tanto, el fundamento de la ingeniería del software y da soporte a las capas de métodos y herramientas.

Importancia de un proceso

Un proceso es útil porque proporciona claridad en cómo ha de realizarse el trabajo. Cualquier conjunto de actividades humanas complejas se puede convertir en caótico si no hay guías para que las personas puedan realizar las actividades. Un proceso definido responde a las siguientes cuestiones:

- ¿Quién se comunica con quién?
- ¿Cómo se coordinan las actividades interdependientes?
- ¿Quién es responsable de qué trabajo?
- ¿Quién produce qué productos de trabajo, y cómo se evalúan?

Un proceso:

- Identifica todas las actividades y tareas de la ingeniería del software
- Define el flujo de trabajo entre las actividades y tareas
- Identifica los productos de trabajo que se producen
- Especifica los puntos de control de calidad requeridos

Algunas personas ven el desarrollo de software con una perspectiva que requiere habilidades artísticas y de artesanía y que es inherentemente caótico. Se resisten a la idea de usar un proceso definido porque lo ven como incómodo y burocrático y por lo tanto dificulta la creatividad.

Aunque no hay duda de que el desarrollo de software requiere creatividad, la mayoría del software de calidad en la industria se desarrolla por el esfuerzo coordinado de más de una persona. Para cualquier esfuerzo de equipo, el control coordinado es mejor alternativa que la anarquía. La hoja de ruta proporcionada por un proceso es útil para las personas que construyen productos software o que gestionan proyectos.

Todos los enfoques de la construcción de software tienen un proceso, pero en muchos casos, son ad hoc, invisibles y caóticos. Una buena ingeniería de software hace que el proceso de software sea más visible, predecible y más útil para aquellos que construyen software.

La capa de proceso abarca las siguientes cuestiones:

- El marco de trabajo de proceso común (CPF)
- Actividades y tareas de la ingeniería de software
- Puntos de control de calidad
- Definiciones de productos de trabajo
- Gestión de proyectos
- Aseguramiento de la calidad del software
- Gestión de la configuración del software
- Monitorización de proyectos
- Medidas y métricas

2.2.6.2. Métodos

La capa de proceso identifica las tareas de ingeniería que se deben realizar para construir software de alta calidad.

La siguiente capa, la capa de métodos se centra en las actividades técnicas que se deben realizar para conseguir las tareas de ingeniería. Proporciona el “cómo” y cubre las actividades de ingeniería fundamentales.

Los métodos abarcan una gran gama de tareas que incluyen análisis de requisitos, diseño, construcción de programas, pruebas y mantenimiento. Los métodos de la ingeniería del software dependen de un conjunto de principios básicos que gobiernan cada una de las áreas de la tecnología e incluyen actividades de modelado y otras técnicas descriptivas.

La construcción de software implica una amplia colección de actividades técnicas. La capa de métodos contiene los métodos definidos para realizar esas actividades de forma eficiente. Se centra en cómo se han de realizar las actividades técnicas. Los personas

involucradas usan los métodos para realizar las actividades de ingeniería fundamentales necesarias para construir el software.

Las actividades técnicas fundamentales para construir software son:

- Análisis: el análisis es el fundamento de todos los trabajos de ingeniería que siguen. Durante el análisis, se crea el modelo de lo que es requerido por el software.
- Diseño: las actividades de diseño siguen el análisis y traducen el modelo del análisis en cómo el producto proporciona estas funciones por medio del software.
- Codificación: una vez que el diseño es completo, la codificación traduce el modelo de diseño en una forma ejecutable.
- Pruebas: el proceso de pruebas ayuda a destapar errores en el código y el diseño subyacente.

También se realizan actividades de soporte: revisiones técnicas y soporte de métricas.

Para varias actividades de proceso, la capa de métodos contiene el correspondiente conjunto de métodos técnicos para usar. Esto abarca un conjunto de reglas, los modos de representación gráficos o basados en texto, y las guías relacionadas para la evaluación de la calidad de la información representada.

Para definir la capa de métodos, es necesario seleccionar un método adecuado de un amplio rango de métodos disponibles.

Consideramos las actividades de análisis y diseño. Hay una amplia variedad de métodos disponibles. El equipo de proyecto debería seleccionar el método que es más apropiado para el problema, el entorno de desarrollo y el conocimiento y experiencia de los miembros del equipo.

2.2.6.3. Herramientas

La capa de herramientas proporciona soporte a las capas de proceso y métodos centrándose en el significado de la automatización de algunas de las actividades manuales. Las herramientas se pueden utilizar para automatizar las siguientes actividades:

- Actividades de gestión de proyectos
- Métodos técnicos usados en la ingeniería del software
- Soporte de sistemas general
- Marcos de trabajo para otras herramientas

La automatización ayuda a eliminar el tedio del trabajo, reduce las posibilidades de errores, y hace más fácil usar buenas prácticas de ingeniería del software. Cuando se usan herramientas, la documentación se convierte en una parte integral del trabajo hecho, en vez de ser una actividad adicional. De ahí que la documentación no se tenga que realizar como actividad adicional. Las herramientas se pueden utilizar para realizar actividades de gestión de proyecto así como para actividades técnicas.

Existen una gran variedad de herramientas para múltiples actividades. Entre ellas se pueden destacar las siguientes:

- Herramientas de gestión de proyectos
- Herramientas de control de cambios
- Herramientas de análisis y diseño
- Herramientas de generación de código
- Herramientas de pruebas
- Herramientas de reingeniería
- Herramientas de documentación
- Herramientas de prototipos

Estas herramientas soportan las capas de proceso y de métodos en varias actividades.

3. CICLOS DE VIDA DE DESARROLLO DEL SOFTWARE

3.1. CICLOS DE VIDA

El ciclo de vida es el conjunto de fases por las que pasa el sistema que se está desarrollando desde que nace la idea inicial hasta que el software es retirado o remplazado (muere). También se denomina a veces paradigma.

Entre las funciones que debe tener un ciclo de vida se pueden destacar:

- Determinar el orden de las fases del proceso de software
- Establecer los criterios de transición para pasar de una fase a la siguiente
- Definir las entradas y salidas de cada fase
- Describir los estados por los que pasa el producto
- Describir las actividades a realizar para transformar el producto
- Definir un esquema que sirve como base para planificar, organizar, coordinar, desarrollar…

Un ciclo de vida para un proyecto se compone de fases sucesivas compuestas por tareas que se pueden planificar. Según el modelo de ciclo de vida, la sucesión de fases puede ampliarse con bucles de realimentación, de manera que lo que conceptualmente se considera una misma fase se pueda ejecutar más de una vez a lo largo de un proyecto, recibiendo en cada pasada de ejecución aportaciones a los resultados intermedios que se van produciendo (realimentación).

- Fases: una fase es un conjunto de actividades relacionadas con un objetivo en el desarrollo del proyecto. Se construye agrupando tareas (actividades elementales) que pueden compartir un tramo determinado del tiempo de vida de un proyecto. La agrupación temporal de tareas impone requisitos temporales correspondientes a la asignación de recursos (humanos, financieros o materiales).
- Entregables: son los productos intermedios que generan las fases. Pueden ser materiales o inmateriales (documentos, software). Los entregables permiten evaluar la marcha del proyecto mediante comprobaciones de su adecuación o no a los requisitos funcionales y de condiciones de realización previamente establecidos.

3.1.1. Tipos de modelo de ciclo de vida

Las principales diferencias entre distintos modelos de ciclo de vida están en:

- El alcance del ciclo dependiendo de hasta dónde llegue el proyecto correspondiente. Un proyecto puede comprender un simple estudio de viabilidad del desarrollo de un producto, o su desarrollo completo o en el extremo, toda la historia del producto con su desarrollo, fabricación y modificaciones posteriores hasta su retirada del mercado.

- Las características (contenidos) de las fases en que dividen el ciclo. Esto puede depender del propio tema al que se refiere el proyecto, o de la organización.
- La estructura y la sucesión de las etapas, si hay realimentación entre ellas, y si tenemos libertad de repetirlas (iterar).

3.2. MODELOS DE CICLO DE VIDA

La ingeniería del software establece y se vale de una serie de modelos que establecen y muestran las distintas etapas y estados por los que pasa un producto software, desde su concepción inicial, pasando por su desarrollo, puesta en marcha y posterior mantenimiento, hasta la retirada del producto. A estos modelos se les denomina “Modelos de ciclo de vida del software”. El primer modelo concebido fue el de Royce, más comúnmente conocido como Cascada o “Lineal Secuencial”. Este modelo establece que las diversas actividades que se van realizando al desarrollar un producto software, se suceden de forma lineal.

Los modelos de ciclo de vida del software describen las fases del ciclo de software y el orden en que se ejecutan las fases.

Un modelo de ciclo de vida de software es una vista de las actividades que ocurren durante el desarrollo de software, intenta determinar el orden de las etapas involucradas y los criterios de transición asociados entre estas etapas.

Un **modelo de ciclo de vida del software**:

- Describe las fases principales de desarrollo de software
- Define las fases primarias esperadas de ser ejecutadas durante esas fases
- Ayuda a administrar el progreso del desarrollo
- Provee un espacio de trabajo para la definición de un proceso detallado de desarrollo de software

En cada una de las etapas de un modelo de ciclo de vida, se pueden establecer una serie de objetivos, tareas y actividades que lo caracterizan. Existen distintos modelos de ciclo de vida, y la elección de un modelo para un determinado tipo de proyecto es realmente importante; el orden es uno de estos puntos importantes.

Existen varias alternativas de modelos de ciclo de vida. A continuación se muestran algunos de los modelos tradicionales y más utilizados.

3.2.1. Modelo en cascada

Es el enfoque metodológico que ordena rigurosamente las etapas del ciclo de vida del software, de forma que el inicio de cada etapa debe esperar a la finalización de la inmediatamente anterior.

El modelo en cascada es un proceso de desarrollo secuencial, en el que el desarrollo se ve fluyendo hacia abajo (como una cascada) sobre las fases que componen el ciclo de vida.

Se cree que el modelo en cascada fue el primer modelo de proceso introducido y seguido ampliamente en la ingeniería el software. La innovación estuvo en la primera vez que la ingeniería del software fue dividida en fases separadas.

La primera descripción formal del modelo en cascada se cree que fue en un artículo publicado en 1970 por Winston W. Royce, aunque Royce no usó el término cascada en este artículo. Irónicamente, Royce estaba presentando este modelo como un ejemplo de modelo que no funcionaba, defectuoso.

En el modelo original de Royce, existían las siguientes fases:

1. Especificación de requisitos
2. Diseño
3. Construcción (Implementación o codificación)
4. Integración
5. Pruebas
6. Instalación
7. Mantenimiento

Para seguir el modelo en cascada, se avanza de una fase a la siguiente en una forma puramente secuencial.

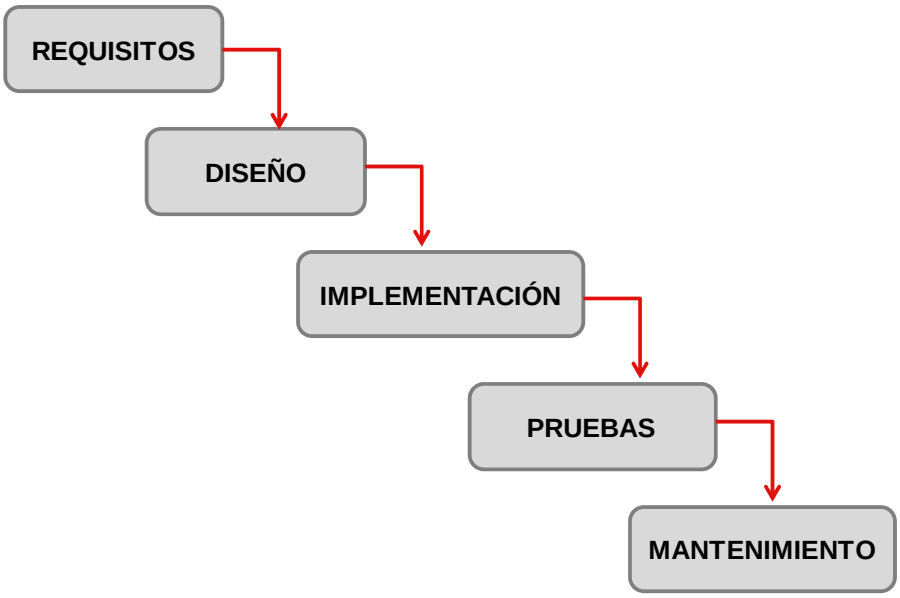

Figura 3. Modelo de ciclo de vida en cascada

Si bien ha sido ampliamente criticado desde el ámbito académico y la industria, sigue siendo el paradigma más seguido a día de hoy.

3.2.1.1. Ventajas

El modelo en cascada puede ser apropiado, en general, para proyectos estables (especialmente los proyectos con requisitos no cambiantes) y donde es posible y probable que los diseñadores predigan totalmente áreas de problema del sistema y produzcan un diseño correcto antes de que empiece la implementación. Funciona bien para proyectos pequeños donde los requisitos están bien entendidos.

Es un modelo en el que todo está bien organizado y no se mezclan las fases. Es simple y fácil de usar.

Debido a la rigidez del modelo es fácil de gestionar ya que cada fase tiene entregables específicos y un proceso de revisión. Las fases son procesadas y completadas de una vez.

3.2.1.2. Inconvenientes

En la vida real, un proyecto rara vez sigue una secuencia lineal, esto crea una mala implementación del modelo, lo cual hace que lo lleve al fracaso.

Difícilmente un cliente va a establecer al principio todos los requisitos necesarios, por lo que provoca un gran atraso trabajando en este modelo, ya que este es muy restrictivo y no permite movilizarse entre fases.

Los resultados y/o mejoras no son visibles progresivamente, el producto se ve cuando ya está finalizado, lo cual provoca una gran inseguridad por parte del cliente que quiere ir viendo los avances en el producto. Esto también implica el tener que tratar con requisitos que no se habían tomado en cuenta desde el principio, y que surgieron al momento de la implementación, lo cual provocará que haya que volver de nuevo a la fase de requisitos.

Hay muchas personas que argumentan que es una mala idea en la práctica, principalmente a causa de su creencia de que es imposible, para un proyecto no trivial, conseguir tener una fase del ciclo de vida del producto software perfecta antes de moverse a las siguientes fases. Por ejemplo, los clientes pueden no ser conscientes exactamente de los requisitos que quieren antes de ver un prototipo del trabajo; pueden cambiar los requisitos constantemente, y los diseñadores e implementadores pueden tener poco control sobre esto. Si los clientes cambian sus requisitos después de que el diseño está terminado, este diseño deberá ser modificado para acomodarse a los nuevos requisitos, invalidando una buena parte del esfuerzo.

Muchas veces se considera un modelo pobre para proyectos complejos, largos, orientados a objetos y por supuesto en aquellos en los que los requisitos tengan un riesgo de moderado a alto de cambiar. Genera altas cantidades de riesgos e incertidumbres.

3.2.1.3. Variantes

Existen muchas variantes de este modelo. En respuesta a los problemas percibidos con el modelo en cascada puro, se introdujeron muchos modelos de cascada modificados. Estos modelos pueden solventar algunas o todas las críticas del modelo en cascada puro.

De hecho muchos de los modelos utilizados tienen su base en el modelo en cascada.

Uno de estos modelos modificados es el modelo **Sashimi**.

El modelo sashimi (llamado así porque tiene fases solapadas, como el pescado japonés sashimi) fue creado originalmente por Peter DeGrace. A veces se hace referencia a él como el modelo en cascada con fases superpuestas o el modelo en cascada con retroalimentación. Ya que las fases en el modelo sashimi se superponen, lo que implica que se puede actuar durante las etapas anteriores. Por ejemplo, ya que las fases de diseño e implementación se superpondrán en el modelo sashimi, los problemas de implementación se pueden descubrir durante las fases de diseño e implementación del proceso de desarrollo. Esto ayuda a aliviar muchos de los problemas asociadas con la filosofía del modelo en cascada.

3.2.2. Modelo en V

El modelo en v se desarrolló para terminar con algunos de los problemas que se vieron utilizando el enfoque de cascada tradicional. Los defectos estaban siendo encontrados demasiado tarde en el ciclo de vida, ya que las pruebas no se introducían hasta el final del proyecto. El modelo en v dice que las pruebas necesitan empezarse lo más pronto posible en el ciclo de vida. También muestra que las pruebas no son sólo una actividad basada en la ejecución. Hay una variedad de actividades que se han de realizar antes del fin de la fase de codificación. Estas actividades deberían ser llevadas a cabo en paralelo con las actividades de desarrollo, y los técnicos de pruebas necesitan trabajar con los desarrolladores y analistas de negocio de tal forma que puedan realizar estas actividades y tareas y producir una serie de entregables de pruebas. Los productos de trabajo generados por los desarrolladores y analistas de negocio durante el desarrollo son las bases de las pruebas en uno o más niveles. El modelo en v es un modelo que ilustra cómo las actividades de prueba (verificación y validación) se pueden integrar en cada fase del ciclo de vida. Dentro del modelo en v, las pruebas de validación tienen lugar especialmente durante las etapas tempranas, por ejemplo, revisando los requisitos de usuario y después por ejemplo, durante las pruebas de aceptación de usuario.

El modelo en v es un proceso que representa la secuencia de pasos en el desarrollo del ciclo de vida de un proyecto. Describe las actividades y resultados que han de ser producidos durante el desarrollo del producto. La parte izquierda de la v representa la descomposición de los requisitos y la creación de las especificaciones del sistema. El lado derecho de la v representa la integración de partes y su verificación. V significa “Validación y Verificación”.

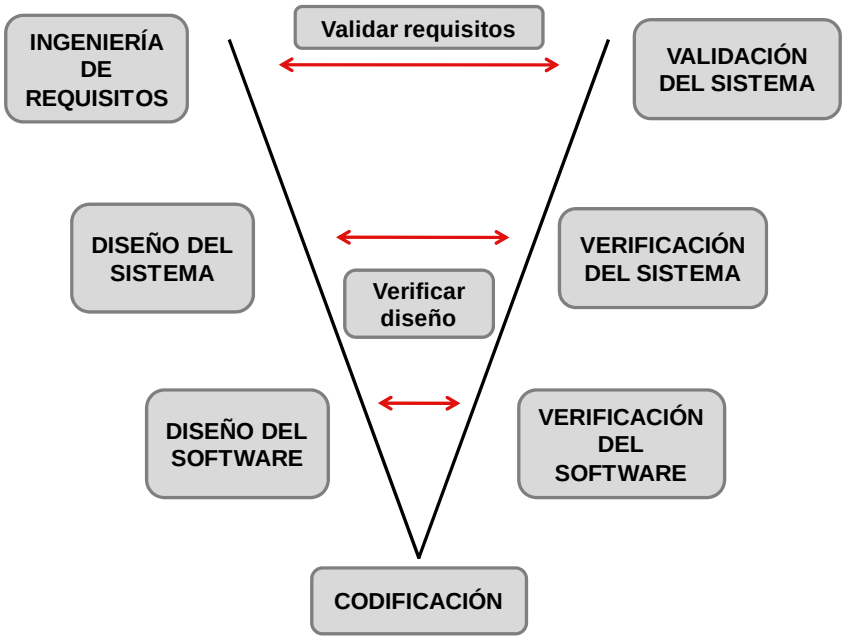

Figura 4. Modelo de ciclo de vida en V

Realmente las etapas individuales del proceso pueden ser casi las mismas que las del modelo en cascada. Sin embargo hay una gran diferencia. En vez de ir para abajo de una forma lineal las fases del proceso vuelven hacia arriba tras la fase de codificación, formando una v. La razón de esto es que para cada una de las fases de diseño se ha encontrado que hay un homólogo en las fases de pruebas que se correlacionan.

3.2.2.1. Ventajas

Las ventajas que se pueden destacar de este modelo son las siguientes:

- Es un modelo simple y fácil de utilizar.
- En cada una de las fases hay entregables específicos.
- Tiene una alta oportunidad de éxito sobre el modelo en cascada debido al desarrollo de planes de prueba en etapas tempranas del ciclo de vida.
- Es un modelo que suele funcionar bien para proyectos pequeños donde los requisitos son entendidos fácilmente.

3.2.2.2. Inconvenientes

Entre los inconvenientes y las críticas que se le hacen a este modelo están las siguientes:

- Es un modelo muy rígido, como el modelo en cascada.
- Tiene poca flexibilidad y ajustar el alcance es difícil y caro.
- El software se desarrolla durante la fase de implementación, por lo que no se producen prototipos del software.

- El modelo no proporciona caminos claros para problemas encontrados durante las fases de pruebas

3.2.3. Modelo iterativo

Es un modelo derivado del ciclo de vida en cascada. Este modelo busca reducir el riesgo que surge entre las necesidades del usuario y el producto final por malos entendidos durante la etapa de recogida de requisitos.

Consiste en la **iteración** de varios ciclos de vida en cascada. Al final de cada iteración se le entrega al cliente una versión mejorada o con mayores funcionalidades del producto. El cliente es quien después de cada iteración evalúa el producto y lo corrige o propone mejoras. Estas iteraciones se repetirán hasta obtener un producto que satisfaga las necesidades del cliente.

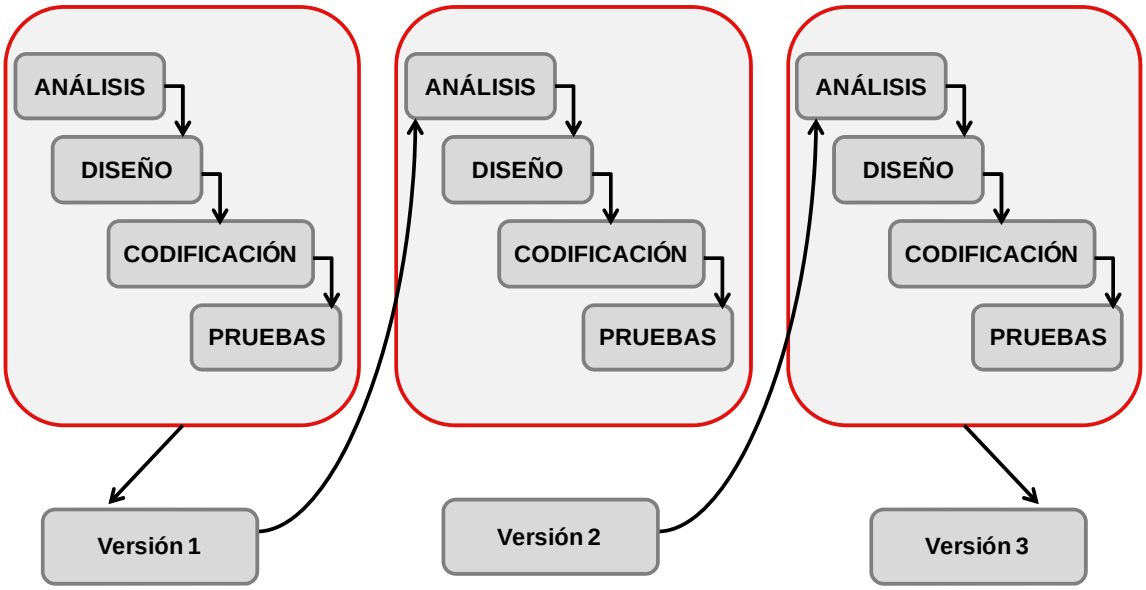

Figura 5. Modelo de ciclo de vida iterativo

Este modelo se suele utilizar en proyectos en los que los requisitos no están claros por parte del usuario, por lo que se hace necesaria la creación de distintos prototipos para presentarlos y conseguir la conformidad del cliente.

3.2.3.1. Ventajas

Una de las principales ventajas que ofrece este modelo es que no hace falta que los requisitos estén totalmente definidos al inicio del desarrollo, sino que se pueden ir refinando en cada una de las iteraciones.

Igual que otros modelos similares tiene las ventajas propias de realizar el desarrollo en pequeños ciclos, lo que permite gestionar mejor los riesgos, gestionar mejor las entregas…

3.2.3.2. Inconvenientes

La primera de las ventajas que ofrece este modelo, el no ser necesario tener los requisitos definidos desde el principio, puede verse también como un inconveniente ya que pueden surgir problemas relacionados con la arquitectura.

3.2.4. Modelo de desarrollo incremental

El modelo incremental combina elementos del modelo en cascada con la filosofía interactiva de construcción de prototipos. Se basa en la filosofía de construir incrementando las funcionalidades del programa. Este modelo aplica secuencias lineales de forma escalonada mientras progresa el tiempo en el calendario. Cada secuencia lineal produce un incremento del software.

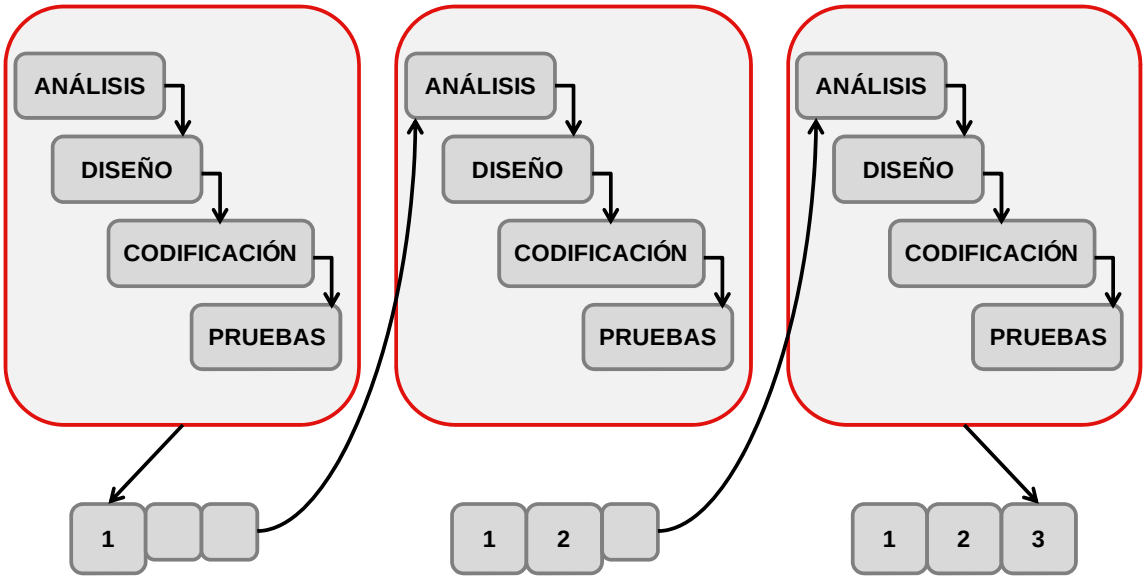

Figura 6. Modelo de ciclo de vida incremental

Cuando se utiliza un modelo incremental, el primer incremento es a menudo un producto esencial, sólo con los requisitos básicos. Este modelo se centra en la entrega de un producto operativo con cada incremento. Los primeros incrementos son versiones incompletas del producto final, pero proporcionan al usuario la funcionalidad que precisa y también una plataforma para la evaluación.

3.2.4.1. Ventajas

Entre las ventajas que puede proporcionar un modelo de este tipo encontramos las siguientes:

- Mediante este modelo se genera software operativo de forma rápida y en etapas tempranas del ciclo de vida del software.
- Es un modelo más flexible, por lo que se reduce el coste en el cambio de alcance y requisitos.

- Es más fácil probar y depurar en una iteración más pequeña.
- Es más fácil gestionar riesgos.
- Cada iteración es un hito gestionado fácilmente

3.2.4.2. Inconvenientes

Para el uso de este modelo se requiere una experiencia importante para definir los incrementos y distribuir en ellos las tareas de forma proporcionada.

Entre los inconvenientes que aparecen en el uso de este modelo podemos destacar los siguientes:

- Cada fase de una iteración es rígida y no se superponen con otras.
- Pueden surgir problemas referidos a la arquitectura del sistema porque no todos los requisitos se han reunido, ya que se supone que todos ellos se han definido al inicio.

3.2.5. Modelo en espiral

El desarrollo en espiral es un modelo de ciclo de vida desarrollado por Barry Boehm en 1985, utilizado de forma generalizada en la ingeniería del software. Las actividades de este modelo se conforman en una espiral, cada bucle representa un conjunto de actividades. Las actividades no están fijadas a priori, sino que las siguientes se eligen en función del análisis de riesgos, comenzando por el bucle anterior.

Boehm, autor de diversos artículos de ingeniería del software; modelos de estimación de esfuerzos y tiempo que se consume en hacer productos software; y modelos de ciclo de vida, ideó y promulgó un modelo desde un enfoque distinto al tradicional en Cascada: el Modelo Evolutivo Espiral. Su modelo de ciclo de vida en espiral tiene en cuenta fuertemente el riesgo que aparece a la hora de desarrollar software. Para ello, se comienza mirando las posibles alternativas de desarrollo, se opta por la de riesgos más asumibles y se hace un ciclo de la espiral. Si el cliente quiere seguir haciendo mejoras en el software, se vuelven a evaluar las nuevas alternativas y riesgos y se realiza otra vuelta de la espiral, así hasta que llegue un momento en el que el producto software desarrollado sea aceptado y no necesite seguir mejorándose con otro nuevo ciclo.

Este modelo de desarrollo combina las características del modelo de prototipos y el modelo en cascada. El modelo en espiral está pensado para proyectos largos, caros y complicados.

Esto modelo no fue el primero en tratar el desarrollo iterativo, pero fue el primer modelo en explicar las iteraciones.

Este modelo fue propuesto por Boehm en 1988 en su artículo “A Spiral Model of Software Development and Enhancement”. Básicamente consiste en una serie de ciclos que se repiten en forma de espiral, comenzando desde el centro. Se suele interpretar como que dentro de cada ciclo de la espiral se sigue un modelo en cascada, pero no necesariamente ha de ser así.

Este sistema es muy utilizado en proyectos grandes y complejos como puede ser, por ejemplo, la creación de un sistema operativo.

Al ser un modelo de ciclo de vida orientado a la gestión de riesgos se dice que uno de los aspectos fundamentales de su éxito radica en que el equipo que lo aplique tenga la necesaria experiencia y habilidad para detectar y catalogar correctamente riesgos.

Tareas:

Para cada ciclo habrá cuatro actividades:

1. Determinar o fijar objetivos:
    a. Fijar también los productos definidos a obtener: requerimientos, especificación, manual de usuario.
    b. Fijar las restricciones
    c. Identificación de riesgos del proyecto y estrategias alternativas para evitarlos
    d. Hay una cosa que solo se hace una vez: planificación inicial o previa
2. Análisis del riesgo:
    a. Se estudian todos los riesgos potenciales y se seleccionan una o varias alternativas propuestas para reducir o eliminar los riesgos
3. Desarrollar, verificar y validar (probar):
    a. Tareas de la actividad propia y de prueba
    b. Análisis de alternativas e identificación de resolución de riesgos
    c. Dependiendo del resultado de la evaluación de riesgos, se elige un modelo para el desarrollo, que puede ser cualquiera de los otros existentes, como formal, evolutivo, cascada, etc. Así, por ejemplo, si los riesgos de la interfaz de usuario son dominantes, un modelo de desarrollo apropiado podría ser la construcción de prototipos evolutivos.
4. Planificar:
    a. Revisamos todo lo que hemos hecho, evaluándolo y con ello decidimos si continuamos con las fases siguientes y planificamos la próxima actividad.

El proceso empieza en la posición central. Desde allí se mueve en el sentido de las agujas del reloj.

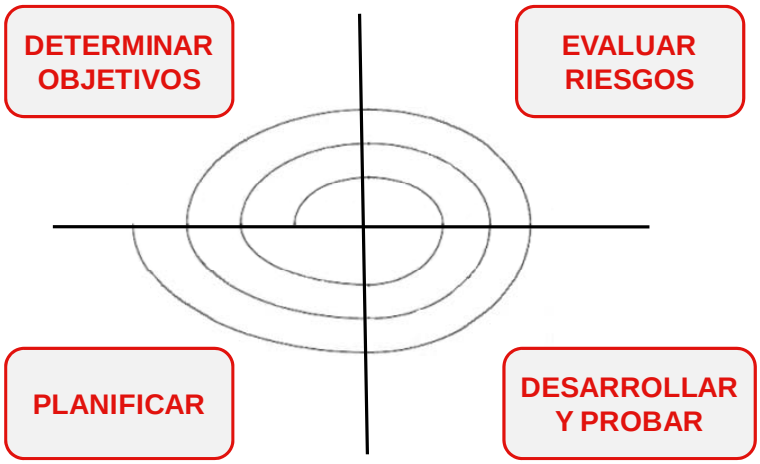

Figura 7. Ciclo de vida en espiral

Las cuatro regiones del gráfico son:

- La tarea de planificación: para definir recursos, responsabilidades, hitos y planificaciones
- La tarea de determinación de objetivos: para definir los requisitos y las restricciones para el producto y definir las posibles alternativas
- La tarea de análisis de riesgos: para evaluar riesgos tanto técnicos como de gestión
- La tarea de ingeniería: para diseñar e implementar uno o más prototipos o ejemplos de la aplicación

3.2.5.1. Ventajas

El análisis de riesgos se hace de forma explícita y clara. Une los mejores elementos de los restantes modelos. Entre ellos:

- Reduce riesgos del proyecto
- Incorpora objetivos de calidad
- Integra el desarrollo con el mantenimiento

Además es posible tener en cuenta mejoras y nuevos requerimientos sin romper con el modelo, ya que el ciclo de vida no es rígido ni estático.

Mediante este modelo se produce software en etapas tempranas del ciclo de vida y suele ser adecuado para proyectos largos de misión crítica.

3.2.5.2. Inconvenientes

Es un modelo que genera mucho trabajo adicional. Al ser el análisis de riesgos una de las tareas principales exige un alto nivel de experiencia y cierta habilidad en los analistas de riesgos (es bastante difícil).

Esto puede llevar a que sea un modelo costoso. Además, no es un modelo que funcione bien para proyectos pequeños.

3.2.5.3. Comparación con otros modelos

La distinción más destacada entre el modelo en espiral y otros modelos de software es la tarea explícita de evaluación de riesgos. Aunque la gestión de riesgos es parte de otros procesos también, no tiene una representación propia en el modelo de proceso. Para otros modelos la evaluación de riesgos es una subtarea, por ejemplo, de la planificación y gestión global. Además no hay fases fijadas para la especificación de requisitos, diseño y pruebas en el modelo en espiral. Se puede usar prototipado para encontrar y definir los requisitos.

La diferencia entre este modelo y el modelo de ciclo incremental es que en el incremental se parte de que no hay incertidumbre en los requisitos iniciales; en este, en cambio, se es consciente de que se comienza con un alto grado de incertidumbre. En el incremental suponemos que conocemos el problema y lo dividimos. Este modelo gestiona la incertidumbre.

3.2.6. Modelo de Prototipos

Un cliente, a menudo, define un conjunto de objetivos generales para el software, pero no identifica los requisitos detallados de entrada, proceso o salida. En otros casos, el responsable del desarrollo del software puede no estar seguro de la eficiencia de un algoritmo, de la calidad de adaptación de un sistema operativo, o de la forma en que debería tomarse la interacción hombre-máquina. En estas y en otras muchas situaciones, un paradigma de construcción de prototipos puede ofrecer el mejor enfoque.

El paradigma de construcción de prototipos comienza con la recolección de requisitos. El desarrollador y el cliente encuentran y definen los objetivos globales para el software, identifican los requisitos conocidos y las áreas del esquema en donde es obligatoria más definición. Entonces aparece un diseño rápido. El diseño rápido se centra en una representación de esos aspectos del software que serán visibles para el usuario/cliente. El diseño rápido lleva a la construcción de un prototipo. El prototipo lo evalúa el cliente/usuario y se utiliza para refinar los requisitos del software a desarrollar. La iteración ocurre cuando el prototipo se pone a punto para satisfacer las necesidades del cliente, permitiendo al mismo tiempo que el desarrollador comprenda mejor lo que se necesita hacer.

Figura 8. Modelo de ciclo de vida de prototipos

3.2.6.1. Ventajas

Entre las ventajas que ofrece este modelo se pueden destacar las siguientes:

Ofrece visibilidad del producto desde el inicio del ciclo de vida con el primer prototipo. Esto puede ayudar al cliente a definir mejor los requisitos y a ver las necesidades reales del producto. Permite introducir cambios en las iteraciones siguientes del ciclo. Permite la realimentación continua del cliente.

El prototipo es un documento vivo de buen funcionamiento del producto final. El cliente reacciona mucho mejor ante el prototipo, sobre el que puede experimentar, que no sobre una especificación escrita.

Este modelo reduce el riesgo de construir productos que no satisfagan las necesidades de los usuarios.

3.2.6.2. Inconvenientes

Entre los inconvenientes que se han observado con este modelo está el hecho de que puede ser un desarrollo lento. Además se hacen fuertes inversiones en un producto desechable ya que los prototipos se descartan. Esto puede hacer que aumente el coste de desarrollo del producto.

Con este modelo pueden surgir problemas con el cliente que ve funcionando versiones del prototipo pero puede desilusionarse porque el producto final aún no ha sido construido. El desarrollador puede caer en la tentación de ampliar el prototipo para construir el sistema final sin tener en cuenta los compromisos de calidad y de mantenimiento que tiene con el cliente.

3.3. ISO/IEC 12207

Esta norma establece un marco de referencia común para los procesos del ciclo de vida del software, con una terminología bien definida a la que puede hacer referencia la industria del software. Contiene procesos, actividades y tareas para aplicar durante la adquisición de un sistema que contiene software, un producto software puro o un servicio software, y durante el suministro, desarrollo, operación y mantenimiento de productos software. El software incluye la parte software del firmware.

Esta norma incluye también un proceso que puede emplearse para definir, controlar y mejorar los procesos del ciclo de vida del software.

La ISO 12207 define un **modelo de ciclo de vida** como un marco de referencia que contiene los procesos, actividades y tareas involucradas en el desarrollo, operación y mantenimiento de un producto software, y que abarca toda la vida del sistema, desde la definición de sus requisitos hasta el final del uso.

Esta norma agrupa las actividades que pueden llevarse a cabo durante el ciclo de vida del software en cinco procesos principales, ocho procesos de apoyo y cuatro procesos organizativos. Cada proceso del ciclo de vida está dividido en un conjunto de actividades; cada actividad se subdivide a su vez en un conjunto de tareas.

- **Procesos principales del ciclo de vida**: son cinco procesos que dan servicio a las partes principales durante el ciclo de vida del software. Una parte principal es la que inicia o lleva a cabo el desarrollo, operación y mantenimiento de productos software. Los procesos principales son:
    - Proceso de adquisición
    - Proceso de suministro
    - Proceso de desarrollo
    - Proceso de operación
    - Proceso de mantenimiento
- **Procesos de apoyo al ciclo de vida**: son procesos que apoyan a otros procesos como parte esencial de los mismos, con un propósito bien definido, y contribuyen al éxito y calidad del proyecto software. Un proceso de apoyo se emplea y ejecuta por otro proceso según sus necesidades.

  Los procesos de apoyo son:
    - Proceso de documentación
    - Proceso de gestión de la configuración
    - Proceso de verificación
    - Proceso de validación
    - Proceso de revisiones conjuntas
    - Proceso de auditoría

  - Proceso de solución de problemas

- **Procesos organizativos del ciclo de vida**: se emplean por una organización para establecer e implementar una infraestructura construida por procesos y personal asociado al ciclo de vida, y para mejorar continuamente esta estructura y procesos.
  - Proceso de gestión
  - Proceso de infraestructura
  - Proceso de mejora
  - Proceso de formación

4. METODOLOGÍAS DE DESARROLLO DE SOFTWARE

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 de décadas 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.

4.1. DEFINICIÓN DE METODOLOGÍA

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.

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

La metodología para el desarrollo de software en 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.

Una definición estándar de metodología puede ser 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.

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.
- 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.

4.2. VENTAJAS DEL USO DE UNA METODOLOGÍA

Son muchas las ventajas que puede aportar el uso de una metodología. A continuación se van a exponer algunas de ellas, clasificadas desde distintos puntos de vista.

Desde el punto de vista de gestión:

- Facilitar la tarea de planificación
- Facilitar la tarea del control y seguimiento de un proyecto
- Mejorar la relación coste/beneficio
- Optimizar el uso de recursos disponibles
- Facilitar la evaluación de resultados y cumplimiento de los objetivos
- Facilitar la comunicación efectiva entre usuarios y desarrolladores

Desde el punto de vista de los ingenieros del software:

- Ayudar a la comprensión del problema
- Optimizar el conjunto y cada una de las fases del proceso de desarrollo
- Facilitar el mantenimiento del producto final
- Permitir la reutilización de partes del producto

Desde el punto de vista del cliente o usuario:

- Garantía de un determinado nivel de calidad en el producto final
- Confianza en los plazos de tiempo fijados en la definición del proyecto
- Definir el ciclo de vida que más se adecue a las condiciones y características del desarrollo

4.3. METODOLOGÍAS TRADICIONALES Y ÁGILES

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.

Según la filosofía de desarrollo se pueden clasificar las metodologías en dos grupos. Las metodologías tradicionales, que se basan en una fuerte planificación durante todo el desarrollo, y las metodologías ágiles, en las que el desarrollo de software es incremental, cooperativo, sencillo y adaptado.

Metodologías tradicionales

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

Centran su atención en llevar una documentación exhaustiva de todo el proyecto y en cumplir con un plan de proyecto, definido todo esto, en la fase inicial del desarrollo del proyecto.

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.

Las metodologías tradicionales (formales) se focalizan en la documentación, planificación y procesos (plantillas, técnicas de administración, revisiones, etc.)

Metodologías ágiles

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.

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

¿Metodologías ágiles o metodologías tradicionales?

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.

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

Figura 9. Tabla comparativa entre metodologías tradicionales y desarrollo ágil

5. DESARROLLO ITERATIVO E INCREMENTAL

El desarrollo iterativo e incremental es un proceso de desarrollo de software cíclico desarrollado en respuesta a la debilidad del modelo en cascada. Empieza con una planificación inicial y termina con el despliegue, con la iteración cíclica en el medio.

Para apoyar al desarrollo de proyectos por medio de este modelo se han creado distintos frameworks, entornos de trabajo, como puede ser el Rational Unified Process. 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.

El desarrollo iterativo e incremental es una parte esencial de RUP, de DSDM, XP y generalmente de los marcos de trabajo de desarrollo de software ágil.

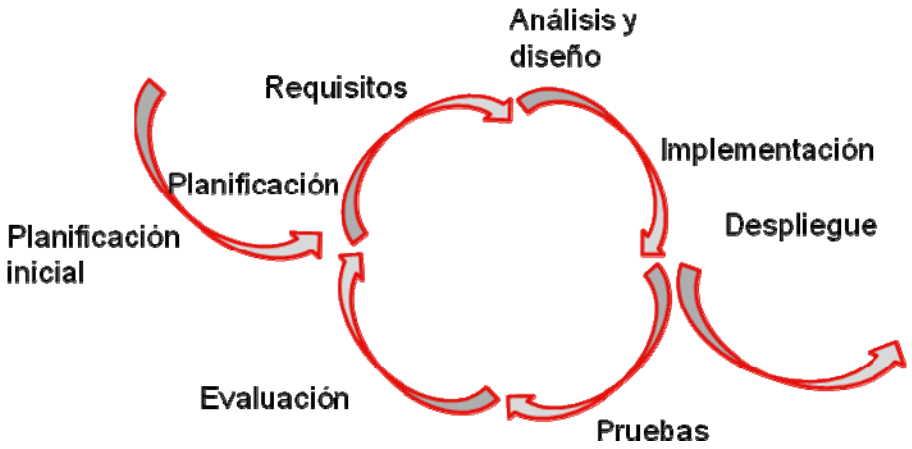

Figura 10. Desarrollo iterativo e incremental

El desarrollo incremental es una estrategia programada y en etapas, en la que las diferentes partes del sistema se desarrollan en diferentes momentos o a diferentes velocidades, y se integran a medida que se completan.

El desarrollo iterativo es una estrategia de programación de reproceso en la que el tiempo se separa para revisar y mejorar partes del sistema. Esto no presupone desarrollo incremental, pero trabaja muy bien con él. Una diferencia típica es que la salida de un incremento no está necesariamente sujeta a más refinamiento, y sus pruebas o la realimentación del usuario no se usa como entrada para revisar los planes o especificaciones de los incrementos sucesivos. Por el contrario, la salida de una iteración se examina para modificación, y especialmente para revisar los objetivos de las sucesivas iteraciones.

Los dos términos se pusieron en práctica a mediados de los 90s. Los autores del Proceso Unificado (UP) y el proceso unificado Rational (RUP) seleccionaron el término desarrollo iterativo e iteraciones para hacer referencia de forma general a cualquier combinación de desarrollo incremental e iterativo. La mayoría de las personas que dicen desarrollo iterativo quieren decir que hacen ambos desarrollo incremental e iterativo.

5.1. LA IDEA BÁSICA

La idea básica detrás de la mejora iterativa es desarrollar un sistema software incrementalmente, permitiendo al desarrollador aprovechar lo que va a ir aprendiendo durante el desarrollo de versiones anteriores, incrementales y entregables del sistema. El aprendizaje viene tanto del desarrollo como del uso del sistema, donde sea posible. Pasos clave en el proceso son empezar con una implementación simple de un subconjunto de requisitos del software y mejorar iterativamente la secuencia evolutiva de versiones hasta que se implementa el sistema entero. En cada iteración, se hacen modificaciones del diseño y se añaden nuevas capacidades.

El procedimiento en sí consiste en el paso de Inicialización, el paso de Iteración y la lista de control del proyecto. El paso de inicialización crea una versión base del sistema. El objetivo de esta implementación inicial es crear un producto ante el que los usuarios puedan reaccionar. Debería ofrecer un muestreo de los aspectos clave del problema y proponer una solución que sea lo suficientemente simple de entender e implementar fácilmente. Para guiar el proceso de iteración, se crea una lista de control de proyecto que contiene un registro de todas las tareas que necesitan ser realizadas. Incluye elementos como pueden ser nuevas características a ser implementadas y áreas de rediseño de la solución existente. La lista de control se revisa constantemente como resultado de la fase de análisis.

La iteración implica el rediseño y la implementación de una tarea de la lista de control del proyecto, y el análisis de la versión actual del sistema. El objetivo del diseño e implementación de cualquier iteración es simple, sencillo y modular, soportando el rediseño en esta etapa o tarea añadida a la lista del control del proyecto. El nivel de detalle del diseño no está establecido por el enfoque iterativo. En un proyecto iterativo de poco peso el código puede representar la mayor fuente de documentación del sistema; sin embargo, en un proyecto iterativo de misión crítica se debe usar un documento de diseño de software formal. El análisis de una iteración se basa en la retroalimentación del usuario y las facilidades de análisis del programa disponibles. Esto implica análisis de la estructura, modularidad, usabilidad, fiabilidad, eficiencia y éxito de los objetivos. La lista de control del proyecto se modifica en vista de los resultados del análisis.

El desarrollo iterativo divide el valor de negocio entregable (funcionalidad del sistema) en iteraciones. En cada iteración se entrega una parte de la funcionalidad a través de un trabajo multidisciplinar, empezando por el modelo/requisitos hasta las pruebas/despliegue. El proceso unificado agrupa las iteraciones en fases: inicio, elaboración, construcción y transición.

- El inicio identifica el alcance del proyecto, los riesgos y los requisitos (funcionales y no funcionales) a un alto nivel en suficiente detalle para que se pueda estimar el trabajo.
- La elaboración entrega una arquitectura de trabajo que mitiga los riesgos altos y cumple los requisitos no funcionales.

- La construcción reemplaza incrementalmente la arquitectura con código listo para producción del análisis, diseño, implementación y pruebas de los requisitos funcionales.
- La transición entrega el sistema al entorno operativo de producción.

Cada una de las fases puede dividirse en una o más iteraciones, que se agrupan en función de tiempo más que de característica. Los arquitectos y analistas trabajan una iteración por delante de los desarrolladores y testers.

5.2. DEBILIDADES EN EL MODELO

- Debido a la interacción con los usuarios finales, cuando sea necesaria la retroalimentación hacia el grupo de desarrollo, utilizar este modelo de desarrollo puede llevar a avances extremadamente lentos.
- Por la misma razón no es una aplicación ideal para desarrollos en los que de antemano se sabe que serán grandes en el consumo de recursos y largos en el tiempo.
- Al requerir constantemente la ayuda de los usuarios finales, se agrega un coste extra a la compañía, pues mientras estos usuarios evalúan el software dejan de ser directamente productivos para la compañía.

5.3. RAPID APPLICATION DEVELOPMENT (RAD)

La metodología de desarrollo rápido de aplicaciones (RAD) se desarrolló para responder a la necesidad de entregar sistemas muy rápido. El enfoque de RAD no es apropiado para todos los proyectos. El alcance, el tamaño y las circunstancias, todo ello determina el éxito de un enfoque RAD.

El método RAD tiene una lista de tareas y una estructura de desglose de trabajo diseñada para la rapidez. El método comprende el desarrollo iterativo, 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 rapidez de ejecución.

A continuación, se muestra un flujo de proceso posible para el desarrollo rápido de aplicaciones:

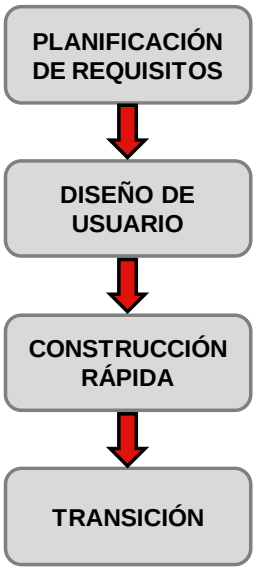

Figura 11. Flujo de proceso de RAD

El desarrollo rápido de aplicaciones es un proceso de desarrollo de software, desarrollado inicialmente por James Martin en 1980. El término fue usado originalmente para describir dicha metodología. La metodología de Martin implicaba desarrollo iterativo y la construcción de prototipos. Más recientemente, el término y su acrónimo se están usando en un sentido genérico, más amplio, que abarca una variedad de técnicas dirigidas al desarrollo de aplicaciones rápidas.

El desarrollo rápido de aplicaciones fue una respuesta a los procesos no ágiles de desarrollo desarrollados en los 70 y 80, tales como el método de análisis y diseño de sistemas estructurados y otros modelos en cascada. Un problema con las metodologías previas era que llevaba mucho tiempo la construcción de las aplicaciones y esto podía llevar a la situación de que los requisitos podían haber cambiado antes de que se completara el sistema, resultando en un sistema inadecuado o incluso no usable. Otro problema era la suposición de que una fase de análisis de requisitos metódica sola identificaría todos los requisitos críticos. Un evidencia amplia avala el hecho de que esto se da rara vez, incluso para proyectos con profesionales de alta experiencia a todos los niveles.

Empezando con las ideas de Brian Gallagher, Alex Balchin, Barry Boehm y Scott Shultz, James Martin desarrolló el enfoque de desarrollo rápido de aplicaciones durante los 80 en IBM y lo formalizó finalmente en 1991, con la publicación del libro, “Desarrollo rápido de aplicaciones”.

Es una fusión de varias técnicas estructuradas, especialmente la ingeniería de información orientada a datos con técnicas de prototipos para acelerar el desarrollo de sistemas software.

RAD requiere el uso interactivo de técnicas estructuradas y prototipos para definir los requisitos de usuario y diseñar el sistema final. Usando técnicas estructuradas, el desarrollador primero construye modelos de datos y modelos de procesos de negocio preliminares de los requisitos. Los prototipos ayudan entonces al analista y los usuarios a

verificar tales requisitos y a refinar formalmente los modelos de datos y procesos. El ciclo de modelos resulta a la larga en una combinación de requisitos de negocio y una declaración de diseño técnico para ser usado en la construcción de nuevos sistemas.

Los enfoques RAD pueden implicar compromisos en funcionalidad y rendimiento a cambio de permitir el desarrollo más rápido y facilitando el mantenimiento de la aplicación.

Ventajas

Las principales ventajas que puede aportar este tipo de desarrollo son las siguientes:

- Velocidad de desarrollo
- Calidad: según lo definido por el RAD, es el grado al cual un uso entregado resuelve las necesidades de usuarios así como el grado al cual un sistema entregado tiene costes de mantenimiento bajos. El RAD aumenta la calidad con la implicación del usuario en las etapas del análisis y del diseño.
- Visibilidad temprana debido al uso de técnicas de prototipado.
- Mayor flexibilidad que otros modelos.
- Ciclos de desarrollo más cortos.

Las ventajas que puede añadir sobre el seguimiento de un método en cascada, por ejemplo, es que en el método en cascada hay un largo periodo de tiempo hasta que el cliente puede ver cualquier resultado. El desarrollo puede llevar tanto tiempo que el negocio del cliente haya cambiado sustancialmente en el momento en el que el software está listo para usar. Con este tipo de métodos no hay visibilidad del producto hasta que el proceso no está finalizado al 100%, que es cuando se entrega el software.

Inconvenientes

Entre los principales inconvenientes que se pueden encontrar en el uso del desarrollo rápido de aplicaciones se pueden encontrar:

- Características reducidas.
- Escalabilidad reducida.
- Más difícil de evaluar el progreso porque no hay hitos clásicos.

Una de las críticas principales que suele generar este tipo de desarrollo es que, ya que el desarrollo rápido de aplicaciones es un proceso iterativo e incremental, puede conducir a una sucesión de prototipos que nunca culmine en una aplicación de producción satisfactoria. Tales fallos pueden ser evitados si las herramientas de desarrollo de la aplicación son robustas, flexibles y colocadas para el uso correcto.

5.4. RATIONAL UNIFIED PROCESS (RUP)

El proceso unificado Rational (RUP) es un marco de trabajo de proceso de desarrollo de software iterativo creado por Rational Software Corporation, una división de IBM desde 2003. RUP no es un proceso preceptivo concreto individual, sino un marco de trabajo de proceso adaptable, con la idea de ser adaptado por las organizaciones de desarrollo y los equipos de proyecto de software que seleccionarán los elementos del proceso que sean apropiados para sus necesidades.

RUP fue originalmente desarrollado por Rational Software, y ahora disponible desde IBM. El producto incluye una base de conocimiento con artefactos de ejemplo y descripciones detalladas para muchos tipos diferentes de actividades.

RUP resultó de la combinación de varias metodologías y se vio influenciado por métodos previos como el modelo en espiral. Las consideraciones clave fueron el fallo de proyectos usando métodos monolíticos del estilo del modelo en cascada y también la llegada del desarrollo orientado a objetos y las tecnologías GUI, un deseo de elevar el modelado de sistemas a la práctica del desarrollo y de resaltar los principios de calidad que aplicaban a las manufacturas en general al software.

Los creadores y desarrolladores del proceso se centraron en el diagnóstico de las características de diferentes proyectos de software fallidos. De esta forma intentaron reconocer las causas raíz de tales fallos. También se fijaron en los procesos de ingeniería del software existentes y sus soluciones para estos síntomas.

El fallo de los proyectos es causado por una combinación de varios síntomas, aunque cada proyecto falla de una forma única. La salida de su estudio fue un sistema de mejores prácticas del software al que llamaron RUP.

El proceso fue diseñado con las mismas técnicas con las que el equipo solía diseñar software; tenía un modelo orientado a objetos subyacente, usando UML (Unified Modeling Language)

5.4.1. Módulos de RUP (building blocks)

RUP se basa en un conjunto de módulos o elementos de contenido, que describen qué se va a producir, las habilidades necesarias requeridas y la explicación paso a paso describiendo cómo se consiguen los objetivos de desarrollo. Los módulos principales, o elementos de contenido, son:

- Roles (quién): un rol define un conjunto de habilidades, competencias y responsabilidades relacionadas.
- Productos de trabajo (qué): un producto de trabajo representa algo que resulta de una tarea, incluyendo todos los documentos y modelos producidos mientras que se trabaja en el proceso.
- Tareas (cómo): una tarea describe una unidad de trabajo asignada a un rol que proporciona un resultado significante.

5.4.2. Fases del ciclo de vida del proyecto

RUP determina que el ciclo de vida del proyecto consiste en cuatro fases. Estas fases permiten que el proceso sea presentado a alto nivel de una forma similar a como sería presentado un proyecto basado en un estilo en cascada, aunque en esencia la clave del proceso recae en las iteraciones de desarrollo dentro de todas las fases. También, cada fase tiene un objetivo clave y un hito al final que denota que el objetivo se ha logrado.

Las cuatro fases en las que divide el ciclo de vida del proyecto son:

- Fase de iniciación: se define el alcance del proyecto.
- Fase de elaboración: se analizan las necesidades del negocio en mayor detalle y se define sus principios arquitectónicos.
- Fase de construcción: se crea el diseño de la aplicación y el código fuente.
- Fase de transición: se entrega el sistema a los usuarios.

RUP proporciona un prototipo al final de cada iteración.

Dentro de cada iteración, las tareas se categorizan en nueve disciplinas:

- Seis disciplinas de ingeniería
    - Modelaje de negocio
    - Requisitos
    - Análisis y diseño
    - Implementación
    - Pruebas
    - Despliegue
- Tres disciplinas de soporte
    - Gestión de la configuración y del cambio
    - Gestión de proyectos
    - Entorno

5.4.3. Certificación

Existe un examen de certificación IBM Certified Solution Designer – Rational Unified Process 7.0. Este examen prueba tanto contenido relacionado con el contenido de RUP como relacionado con los elementos de estructura del proceso.

Es una certificación a nivel personal. El examen tiene una duración de 75 minutos y son 52 preguntas. Hay que conseguir un 62% para aprobar.

5.5. DESARROLLO ÁGIL

El desarrollo de software ágil hace referencia a un grupo de metodologías de desarrollo de software que se basan en principios similares. Generalmente promueven:

- Un proceso de gestión de proyectos que fomenta la inspección y adaptación frecuente.
- Una filosofía líder que fomenta trabajo en equipo, organización propia y responsabilidad.
- Un conjunto de mejores prácticas de ingeniería que permite la entrega rápida de software de alta calidad.
- Un enfoque de negocio que alinea el desarrollo con las necesidades de los clientes y los objetivos de la compañía

En el apartado siguiente se va a describir el desarrollo ágil en profundidad, haciendo hincapié en los métodos y prácticas más extendidas.

6. DESARROLLO ÁGIL

Hasta hace poco el proceso de desarrollo llevaba asociado un marcado énfasis en el control del proceso mediante una rigurosa definición de roles, actividades y artefactos, incluyendo modelado y documentación detallada. Este esquema tradicional para abordar el desarrollo del software ha demostrado ser efectivo y necesario en proyectos de gran tamaño (respecto a tiempo y recursos), donde por lo general se exige un alto grado de ceremonia en el proceso. Sin embargo, este enfoque no resulta ser el más adecuado para muchos de los proyectos actuales donde el entorno del sistema es muy cambiante y en donde se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. Ante las dificultades para utilizar metodologías tradicionales con estas restricciones de tiempo y flexibilidad, muchos equipos de desarrollo se resignan a prescindir del "buen hacer" de la ingeniería del software, asumiendo el riesgo que ello conlleva. En este escenario, las metodologías ágiles emergen como una posible respuesta para llenar ese vacío metodológico. Por estar especialmente orientadas para proyectos pequeños, las metodologías ágiles constituyen una solución a medida para ese entorno, aportando una elevada simplificación que a pesar de ello no renuncia a las prácticas esenciales para asegurar la calidad del producto.

Las metodologías ágiles son sin duda uno de los temas recientes en la ingeniería de software que están acaparando gran interés. Prueba de ello es que se están haciendo un espacio destacado en la mayoría de conferencias y talleres celebrados en los últimos años. En la comunidad de la ingeniería del software, se está viviendo con intensidad un debate abierto entre los partidarios de las metodologías tradicionales y aquellos que apoyan las ideas surgidas del "manifiesto ágil".

El desarrollo ágil de software es un grupo de metodologías de desarrollo de software que se basan en principios similares. Las metodologías ágiles promueven generalmente un proceso de gestión de proyectos que fomenta el trabajo en equipo, la organización y responsabilidad propia, un conjunto de mejores prácticas de ingeniería que permiten la entrega rápida de software de alta calidad, y un enfoque de negocio que alinea el desarrollo con las necesidades del cliente y los objetivos de la compañía.

El desarrollo ágil elige hacer las cosas en incrementos pequeños con una planificación mínima, más que planificaciones a largo plazo. Las iteraciones son estructuras de tiempo pequeñas (conocidas como "timeboxes") que típicamente duran de 1 a 4 semanas. De cada iteración se ocupa un equipo realizando un ciclo de desarrollo completo, incluyendo planificación, análisis de requisitos, diseño, codificación, pruebas unitarias y pruebas de aceptación. Esto ayuda a minimizar el riesgo general, y permite al proyecto adaptarse a los cambios rápidamente. La documentación se produce a medida que es requerida por los agentes involucrados. Una iteración puede no añadir suficiente funcionalidad para garantizar una liberación del producto al mercado, pero el objetivo es tener una versión disponible (con errores mínimos) al final de cada iteración. Se requerirán múltiples iteraciones para liberar un producto o nuevas características.

La composición del **equipo** en un proyecto ágil es normalmente **multidisciplinar** y de organización propia sin consideración de cualquier jerarquía corporativa existente o los roles corporativos de los miembros de los equipos. Los miembros de los equipos normalmente toman responsabilidades de tareas que consigan la funcionalidad de una iteración. Deciden ellos mismos cómo realizarán las tareas durante una iteración.

Los métodos ágiles enfatizan la **comunicación cara a cara** sobre los documentos escritos. La mayoría de los equipos ágiles se encuentran localizados en una única ubicación abierta para facilitar esta comunicación. El tamaño del equipo es normalmente pequeño (5-9 personas) para ayudar a hacer la comunicación y la colaboración más fácil. Los esfuerzos de desarrollos largos deben ser repartidos entre múltiples equipos trabajando hacia un objetivo común o partes diferentes de un esfuerzo. Esto puede requerir también una coordinación de prioridades a través de los equipos.

La mayoría de las metodologías ágiles incluyen una comunicación cara a cara rutinaria, diaria y formal entre los miembros del equipo. Esto incluye específicamente al representante del cliente y a cualquier persona involucrada en el negocio interesada como observadores. En una breve sesión, los miembros del equipo informan al resto de lo que hicieron el día anterior, lo que van a hacer hoy y cuáles son sus principales obstáculos. Esta comunicación cara a cara permanente previene problemas que se puedan esconder.

No importa qué disciplinas de desarrollo se requieran, cada equipo ágil contendrá un **representante del cliente**. Esta persona es designada por las personas involucradas en el negocio para actuar en su nombre y hacer un compromiso personal de estar disponible para los desarrolladores para responder preguntas. Al final de cada iteración, las personas involucradas en el negocio y el representante del cliente revisan el progreso y reevalúan las prioridades con vistas a optimizar el retorno de la inversión y asegurando la alineación con las necesidades del cliente y los objetivos de la compañía.

Los métodos ágiles enfatizan software operativo como la medida principal del progreso. Combinado con la preferencia de comunicación cara a cara, los métodos ágiles normalmente producen menos documentación escrita que otros métodos.

6.1. HISTORIA

La definición moderna del desarrollo de software ágil se desarrolló hacia la mitad de los 90 como parte de una reacción contra los métodos denominados “pesados”, como el modelo en cascada. El proceso originado por el uso del modelo en cascada se veía como burocrático, lento, degradante e inconsistente con las formas en las que los desarrolladores de software realizaban trabajo efectivo. Inicialmente, a los métodos ágiles se les llamó métodos ligeros. En 2001, miembros relevantes de la comunidad se juntaron en Utah y adoptaron el nombre de metodologías ágiles. Después, algunas de estas personas formaron la alianza ágil, una organización sin ánimo de lucro que promociona el desarrollo ágil.

Entre los primeros métodos ágiles más notables están Scrum, Crystal Clear, Extreme Programming, Adaptative Software Development, Feature Driven Development and Dynamic Systems Development method. Se hace referencia a ellos como metodologías ágiles desde que se publicó el manifiesto ágil en 2001.

6.2. COMPARACIÓN CON OTROS MÉTODOS

Los métodos ágiles se caracterizan a veces como diametralmente opuestos a métodos orientados a pruebas o disciplinados. Esta distinción es errónea, ya que esto implica que los métodos ágiles no son planificados ni disciplinados. Una distinción más exacta es que los métodos pueden ser desde "adaptativos" a "predictivos". Los métodos ágiles se acercan más al lado adaptativo.

Los **métodos adaptativos** se centran en la adaptación rápida a los cambios. Cuando las necesidades de un proyecto cambian, un equipo adaptativo cambia también. Un equipo adaptativo tendrá dificultades para describir lo que pasará en el futuro. Cuanto más lejana sea una fecha, más vago será un método adaptativo en cuanto a saber qué pasará en esa fecha. Un equipo adaptativo puede informar exactamente de las tareas que se realizarán la semana que viene, pero sólo las características planificadas para el próximo mes. Cuando se pregunta por una versión de dentro de seis meses, un equipo adaptativo sólo podrá ser capaz de reportar la declaración de la misión de las versiones o una declaración de la relación valor coste esperada.

Los **métodos predictivos**, por el contrario, se centran en la planificación del futuro en detalle. Un equipo predictivo puede informar exactamente de las características y tareas planificadas para el proceso de desarrollo entero. Los equipos predictivos tienen dificultades a la hora de cambiar de dirección. El plan está típicamente optimizado para el destino original y el cambio de dirección puede causar que se desperdicie trabajo realizado y se tenga que hacer de manera muy diferente. Los equipos predictivos inician un comité de control de cambios para asegurar que sólo los cambios que aporten más valor sean considerados.

Los métodos ágiles tienen mucho en común con las técnicas de RAD (desarrollo rápido de aplicaciones).

6.2.1. Comparación con otros métodos de desarrollo iterativos

La mayoría de métodos ágiles comparten con otros métodos de desarrollo iterativos e incrementales el énfasis en construir software liberable en cortos periodos de tiempo. El desarrollo ágil se diferencia de otros modelos de desarrollo en que en este modelo los periodos de tiempo se miden en semanas más que en meses y el trabajo se realiza de una forma colaborativa.

6.2.2. Comparación con el modelo en cascada

El desarrollo ágil tiene poco en común con el modelo en cascada. El modelo en cascada es el más estructurado de los métodos, pasando a través de la secuencia pre-planificada y estricta de captura de requisitos, análisis, diseño, codificación y pruebas. El progreso se mide generalmente en términos de artefactos entregables: especificaciones de requisitos, documentos de diseño, planes de pruebas, revisiones de código y similares.

El problema principal con el modelo en cascada es la división inflexible de un proyecto en etapas separadas, de tal forma que los compromisos se hacen al principio y es difícil

reaccionar a los cambios en los requisitos. Las iteraciones son caras. Esto significa que el modelo en cascada es probable que no sea adecuado si los requisitos no están bien entendidos o cuando es probable que cambien en el curso del proyecto.

Los métodos ágiles, por el contrario, producen características completamente desarrolladas y probadas (pero sólo un pequeño subconjunto del proyecto) cada pocas semanas. El énfasis está en obtener la menor parte de funcionalidad factible para entregar valor de negocio pronto y mejorar/ añadir más funcionalidad continuamente durante la vida del proyecto.

En este aspecto, las críticas ágiles afirman que estas características no tienen lugar en el contexto del proyecto global, concluyendo que, si el sponsor del proyecto está preocupado por contemplar ciertos objetivos con un periodo de tiempo o un presupuesto definido, las metodologías ágiles no son apropiadas. Los partidarios del desarrollo ágil responden que las adaptaciones de Scrum muestran cómo los métodos ágiles están incrementando la producción y mejora continua de un plan estratégico.

Algunos equipos ágiles usan el modelo en cascada a pequeña escala, repitiendo el ciclo de cascada entero en cada iteración. Otros equipos, los más notables los equipos de XP, trabajan en actividades simultáneamente.

6.2.3. Comparación con codificación “cowboy”

La codificación cowboy es la ausencia de un método definido: los miembros del equipo hacen lo que creen que está bien. Las frecuentes reevaluaciones de planes del desarrollo ágil, enfatizan en la comunicación cara a cara y el uso escaso de documentos causa a veces que la gente se confunda con la codificación cowboy. Los equipos ágiles, sin embargo, siguen procesos definidos (con frecuencia disciplinados y rigurosos).

Como con todos los métodos de desarrollo, las habilidades y la experiencia de los usuarios determinan el grado de éxito.

6.3. IDONEIDAD DE LOS MÉTODOS ÁGILES

No hay mucho consenso en qué tipo de proyectos de software son los más adecuados para las metodologías ágiles. Muchas organizaciones grandes tienen dificultades en salvar el espacio entre un método más tradicional como el de cascada y uno ágil.

El desarrollo software ágil a larga escala permanece en un área de investigación activa.

El desarrollo ágil ha sido ampliamente documentado como que funciona bien para equipos pequeños localizados en un mismo sitio (<10 desarrolladores).

Algunas cosas que pueden afectar negativamente al éxito de un proyecto ágil son:

- Esfuerzos de desarrollo a gran escala (>20 desarrolladores), aunque se han descrito estrategias de escalado y evidencias que dicen lo contrario.
- Esfuerzos de desarrollo distribuido (equipos no localizados en un mismo sitio)
- Culturas de compañías de ordenar y controlar

- Forzar un proceso ágil en un equipo de desarrollo

Se han documentado varios proyectos ágiles a gran escala exitosos.

Barry Boehm y Richard Turner sugieren que se pueden usar un análisis de riesgos para elegir entre métodos adaptativos (ágiles) y predictivos (orientados a planes).

A continuación de muestran algunos de los elementos que pueden caracterizar a proyectos desarrollados con metodologías ágiles:

- Baja criticidad
- Desarrolladores seniors
- Los requisitos cambian con mucha frecuencia
- Pequeño número de desarrolladores
- Cultura que prospera al caos

Y para el caso de proyectos desarrollados con metodologías orientadas a planes, se pueden observar los siguientes elementos:

- Alta criticidad
- Desarrolladores juniors
- Los requisitos no cambian con mucha frecuencia
- Gran número de desarrolladores
- Cultura que demanda orden

6.4. EL MANIFIESTO ÁGIL

Las metodologías ágiles son una familia de metodologías, no un enfoque individual de desarrollo de software. En 2001, 17 figuras destacadas en el campo del desarrollo ágil (llamado entonces metodologías de peso ligero) se juntaron en la estación de esquí Snowbird en Utah para tratar el tema de la unificación de sus metodologías. Crearon el **manifestó ágil**, ampliamente considerado como la definición canónica del desarrollo ágil.

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 fue el Manifiesto Ágil, documento que resume la filosofía “ágil”.

6.4.1. Manifiesto para el desarrollo de software ágil

Estamos descubriendo nuevas formas de desarrollar software haciéndolo y ayudando a otros a hacerlo. A través de este trabajo hemos venido a valorar:

- Individuos e interacciones sobre procesos y herramientas. La gente es el principal factor de éxito de un proyecto software. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte automáticamente. Es mejor crear el equipo y que éste configure su propio entorno de desarrollo en base a sus necesidades.
- Software que funciona sobre documentación exhaustiva. La regla a seguir es “no producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante”. Estos documentos deben ser cortos y centrarse en lo fundamental.
- Colaboración de clientes sobre la negociación del contrato. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.
- Respuestas a cambios sobre seguir un plan. Responder a los cambios más que seguir estrictamente un plan. La habilidad de responder a los cambios que puedan surgir a lo largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

En cada una de las cuatro afirmaciones lo que quieren resaltar es que aunque los elementos de la parte de la derecha de la comparación tienen valor, ellos valoran más los elementos de la izquierda.

6.4.2. Principios detrás del manifiesto ágil

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

- La prioridad más alta es satisfacer al cliente a través de la entrega temprana y continua de software de valor.
- Los cambios en los requisitos son bienvenidos, incluso tarde en el desarrollo. Los procesos ágiles aprovechan los cambios como ventaja competitiva del cliente.
- Entregar software que funcione con frecuencia, desde un par de semanas hasta un par de meses, con preferencia de escalas de tiempo cortas (con el menor intervalo de tiempo posible).
- Las personas de negocio y los desarrolladores deben trabajar juntos diariamente durante todo el proyecto.

- Construir proyectos alrededor de individuos motivados. Darles el entorno y el soporte necesario, y confiar en que ellos harán el trabajo.
- El método más eficiente y efectivo de hacer llegar información a o dentro de un equipo de desarrollo es en una conversación cara a cara.
- Software que funciona es la medida primaria (principal) del progreso.
- Los procesos ágiles promueven desarrollo sostenible. Los sponsors, desarrolladores y usuarios deberían ser capaces de mantener un ritmo constante indefinido.
- La atención continua a la excelencia técnica y los buenos diseños aumentan la agilidad.
- Simplicidad, el arte de maximizar la cantidad de trabajo no hecho, es esencial.
- Las mejores arquitecturas, requisitos y diseño surgen de equipos organizados por sí mismos.
- A intervalos regulares, los equipos reflexionan sobre cómo ser más efectivos, entonces afinan y ajustan su comportamiento de acuerdo con ello.

6.5. MÉTODOS ÁGILES

La adaptación de métodos se define como: “Un proceso o capacidad en el que agentes humanos a través de cambios responsables, e interacciones dinámicas entre contextos, y métodos determinan un enfoque de desarrollo de un sistema para una situación de proyecto específica”

Potencialmente, casi todos los métodos ágiles son adecuados para la adaptación. La conveniencia de situación puede considerarse como una característica de distinción entre los métodos ágiles y los métodos tradicionales, que son mucho más rígidos y perceptivos. La implicación práctica es que los métodos ágiles permiten a los equipos de proyecto adaptar las prácticas de trabajo de acuerdo con las necesidades del proyecto individual. Las prácticas son actividades y productos concretos que son parte del marco de trabajo del método. En un nivel más extremo, la filosofía detrás del método, que consiste en un número de principios, podría ser adaptada.

XP hace la necesidad de la adaptación de métodos explícita. Una de las ideas fundamentales de XP es que un proceso no es adecuado para todos los proyectos, sino más bien que las prácticas deberían ser adaptadas a las necesidades de los proyectos individuales. La adopción parcial de prácticas XP ha sido informada en varias ocasiones.

6.5.1. Gestión de proyectos

Los métodos ágiles suelen cubrir la gestión de proyectos. Algunos métodos se suplementan con guías en gestión de proyectos. PRINCE2 se ha propuesto como un sistema de gestión de proyectos complementario y adecuado.

Algunas herramientas de gestión de proyectos están dirigidas para el desarrollo ágil. Están diseñadas para planificar, hacer seguimiento, analizar e integrar trabajo. Estas herramientas

juegan un papel importante en el desarrollo ágil, como medios para la gestión del conocimiento.

Algunas de las características comunes incluyen: integración de control de versiones, seguimiento de progreso, asignación de trabajo, versiones integradas y planificación de iteraciones, foros de discusión e informe y seguimiento de defectos de software. La gestión del valor ganado es una técnica de gestión de proyectos aprobada por el PMI (Project Management Institute) para medir objetivamente el éxito del proyecto.

6.5.2. Extreme Programming (XP)

La programación extrema (XP) es un enfoque de la ingeniería del software formulado por Kent Beck. 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.

Se puede considerar la programación extrema como la adopción de las mejores metodologías de desarrollo de acuerdo a lo que se pretende llevar a cabo con el proyecto y aplicarlo de manera dinámica durante el ciclo de vida del software.

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

Los principios y prácticas son de sentido común pero llevadas al extremo, de ahí proviene su nombre.

6.5.2.1. Elementos de la metodología

- **Las historias de usuario**: son la técnica utilizada para especificar los requisitos del software. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales. El tratamiento de las historias de usuario es muy dinámico y flexible. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarlas en unas semanas. Las historias de usuario se descomponen en tareas de programación y se asignan a los programadores para ser implementadas durante una iteración

- **Roles XP**: los roles de acuerdo con la propuesta de Beck son:
    - Programador: el programador escribe las pruebas unitarias y produce el código del sistema
    - Cliente: escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en apoyar mayor valor al negocio.
    - Encargado de pruebas (tester): ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para las pruebas.
    - Encargado de seguimiento (tracker): proporciona realimentación al equipo. Verifica el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, para mejorar futuras estimaciones. Realiza el seguimiento del progreso de cada iteración.
    - Entrenador (coach): es el responsable del proceso global. Debe proveer guías al equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.
    - Consultor: es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto, en el que puedan surgir problemas.
    - Gestor (big boss): es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.
- **Proceso XP**: el ciclo de desarrollo consiste (a grandes rasgos) en los siguientes pasos:

    1. El cliente define el valor de negocio a implementar
    2. El programador estima el esfuerzo necesario para su implementación
    3. El programador construye ese valor
    4. Vuelve al paso 1

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

El ciclo de vida ideal de XP consisten en 6 fases: exploración, planificación de la entrega, iteraciones, producción, mantenimiento y muerte del proyecto.

6.5.2.2. Prácticas

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

- El juego de la planificación. Hay una comunicación frecuente entre el cliente y los programadores. El equipo técnico realiza una estimación del esfuerzo requerido para la implementación de las historias de usuario y los clientes deciden sobre el ámbito y tiempo de las entregas y de cada iteración.
- Entregas pequeñas. Producir rápidamente versiones del sistema que sean operativas, aunque no cuenten con toda la funcionalidad del sistema. Esta versión ya constituye un resultado de valor para el negocio. Una entrega no debería tardar más de 3 meses.
- Metáfora. El sistema es definido mediante una metáfora o un conjunto de metáforas compartidas por el cliente y el equipo de desarrollo. Una metáfora es una historia compartida que describe cómo debería funcionar el sistema (conjunto de nombre que actúen como vocabulario para hablar sobre el dominio del problema, ayudando a la nomenclatura de clases y métodos del sistema).
- Diseño simple. Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto.
- Pruebas. La producción de código está dirigida por las pruebas unitarias. Éstas son establecidas por el cliente antes de escribirse el código y son ejecutadas constantemente ante cada modificación del sistema.
- Refactorización (refactoring). Es una actividad constante de reestructuración del código con el objetivo de evitar duplicación del código, mejorar su legibilidad, simplificarlo y hacerlo más flexible para facilitar los posteriores cambios. Se mejora la estructura interna del código sin alterar su comportamiento externo.
- Programación en parejas. Toda la producción de código debe realizarse con trabajo en parejas de programadores. Esto conlleva ventajas implícitas (menor tasa de errores, mejor diseño, mayor satisfacción de los programadores…). Esta práctica se explicará más en profundidad en apartados sucesivos.
- Propiedad colectiva del código. Cualquier programador puede cambiar cualquier parte del código en cualquier momento.
- Integración continua. Cada pieza de código es integrada en el sistema una vez que esté lista. Así, el sistema puede llegar a ser integrado y construido varias veces en un mismo día. Esta práctica se desarrollará más en profundidad en apartados sucesivos.
- 40 horas por semana. Se debe trabajar un máximo de 40 horas por semana. No se trabajan horas extras en dos semanas seguidas. Si esto ocurre, probablemente está ocurriendo un problema que debe corregirse. El trabajo extra desmotiva al equipo.

- Cliente in-situ. El cliente tiene que estar presente y disponible todo el tiempo para el equipo. Éste es uno de los principales factores de éxito del proyecto XP. El cliente conduce constantemente el trabajo hacia lo que aportará mayor valor al negocio y los programadores pueden resolver de manera más inmediata cualquier duda asociada. La comunicación oral es más efectiva que la escrita.
- Estándares de programación. XP enfatiza que la comunicación de los programadores es a través del código, con lo cual es indispensable que se sigan ciertos estándares de programación para mantener el código legible.

El mayor beneficio de las prácticas se consigue con su aplicación conjunta y equilibrada puesto que se apoyan unas en otras. La mayoría de las prácticas propuestas por XP no son novedosas sino que en alguna forma ya habían sido propuestas en ingeniería del software e incluso demostrado su valor en la práctica. El mérito de XP es integrarlas de una forma efectiva y complementarlas con otras ideas desde la perspectiva del negocio, los valores humanos y el trabajo en equipo.

6.5.2.3. Principios

Los principios básicos de la programación extrema son: simplicidad, comunicación, retroalimentación y coraje.

- **Simplicidad**: la simplicidad es la base de la programación extrema. Se simplifica el diseño para agilizar el desarrollo y facilitar el mantenimiento. Un diseño complejo del código junto a sucesivas modificaciones por parte de diferentes desarrolladores hacen que la complejidad aumente exponencialmente. Para mantener la simplicidad es necesaria la refactorización del código, ésta es la manera de mantener el código simple a medida que crece. También se aplica la simplicidad en la documentación, de esta manera el código debe comentarse en su justa medida, intentando eso sí que el código esté autodocumentado. Para ello se deben elegir adecuadamente los nombres de las variables, métodos y clases. Los nombres largos no disminuyen la eficiencia del código ni el tiempo de desarrollo gracias a las herramientas de autocompletado y refactorización que existen actualmente. Aplicando la simplicidad junto con la autoría colectiva del código y la programación por parejas se asegura que mientras más grande se haga el proyecto, todo el equipo conocerá más y mejor el sistema completo.
- **Comunicación**: la comunicación se realiza de diferentes formas. Para los programadores el código comunica mejor mientras más simple sea. Si el código es complejo hay que esforzarse para hacerlo inteligible. El código autodocumentado es más fiable que los comentarios ya que éstos últimos pronto quedan desfasados con el código a medida que es modificado. Debe comentarse sólo aquello que no va a variar, por ejemplo, el objetivo de una clase o la funcionalidad de un método. Las pruebas unitarias son otra forma de comunicación ya que describen el diseño de las clases y los métodos al mostrar ejemplos concretos de cómo utilizar su funcionalidad. Los programadores se comunican constantemente gracias a la programación por parejas. La comunicación con el cliente es fluida ya que el cliente forma parte del

equipo de desarrollo. El cliente decide qué características tienen prioridad y siempre debe estar disponible para solucionar dudas.

- **Retroalimentación** (feedback): al estar el cliente integrado en el proyecto, su opinión sobre el estado del proyecto se conoce en tiempo real. Al realizarse ciclos muy cortos tras los cuales se muestran resultados, se minimiza el tener que rehacer partes que no cumplen con los requisitos y ayuda a los programadores a centrarse en los que es más importante. Considérense los problemas que derivan de tener ciclos muy largos. Meses de trabajo pueden tirarse por la borda debido a cambios en los criterios del cliente o malentendidos por parte del equipo de desarrollo. El código también es una fuente de retroalimentación gracias a las herramientas de desarrollo. Por ejemplo, las pruebas unitarias informan sobre el estado de salud del código. Ejecutar las pruebas unitarias frecuentemente permite descubrir fallos debidos a cambios recientes en el código.
- **Coraje o valentía**: para los gerentes la programación en parejas puede ser difícil de aceptar, parece como si la productividad se fuese a reducir a la mitad ya que sólo la mitad de los programadores está escribiendo código. Hay que ser valiente para confiar en que la programación por parejas beneficia la calidad del código sin repercutir negativamente en la productividad. La simplicidad es uno de los principios más difíciles de adoptar. Se requiere coraje para implementar las características que el cliente quiere ahora sin caer en la tentación de optar por un enfoque más flexible que permite futuras modificaciones. No se debe emprender el desarrollo de grandes marcos de trabajo mientras el cliente espera. En ese tiempo el cliente no recibe noticias sobre los avances del proyecto y el equipo de desarrollo no recibe retroalimentación para saber si va en la dirección correcta. La forma de construir marcos de trabajo es mediante la refactorización del código en sucesivas aproximaciones.

6.5.2.4. Actividades

XP describe cuatro actividades que se llevan a cabo dentro del proceso de desarrollo de software.

Codificar

Los defensores de XP argumentan que el único producto realmente importante del proceso de desarrollo de sistemas es el código (un concepto al que dan una definición más amplia que la que pueden dar otros). Sin el código no se tiene nada. La codificación puede ser dibujar diagramas que generarán código, hacer scripts de sistemas basados en web o codificar un programa que ha de ser compilado.

La codificación también puede usarse para entender la solución más apropiada. Por ejemplo, XP recomendaría que si nos enfrentamos con varias alternativas para un problema de programación, uno debiera simplemente codificar todas las soluciones y determinar con pruebas automatizadas qué solución es la más adecuada. La codificación puede ayudar también a comunicar pensamientos sobre problemas de programación. Un programador que trate con un problema de programación complejo y encuentre difícil explicar la solución al

resto, podría codificarlo y usar el código para demostrar lo que quería decir. El código, dicen los partidarios de esta posición, es siempre claro y conciso y no se puede interpretar de más de una forma. Otros programadores pueden dar retroalimentación de ese código codificando también sus pensamientos.

Probar

Nadie puede estar seguro de algo si no lo ha probado. Las pruebas no es una necesidad primaria percibida por el cliente. Mucho software se libera sin unas pruebas adecuadas y funciona. En el desarrollo de software, XP dice que esto significa que uno no puede estar seguro de que una función funciona si no la prueba. Esto sugiere la pregunta de definir de lo que uno puede no estar seguro.

- No puedes estar seguro de si lo que has codificado es lo que querías significar. Para probar esta incertidumbre, XP usa pruebas unitarias. Son pruebas automatizadas que prueban el código. El programador intentará escribir todas las pruebas en las que pueda pensar que puedan cargarse el código que está escribiendo; si todas las pruebas se ejecutan satisfactoriamente entonces el código está completo.
- No puedes estar seguro de si lo que querías significar era lo que deberías. Para probar esta incertidumbre, XP usa pruebas de aceptación basadas en los requisitos dados por el cliente.

Escuchar

Los programadores no saben necesariamente todo sobre el lado del negocio del sistema bajo desarrollo. La función del sistema está determinada por el lado del negocio. Para que los programadores encuentren cual debe ser la funcionalidad del sistema, deben escuchar al negocio. Tienen que escuchar las necesidades de los clientes. También tienen que intentar entender el problema del negocio y dar a los clientes retroalimentación sobre el problema, para mejorar el propio entendimiento del cliente sobre el problema.

Diseñar

Desde el punto de vista de la simplicidad, uno podría decir que el desarrollo de sistemas no necesita más que codificar, probar y escuchar. Si estas actividades se desarrollan bien, el resultado debería ser un sistema que funcionase. En la práctica, esto no ocurre. Uno puede seguir sin diseñar, pero un momento dado se va a atascar. El sistema se vuelve muy complejo y las dependencias dentro del sistema dejan de estar claras. Uno puede evitar esto creando una estructura de diseño que organice la lógica del diseño. Buenos diseños evitarán perdidas de dependencias dentro de un sistema; esto significa que cambiar una parte del sistema no tendrá por qué afectar a otras.

6.5.3. SCRUM

Scrum es un proceso ágil que se puede usar para gestionar y controlar desarrollos complejos de software y productos usando prácticas iterativas e incrementales.

Scrum es un proceso incremental iterativo para desarrollar cualquier producto o gestionar cualquier trabajo.

Aunque Scrum estaba previsto que fuera para la gestión de proyectos de desarrollo de software, se puede usar también para la ejecución de equipos de mantenimiento de software o como un enfoque de gestión de programas.

6.5.3.1. Historia

En 1986, Hirotaka Takeuchi e Ikujiro Nonaka describieron un enfoque integral que incrementaba la velocidad y flexibilidad del desarrollo de nuevos productos comerciales. Compararon este nuevo enfoque integral, en el que las fases se solapan fuertemente y el proceso entero es llevado a cabo por un equipo multifuncional a través de las diferentes fases, al rugby, donde todo el equipo trata de ganar distancia como una unidad y pasando el balón una y otra vez.

En 1991, DeGrace y Stahl hicieron referencia a este enfoque como Scrum, un término de rugby mencionado en el artículo de Takeuchi y Nonaka. A principios de 1990s, Ken Schwaber usó un enfoque que guió a Scrum a su compañía, Métodos de Desarrollo Avanzados. Al mismo tiempo, Jeff Sutherland desarrolló un enfoque similar en Easel Corporation y fue la primera vez que se llamó Scrum. En 1995 Sutherland y Schwaber presentaron de forma conjunta un artículo describiendo Scrum en OOPSLA ’95 en Austin, su primera aparición pública. Schwaber y Sutherland colaboraron durante los siguientes años para unir los artículos, sus experiencias y las mejores prácticas de la industria en lo que ahora se conoce como Scrum. En 2001, Schwaber se asoció con Mike Beedle para poner en limpio el método en el libro Agile Software Devlopment with Scrum.

6.5.3.2. Características

Scrum es un esqueleto de proceso que incluye un conjunto de prácticas y roles predefinidos. Los roles principales en Scrum son el “ScrumMaster” que mantiene los procesos y trabaja junto con el jefe de proyecto, el “Product Owner” que representa a las personas implicadas en el negocio y el “Team” que incluye a los desarrolladores.

Durante cada iteración (sprint- periodos de tiempo), típicamente un periodo de 2 a 4 semanas (longitud decidida por el equipo), el equipo crea un incremento de software operativo. El conjunto de características que entra en una iteración viene del “backlog” del producto, que es un conjunto priorizado de requisitos de trabajo de alto nivel que se han de hacer. Los ítems que entran en una iteración se determinan durante la reunión de planificación de la iteración. Durante esta reunión, el Product Owner informa al equipo de los ítems en el backlog del producto que quiere que se completen. El equipo determina entonces a cuanto de eso puede comprometerse a completar durante la siguiente iteración. Durante una iteración, nadie puede cambiar el backlog de la iteración, lo que significa que los requisitos están congelados para esa iteración. Cuando se completa una iteración, el equipo demuestra el uso del software.

Scrum permite la creación de equipos con propia organización fomentando la localización conjunta de todos los miembros del equipo y la comunicación verbal entre todos los miembros del equipo y las disciplinas implicadas en el proyecto.

Un principio clave de Scrum es el reconocimiento de que durante un proyecto los clientes pueden cambiar sus pensamientos sobre lo que quieren y necesitan, y de que los desafíos que no se pueden predecir no se pueden tratar fácilmente de una forma predictiva o planificada tradicional. Por esto, Scrum adopta un enfoque empírico, aceptando que el problema no se puede entender o definir completamente, centrándose en cambio en maximizar las habilidades del equipo para entregar rápidamente y responder a los requisitos emergentes.

Una de las mayores ventajas de Scrum es que es muy fácil de entender y requiere poco esfuerzo para comenzar a usarse.

Una parte muy importante de Scrum son las reuniones que se realizan durante cada una de las iteraciones. Hay distintos tipos:

- Scrum diario: cada día durante la iteración, tiene lugar una reunión de estado del proyecto. A esta reunión se le domina Scrum
- Reunión de planificación de iteración (sprint): se lleva a cabo al principio del ciclo de la iteración.
- Reunión de revisión de iteración: al final del ciclo de la iteración.
- Iteración retrospectiva: al final del ciclo de la iteración.

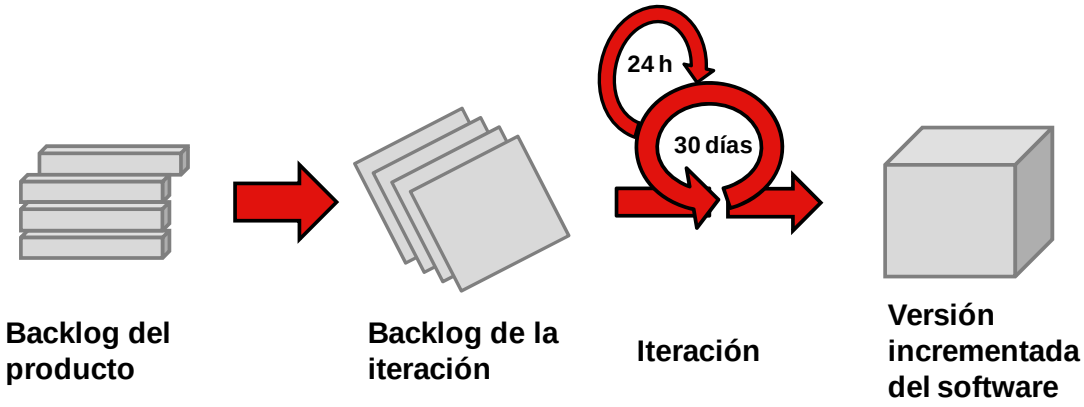

Figura 12. Flujo de proceso de SCRUM

6.5.3.3. Prácticas

A continuación se enumeran algunas de las prácticas de Scrum:

- Los clientes se convierten en parte del equipo de desarrollo.
- Scrum tiene frecuentes entregables intermedios con funcionalidad que funciona, como otras formas de procesos de software ágiles. Esto permite al cliente conseguir trabajar con el software antes y permite al proyecto cambiar los requisitos de acuerdo con las necesidades.

- Se desarrollan planes de riesgos y mitigación frecuentes por parte del equipo de desarrollo, la mitigación de riesgos, la monitorización y la gestión de riesgos se lleva a cabo en todas las etapas y con compromiso.
- Transparencia en la planificación y desarrollo de módulos, permitir a cada uno saber quién es responsable de qué y cuándo.
- Frecuentes reuniones de las personas involucradas en el negocio para monitorizar el progreso.
- Debería haber un mecanismo de advertencias avanzado.
- Los problemas no se han de barrer a debajo de la alfombra. Nadie es penalizado por reconocer o describir un problema imprevisto.

6.5.4. Dynamic Systems Development Method (DSDM)

El método de desarrollo de sistemas dinámico (DSDM) es una metodología de desarrollo de software originalmente basada en la metodología RAD. DSDM es un enfoque iterativo e incremental que enfatiza la participación continua del usuario.

Su objetivo es entregar sistemas software en tiempo y presupuesto ajustándose a los cambios de requisitos durante el proceso de desarrollo. DSDM es uno de los métodos ágiles para el desarrollo de software, y forma parte de la Alianza Ágil.

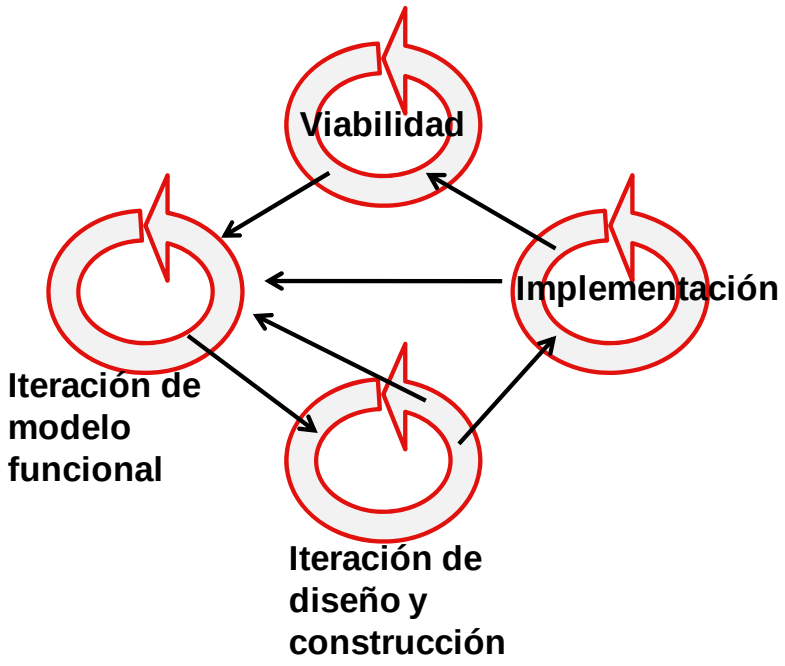

Figura 13. Ciclo de desarrollo de DSDM

Como extensión del desarrollo rápido de aplicaciones, DSDM se centra en proyectos de sistemas de información que se caracterizan por planificaciones y presupuestos estrictos. DSDM trata las características más comunes de los proyectos de sistemas de información, incluyendo presupuestos sobrepasados, plazos de entrega desaparecidos y falta de

participación del usuario y compromiso de la alta gerencia. DSDM consiste en tres fases: fase pre-proyecto, fase de ciclo de vida del proyecto y fase post-proyecto. La fase de ciclo de vida del proyecto está subdividida en 5 etapas: estudio de viabilidad, estudio de negocio, iteración de modelo funcional, iteración de diseño y construcción e implementación.

En algunas circunstancias, hay posibilidades de integrar prácticas de otras metodologías tales como RUP, XP y PRINCE2 como complemento a DSDM. Otro método ágil que tiene alguna similitud en el proceso y concepto de DSDM es Scrum.

DSDM fue desarrollado en Reino Unido en los 90 por el DSDM Consortium, una asociación de vendedores y expertos en el campo de la ingeniería creado con el objetivo de "desarrollar y promover conjuntamente un marco de trabajo RAD independiente" combinando sus propias experiencias. El DSDM Consortium es una organización sin ánimo de lucro que tiene la propiedad y se encarga de la administración del marco de trabajo DSDM. La primera versión se completó en Enero de 1995 y se publicó en Febrero de 1995.

Existe una carrera de certificación en DSDM.

6.5.4.1. El enfoque DSDM

En DSDM hay nueve procesos subyacentes que consisten en cuatro fundamentos y cinco puntos de partida.

- La participación del usuario es la clave principal para llevar a cabo un proyecto eficiente y efectivo, donde ambos, usuarios y desarrolladores compartan un sitio de trabajo, de tal forma que las decisiones se puedan hacer de la forma más exacta.
- Se le deben otorgar poderes al equipo del proyecto para tomar decisiones que son importantes para el progreso del proyecto, sin tener que esperar a aprobaciones de más alto nivel.
- Poner foco en la entrega frecuente de productos, con el supuesto de que entregar algo suficientemente bueno pronto es siempre mejor que entregar todo perfecto al final. Entregando el producto con frecuencia desde una etapa temprana del proyecto, el producto se puede probar y revisar y el registro de pruebas y el documento de revisión se pueden tener en cuenta en la siguiente iteración o fase.
- El principal criterio de aceptación de un entregable es que entregue un sistema que trate las necesidades del negocio actuales. Entregar un sistema perfecto que trate todas las necesidades del negocio posibles es menos importante que centrarse en las funcionalidades críticas.
- El desarrollo es iterativo e incremental y conducido por la retroalimentación del usuario para converger en una solución de negocio efectiva.
- Todos los cambios durante el desarrollo son reversibles
- Se debería hacer una línea base del alcance y los requisitos a alto nivel antes de que el proyecto empiece
- Las pruebas se llevan a cabo a lo largo de todo el ciclo de vida del proyecto.

- La comunicación y la cooperación entre todas las personas involucradas en el negocio son necesarias para ser eficientes y efectivos.

Supuestos adicionales:

- Ningún sistema es construido perfectamente al primer intento. En pocas palabras, el 80% de los beneficios del negocio vienen de un 20% de los requisitos del diseño, por lo tanto DSDM empieza implementando este 20% crítico primero; esto puede producir un sistema que proporcione funcionalidad suficiente para satisfacer al usuario final y el 80% restante se puede añadir después en iteraciones. Esto mitiga el riesgo del proyecto de salirse de los plazos o el presupuesto.
- La entrega de los proyectos ha de ser en tiempo, en presupuesto y con buena calidad
- Cada paso del desarrollo sólo necesita completarse hasta que empiece el siguiente paso. Esto permite que la nueva iteración del proyecto comience sin un retraso innecesario. Los cambios en el diseño pueden coincidir con los cambios en la demanda de los usuarios finales ya que cada iteración del sistema es incremental.
- Se incorporan técnicas de gestión de proyectos y desarrollo
- DSDM se puede aplicar en proyectos nuevos o para ampliar sistemas actuales
- La evaluación de riesgos debería centrarse en la funcionalidad del negocio que se va a entregar, no en el proceso de desarrollo o sus artefactos (tales como documentos de requisitos o diseño)
- La gestión premia la entrega de productos más que la compleción de tareas
- La estimación debería basarse en la funcionalidad del negocio en vez de las líneas de código.

6.5.4.2. Factores de éxito críticos de DSDM

Se han identificado una serie de factores que tienen gran importancia para asegurar proyectos con éxito.

- La aceptación de DSDM por parte de la gerencia y otros empleados. Esto asegura que los diferentes actores del proyecto están motivados desde el principio y que permanece a lo largo del proyecto.
- El compromiso de la gestión de asegurar la participación de usuario final. El enfoque de prototipos requiere una participación fuerte y dedicada del usuario final para probar y juzgar los prototipos funcionales.
- El equipo ha de estar formado por miembros habilidosos que formen una unión estable. Un tema importante es el otorgamiento de poderes al equipo del proyecto. Esto quiere decir que el equipo tiene que poseer el poder y posibilidad de tomar decisiones importantes relacionadas con el proyecto sin tener que escribir propuestas formales a la alta gerencia, lo que puede llevar mucho tiempo. Para que

el equipo del proyecto pueda ejecutar un proyecto con éxito, necesitan también la tecnología correcta para llevar a cabo el proyecto. Esto implica un entorno de desarrollo, herramientas de gestión de proyectos, etc.

6.5.4.3. Comparación con otros métodos de desarrollo

Durante años se han desarrollado y aplicado un gran número de métodos de desarrollo de sistemas de información. Muchos de esos métodos muestran similitudes entre ellos y también con DSDM. Por ejemplo, XP tiene también un enfoque iterativo para el desarrollo con una participación amplia del usuario.

RUP es probablemente el método que más tiene en común con DSDM ya que también es una forma dinámica de desarrollo de sistemas de información. De nuevo el enfoque iterativo se usa en este método de desarrollo.

Como XP y RUP hay muchos otros métodos de desarrollo que muestran similitudes con DSDM, pero DSDM se diferencia de ellos en una serie de cosas. Primero está el hecho de que proporciona un marco de trabajo independiente. Permite a los usuarios completar los pasos específicos del proceso con sus propias técnicas y ayudas software. Otra característica única es el hecho de que las variables en el desarrollo no son tiempo/recursos, sino los requisitos. Y por último el fuerte foco en la comunicación y la participación de todas las personas involucradas en el sistema. Aunque esto también se trata en otros métodos, DSDM cree fuertemente en que el compromiso con el proyecto asegura un resultado satisfactorio.

6.5.5. Otros métodos ágiles

6.5.5.1. Crystal Clear

Crystal Clear es un miembro de la familia de metodologías Crystal como describe Alistair Cockburn y se considera un ejemplo de metodología ágil.

Crystal Clear está pensado para aplicarse a equipos pequeños de 6 a 8 desarrolladores ubicados en el mismo sitio trabajando en sistemas que no son críticos. La familia de metodologías Crystal se centra en la eficiencia y habitabilidad (las personas pueden vivir con él e incluso usarlo) como componentes de la seguridad del proyecto.

Crystal Clear se centra en las personas, no en los procesos o artefactos.

Crystal Clear cuenta con las siguientes propiedades (las tres primeras son requeridas):

- Entrega frecuente de código usable a los usuarios
- Mejora reflexiva
- Comunicación osmótica preferiblemente estando en la misma ubicación
- Seguridad personal
- Fácil acceso a los usuarios expertos

- Pruebas automatizadas, gestión de la configuración e integración frecuente

6.5.5.2. Agile Unified Process (AUP)

El proceso unificado ágil (AUP) es una versión simplificada de RUP desarrollada por Scott Ambler. Describe un enfoque simple, fácil de entender, del desarrollo de software de aplicación de negocios usando técnicas y conceptos ágiles. AUP aplica técnicas ágiles incluyendo desarrollo orientado a pruebas, modelado ágil, gestión de cambios ágil y refactorización de bases de datos para mejorar la productividad.

La naturaleza en serie de AUP se presenta en cuatro fases:

- Inicio: el objetivo es identificar el alcance inicial del proyecto, una arquitectura potencial para el sistema y obtener fondos y aceptación por parte de las personas involucradas en el negocio.
- Elaboración: el objetivo es probar la arquitectura del sistema.
- Construcción: el objetivo es construir software operativo de forma incremental que cumpla con las necesidades de prioridad más altas de las personas involucradas en el negocio.
- Transición: el objetivo es validar y desplegar el sistema en el entorno de producción.

6.5.5.2.1. Disciplinas

AUP tiene siete disciplinas:

1. Modelado. Entender el negocio de la organización, tratar el dominio del problema e identificar una solución viable para tratar el dominio del problema
2. Implementación. Transformar el modelo en código ejecutable y realizar un nivel básico de pruebas, en particular pruebas unitarias
3. Pruebas. Realizar una evaluación objetiva para asegurar calidad. Esto incluye encontrar defectos, validar que el sistema funciona como fue diseñado y verificar que se cumplen los requisitos
4. Despliegue. Planificar el despliegue del sistema y ejecutar el plan para poner el sistema a disposición de los usuarios finales
5. Gestión de configuración. Gestión de acceso a los artefactos del proyecto. Esto no sólo incluye el seguimiento de las versiones de los artefactos sino también controlar y gestionar los cambios en ellos
6. Gestión de proyecto. Dirección de las actividades que tienen lugar dentro del proyecto. Esto incluye gestionar riesgos, dirigir a las personas y coordinar las personas y sistemas fuera del alcance del proyecto para asegurar que se entrega a tiempo y dentro del presupuesto.
7. Entorno. Soporte del resto del esfuerzo asegurando que el proceso, la orientación (estándares y guías) y las herramientas (software, hardware…) adecuadas están disponibles para el equipo cuando son necesarias.

6.5.5.2.2. Filosofías

AUP se basa en las siguientes filosofías:

1. Los empleados saben lo que están haciendo. La gente no va a leer documentación del proceso detallada, pero quieren algo de orientación a alto nivel y/o formación de vez en cuando. El producto AUP proporciona enlaces a muchos de los detalles pero no fuerza a ellos.
2. Simplicidad. Todo está descrito de forma concisa.
3. Agilidad. AUP se ajusta a los valores y principios de desarrollo de software ágil y la Alianza Ágil
4. Foco en las actividades de alto valor. El foco está en las actividades que realmente cuentan, no en todas las posibles cosas que pudieran pasar en un proyecto.
5. Independencia de herramientas. Se puede usar cualquier conjunto de herramientas. La recomendación es que se usen las herramientas que mejor se adapten al trabajo, que son con frecuencia herramientas simples.
6. Habrá que adaptar AUP para cumplir con las necesidades propias.

6.6. PRÁCTICAS ÁGILES

Son varias las prácticas que se utilizan en el desarrollo rápido. Aunque algunas de ellas se consideran metodologías en sí mismas, son simplemente prácticas usadas en diferentes metodologías.

A continuación vamos a explicar algunas de las más extendidas.

6.6.1. Test Driven Development (TDD)

El desarrollo orientado a pruebas (TDD) es una técnica de desarrollo de software que usa iteraciones de desarrollo cortas basadas en casos de prueba escritos previamente que definen las mejoras deseadas o nuevas funcionalidades. Cada iteración produce el código necesario para pasar la prueba de la iteración. Finalmente, el programador o equipo refactoriza el código para acomodar los cambios. Un concepto clave de TDD es que las pruebas se escriben antes de que se escriba el código para que éste cumpla con las pruebas. Hay que tener en cuenta que TDD es un método de diseño de software, no sólo un método de pruebas.

TDD está relacionado con los primeros conceptos de pruebas de programación de Extreme Programming, en 1999, pero más recientemente se está creando un interés general mayor en sí mismo.

TDD requiere que los desarrolladores creen pruebas unitarias automatizadas para definir los requisitos del código antes de escribir el código en sí mismo. Las pruebas contienen afirmaciones que son verdaderas o falsas. El objetivo es escribir código claro que funcione. Ejecutar las pruebas rápidamente confirma el comportamiento correcto a medida que los

desarrolladores evolucionan y refactorizan el código. Los desarrolladores se ayudan de herramientas para crear y ejecutar automáticamente conjuntos de casos de prueba.

Una ventaja de esta forma de programación es evitar escribir código innecesario. Se intenta escribir el mínimo código posible.

La idea de escribir las pruebas antes que el código tiene dos beneficios principales. Ayuda a asegurar que la aplicación se escribe para poder ser probada, ya que los desarrolladores deben considerar cómo probar la aplicación desde el principio, en vez de preocuparse por ello luego. También asegura que se escriben pruebas para cada característica.

6.6.1.1. Ciclo de desarrollo orientado a pruebas

La siguiente secuencia está basada en el libro Test-Driven Development by Example, que puede considerarse el texto fuente canónico del concepto en su forma moderna.

1. Añadir una prueba

   En el desarrollo orientado a pruebas, cada nueva característica empieza con la escritura de una prueba. Esta prueba deberá fallar inevitablemente porque se escribe antes de que se haya implementado la característica. Para escribir una prueba, el desarrollador debe entender claramente la especificación y los requisitos de la característica. El desarrollador puede llevar a cabo esto a través de casos de uso e historias de usuario que cubran los requisitos y las condiciones de excepción. No hace falta que sea una nueva funcionalidad, puede implicar también una variación o modificación de una prueba existente.

2. Ejecutar las pruebas y comprobar que la última que se ha añadido falla

   Esto valida que las pruebas están funcionando correctamente y que las nuevas pruebas no pasan erróneamente, sin la necesidad de código nuevo.

3. Escribir algo de código, realizar cambios en la implementación

   El siguiente paso es escribir algo de código que haga que se puedan pasar las pruebas. El código escrito en esta etapa no será perfecto y puede, por ejemplo pasar la prueba de una forma poco elegante. Esto es aceptable porque en pasos sucesivos se mejorará y perfeccionará.

   Es importante resaltar que el código escrito sólo se diseña para pasar la prueba; no se debería predecir más funcionalidad.

4. Ejecutar las pruebas automatizadas y ver que tienen éxito

   Si todas las pruebas ahora pasan, el programador puede estar seguro de que el código cumple todos los requisitos probados. Este es un buen punto para empezar el paso final del ciclo.

5. Refactorizar el código para mejorar su diseño

Ahora se puede limpiar el código si es necesario. Volviendo a ejecutar las pruebas, el desarrollador puede estar seguro de que la refactorización no ha dañado ninguna funcionalidad existente.

Repetir el ciclo con una nueva prueba.

6.6.1.2. Ventajas

Entre las ventajas que se desprenden del uso de esta práctica encontramos las siguientes:

- Al escribir primero los casos de prueba, se definen de manera formal los requisitos que se espera que cumpla la aplicación. Los casos de prueba sirven como documentación del sistema.
- Al escribir una prueba unitaria, se piensa en la forma correcta de utilizar un módulo que aún no existe.
- Las pruebas permiten perder el miedo a realizar modificaciones en el código, ya que tras realizar modificaciones se volverán a ejecutar los casos de pruebas para comprobar si se ha cometido algún error.

6.6.1.3. Inconvenientes

- TDD es difícil de usar en situaciones donde hacen falta todas las pruebas funcionales para determinar éxito o fracaso. Ejemplos de esto son interfaces de usuario, programas que trabajan con bases de datos, y algunos que dependen de configuraciones de red específicas.
- El soporte de la gestión es esencial. Sin la creencia de toda la organización de que TDD va a mejorar el producto, la gestión sentirá que se pierde tiempo escribiendo pruebas.
- Las pruebas se han visto históricamente como una posición más baja que los desarrolladores o arquitectos.

6.6.2. Integración continua

La integración continua es un conjunto de prácticas de ingeniería del software que aumentan la velocidad de entrega de software disminuyendo los tiempos de integración (entendiendo por integración la compilación y ejecución de pruebas en todo un proyecto).

La integración continua es un proceso que permite comprobar continuamente que todos los cambios que lleva cada uno de los desarrolladores no producen problemas de integración con el código del resto del equipo. Los entornos de integración continua construyen el software desde el repositorio de fuentes y lo despliegan en un entorno de integración sobre el que realizar pruebas. El concepto de integración continua es que se debe integrar el desarrollo de una forma incremental y continua.

Cuando se embarca en un cambio, un desarrollador coge una copia del código actual en el que trabajar. Cuando el código cambiado se sube al repositorio por otros desarrolladores,

esta copia deja de reflejar el código del repositorio. Cuando el desarrollador sube el código al repositorio debe primero actualizar su código para reflejar los cambios que se han producido en el repositorio desde que cogió su copia. Cuantos más cambios haya en el repositorio, más trabajo debe hacer el desarrollador antes de subir sus propios cambios.

Finalmente, el repositorio se puede convertir tan diferente de la línea base del desarrollador que entran en algo llamado a veces, “infierno de integración”, donde el tiempo que usan para integrar es mayor que el tiempo que le han llevado sus cambios originales. En un caso peor, los cambios que está haciendo el desarrollador pueden tener que ser descartados y el trabajo se ha de rehacer.

6.6.2.1. Prácticas recomendadas

La integración continua en sí misma hace referencia a la práctica de la integración frecuente del código de uno con el código que se va a liberar. El término frecuente es abierto a interpretación, pero con frecuencia se interpreta como “muchas veces al día”.

- Mantener un repositorio de código:

Esta práctica recomienda el uso de un sistema de control de revisiones para el código fuente del proyecto. Todos los artefactos que son necesarios para construir el proyecto se colocan en el repositorio. En esta práctica y en la comunidad de control de revisión, la convención es que el sistema debería ser construido de un checkout nuevo y no debería requerir dependencias adicionales.

- Automatizar la construcción:

El sistema debería ser construible usando un comando único. La automatización de la construcción debería incluir la integración, que con frecuencia incluye despliegue en un entorno similar al de producción. En muchos casos, el script de construcción no sólo compila binarios, sino que también genera documentación, páginas web, estadísticas y distribución.

- Hacer las pruebas propias de la construcción:

Esto toca otro aspecto de mejor práctica, desarrollo orientado a pruebas. Esta es la práctica de escribir una prueba que demuestre la falta de funcionalidad en el sistema y entonces escribir el código que hace que esa prueba se pueda pasar.

Una vez que el código está construido, se deben pasar todas las pruebas para confirmar que se comporta como el desarrollador esperaba que lo hiciera.

- Mantener la construcción rápida:

La construcción ha de ser rápida, de tal manera que si hay un problema con la integración, se identifica rápidamente.

- Probar en un clon del entorno de producción:

Tener un entorno de pruebas puede conducir a fallos en los sistemas probados cuando se despliegan en el entorno de producción, porque el entorno de producción puede diferir del entorno de pruebas en una forma significativa.

- Hacer fácil conseguir los últimos entregables:

  Mantener disponibles las construcciones para las personas involucradas en el negocio y los técnicos de pruebas puede reducir la cantidad de re-trabajo necesaria cuando se reconstruye una característica que no cumplía los requisitos. Adicionalmente, las pruebas tempranas reducen las opciones de que los defectos sobrevivan hasta el despliegue. Encontrar incidencias también de forma temprana, en algunos casos, reduce la cantidad de trabajo necesario para resolverlas.

- Todo el mundo puede ver los resultados de la última construcción.
- Despliegue automático.

6.6.2.2. Ventajas

La integración continua tiene muchas ventajas:

- Reducción del tiempo de integración.
- Cuando las pruebas unitarias fallan, o se descubre un defecto, los desarrolladores pueden revertir el código base a un estado libre de defectos, sin perder tiempo depurando.
- Los problemas de integración se detectan y arreglan continuamente, no hábitos de último minuto antes de la fecha de liberación.
- Avisos tempranos de código no operativo/incompatible.
- Avisos tempranos de cambios conflictivos.
- Pruebas unitarias inmediatas de todos los cambios.
- Disponibilidad constante de una construcción actual para propósitos de pruebas, demo o liberación.
- El impacto inmediato de subir código incompleto o no operativo actúa como incentivo para los desarrolladores para aprender a trabajar de forma más incremental con ciclos de retroalimentación más cortos.

6.6.2.3. Inconvenientes

Las desventajas que se pueden observar con el uso de esta práctica son las siguientes:

- Sobrecarga por el mantenimiento del sistema.
- Necesidad potencial de un servidor dedicado a la construcción del software.
- El impacto inmediato al subir código erróneo provoca que los desarrolladores no hagan tantos commits como sería conveniente como copia de seguridad.

6.6.3. Pair programming

La programación por pares es una técnica de desarrollo de software en la que dos programadores trabajan juntos en el mismo ordenador. Uno teclea el código mientras que el otro revisa cada línea del código a medida que el primero lo va escribiendo. La persona que teclea es el denominado “driver” (conductor-controlador). La persona que revisa el código recibe el nombre de “observer” o “navigator”. Los dos programadores cambian los roles con frecuencia (posiblemente cada 30 minutos).

Mientras revisa, el observador también considera la dirección del trabajo, generando ideas para mejoras y problemas futuros posibles a arreglar. Esto libera al driver para centrar toda su atención en los aspectos “tácticos” de completar su tarea actual, usando al observador como una red y guía segura.

6.6.3.1. Ventajas

Entre las ventajas que puede ofrecer una práctica como la programación por pares encontramos:

- Calidad de diseño: programas más cortos, diseños mejores, pocos defectos. El código del programa debe ser legible para ambos compañeros, no sólo para el driver, para poder ser chequeado. Las parejas típicamente consideran más alternativas de diseño que los programadores que trabajan solos, y llegan a diseños más simples, más fáciles de mantener, así como encuentran defectos de diseño muy pronto.
- Coste reducido del desarrollo: siendo los defectos una parte particularmente cara del desarrollo de software, especialmente si se encuentran tarde en el proceso de desarrollo, la gran reducción en la tasa de defectos debido a la programación por pares puede reducir significativamente los costes del desarrollo de software.
- Aprendizaje y formación: el conocimiento pasa fácilmente entre programadores: comparten conocimiento del sistema, y aprenden técnicas de programación unos de otro a medida que trabajan.
- Superar problemas difíciles: los pares a menudo encuentran que problemas “imposibles” se convierten en más sencillos o incluso más rápidos, o al menos posibles de resolver cuando trabajan juntos.
- Moral mejorada: los programadores informan de una mayor alegría en su trabajo y mayor confianza en que su trabajo es correcto.
- Disminución del riesgo de gestión: ya que el conocimiento del sistema se comparte entre varios programadores, hay menos riesgo para gestionar si un programador abandona el equipo.
- Incremento de la disciplina y mejor gestión del tiempo: es menos probable que los programadores pierdan tiempo navegando en la web o en mails personales u otras violaciones de disciplina cuando trabajan con un compañero.

- Flujo elástico: la programación por pares conduce a un diferente tipo de flujo que la programación individual. Es más rápido y más fuerte o elástico a las interrupciones ya que cuando uno trata la interrupción el otro puede seguir trabajando.
- Menos interrupciones: la gente es más reacia a interrumpir a una pareja que a una persona que trabaja sola.
- Menos puestos de trabajo requeridos.

6.6.3.2. Inconvenientes

Pero esta técnica también genera una serie de desventajas o factores que pueden afectar de forma negativa:

- Preferencia de trabajo: muchos ingenieros prefieren trabajar solos.
- Intimidación: un desarrollador con menos experiencia puede sentirse intimidado cuando su compañero sea un desarrollador con más experiencia y como resultado puede que participe menos.
- Coste de tutoría: los desarrolladores con experiencia pueden encontrar tedioso enseñar a un desarrollador con menos experiencia. Desarrolladores con experiencia que trabajen solos pueden ser capaces de producir código limpio y exacto desde el principio, y los beneficios de los pares pueden no valer el coste que supone un desarrollador adicional en algunas situaciones. Esto puede aplicar especialmente cuando se producen las partes más triviales del sistema.
- Egos y conflictos potenciales: conflictos personales que pueden resultar en que uno o los dos desarrolladores se sientan incómodos. Las diferencias en el estilo de codificación pueden llevar a conflicto.
- Hábitos personales molestos: las personas se pueden sentir molestas por otros miembros del equipo debido a objeciones con algunos de sus hábitos.
- Coste: ya que hay dos personas a las que pagar, el beneficio de la programación por pares no empieza hasta que la eficiencia es por lo menos el doble.

6.7. CRÍTICAS AL DESARROLLO ÁGIL

Los rumores iniciales y los principios controvertidos de Extrene programming, tales como programación por pares e integración continua, han atraído críticas particulares, como las de McBreen y Boehm y Turner. Muchas de las críticas, sin embargo, son consideradas por parte de los partidarios del desarrollo ágil como malentendidos del desarrollo ágil.

Las críticas incluyen:

- Con frecuencia se usa como medio de sacar dinero al cliente a través de la falta de definición de un entregable.
- Falta de estructura y documentación necesaria.
- Sólo funciona con desarrolladores experimentados.

- Incorpora diseño de software insuficiente.
- Requiere encuentros a intervalos frecuentes con un enorme coste para los clientes.
- Requiere demasiado cambio cultural para adaptarlo.
- Puede conducir a negociaciones contractuales más difíciles.
- Puede ser muy ineficiente, si los requisitos de un área de código cambian durante varias iteraciones, se puede necesitar hacer la misma programación varias veces. Mientras que si se tiene que seguir un plan, un área de código individual se supone que sólo se va a escribir una vez.
- Imposible desarrollar estimaciones realistas del esfuerzo necesario para proporcionar un presupuesto, porque al principio del proyecto nadie sabe el alcance o los requisitos enteros.
- Puede aumentar el riesgo de cambios del alcance, debido a la falta de documentación de requisitos detallada.
- El desarrollo ágil está orientado a características, los atributos de calidad no funcionales son difícil de plasmar como historias de usuario.

7. CONCLUSIONES

No existe una metodología universal para hacer frente con éxito a cualquier proyecto de desarrollo de software. Toda metodología debe ser adaptada al contexto del proyecto (recursos técnicos y humano, tiempo de desarrollo, tipo de sistema, etc.). Históricamente, las metodologías tradicionales han intentado abordar la mayor cantidad de situaciones de contexto del proyecto, exigiendo un esfuerzo considerable para ser adaptadas, sobre todo en proyectos pequeños y con requisitos muy cambiantes. Las metodologías ágiles ofrecen una solución casi a medida para una gran cantidad de proyectos que tienen estas características. Una de las cualidades más destacables en una metodología ágil es su sencillez, tanto en su aprendizaje como en su aplicación, reduciéndose así los costes de implantación en un equipo de desarrollo. Esto ha llevado hacia un interés creciente en las metodologías ágiles. Sin embargo, hay que tener presente una serie de inconvenientes y restricciones para su aplicación, tales como: están dirigidas a equipos pequeños o medianos, el entorno físico debe ser un ambiente que permita la comunicación y colaboración entre todos los miembros del equipo durante todo el tiempo, cualquier resistencia del cliente o del equipo de desarrollo hacia las prácticas y principios puede llevar al proceso al fracaso, el uso de tecnologías que no tengan un ciclo rápido de realimentación o que no soporten fácilmente el cambio.

En la ingeniería del software, el “post-agilism” es un movimiento informal de partidarios que prefieren evitar ser obligados por lo que ellos consideran “Dogma Ágil”.

Algunos han argumentado que el significado de ágil es ambiguo y se ha aplicado de forma inapropiada para un amplio rango de enfoques como Six Sigma y CMMi®.

Los partidarios también argumentan que los métodos orientados a procesos, especialmente los métodos que confían en resultados repetibles y que reducen incrementalmente la pérdida de tiempo y la variación de los procesos como Six Sigma, tienen una tendencia a limitar la capacidad de adaptación de una organización, haciéndolas menos capaces de responder al cambio discontinuo, p.ej., menos ágil. Se propone que “el desarrollo ágil” necesita ser entendido de forma adecuada y aplicado de forma apropiada a cualquier contexto específico. Algunos “agilistas” están de acuerdo con esta posición, promoviendo el concepto de los métodos ágiles como un conjunto de herramientas que deberían estar disponibles para su uso en situaciones apropiadas, no como un método que debería valer en todas las situaciones.

8. GLOSARIO

- **Aseguramiento de la calidad del software (SQA)**: es un conjunto de actividades planificadas y ejecutadas sistemáticamente que apunta a asegurar que el software que se está construyendo es de alta calidad.
- **Ciclo de desarrollo de software**: tiempo entre la iniciación del desarrollo del software y la entrega del mismo.
- **Marco de proceso común**: es un proceso genérico que define las actividades requeridas para ejecutar un proyecto software. Puede ser adaptado para un proyecto específico.
- **Métodos de ingeniería del software**: los métodos de ingeniería del software proporcionan el "cómo" para la construcción del software. Abarcan las tareas técnicas requeridas para realizar y documentar el análisis de requisitos, el diseño, la construcción de programas, las pruebas y el mantenimiento.
- **Modelos de proceso ágiles**: los modelos de proceso ágiles son enfoques de desarrollo donde los requisitos del cliente se cumplen temprano en el ciclo de vida de desarrollo del software a través de continuas entregas de software. En estos modelos, los cambios en los requisitos son bienvenidos.
- **Modelo de proceso de software**: un modelo de proceso de software es una estrategia abstracta para la construcción y mantenimiento de un producto software. Ayuda a los jefes de proyecto en la planificación y ejecución de un proyecto. También se le llama modelo de ciclo de vida o paradigma de ingeniería del software.
- **Modelo de proceso incremental**: el modelo de proceso es un enfoque en el que el producto se entrega al cliente incrementalmente sobre un periodo de tiempo planificado.
- **Proceso**: secuencia de pasos para realizar alguna actividad e incluye la descripción de entradas, salidas, procedimientos, herramientas, responsabilidades y criterios de salida.
- **Productos de trabajo**: elementos que se crean durante el curso del proceso software, tales como documentos, software y manuales.
- **Refactorización**: es una actividad de examinar la estructura del software para eliminar redundancias, funcionalidad no utilizada y rejuvenecer objetos obsoletos mientras se mantiene el comportamiento observable. Esto asegura que la estructura del software permanece simple y fácil de modificar.

9. ACRÓNIMOS

- **CAD**: Computer-Aided Design (Diseño asistido por computador)
- **CASE**: Computer Aided Software Engineeirng (Ingeniería de software asistida por computador)
- **CPF**: Common Process Framework (Marco de proceso común)
- **DSDM**: Dynamic Systems Development Method (Método de desarrollo de sistemas dinámico)
- **IEC**: International Electrotechnical Commission
- **IEEE**: Institute of Electrical and Electronics Engineers
- **ISO**: International Organization for Standardization
- **UP**: Unified Process (Proceso Unificado)
- **RAD**: Rapid Application Development (Desarrollo Rápido de Aplicaciones)
- **RUP**: Rational Unified Process (Proceso Unificado Rational)
- **SDLC**: Software Development Life Cycle (Ciclo de vida de desarrollo de software)
- **TDD**: Test Driven Development (Desarrollo orientado a pruebas)
- **UML**: Unified Modeling Language
- **XP**: Extreme Programming (Programación extrema)

10. REFERENCIAS

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*"

Sitios web

Agile Spain www.agile-spain.com

Alianza ágil www.agilealliance.org

IEEE www.ieee.org/portal/site

Manifiesto ágil www.agilemanifesto.org

Sitio web de la Organización Internacional para la Estandarización www.iso.org

Swebok www.swebok.org