GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, TURISMO Y COMERCIO
plan avanza2
INTECO
Instituto Nacional de Tecnologías de la Comunicación

CURSO DE DESARROLLO ÁGIL

Laboratorio Nacional de Calidad del Software

NOTA DE EDICIÓN

Este curso ha sido desarrollado por el Laboratorio Nacional de Calidad del Software de INTECO. Esta primera versión ha sido editada en Junio del 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
- PMBOK® es una marca registrada por el Project Management Institute, Inc.

Todas las demás marcas registradas que se mencionan, usan o citan en el presente curso 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 todos los cursos editados 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

Escenario de apertura

COMPASS S.A. es una empresa de desarrollo de software que en los últimos años ha ido adquiriendo un número de pequeñas empresas.

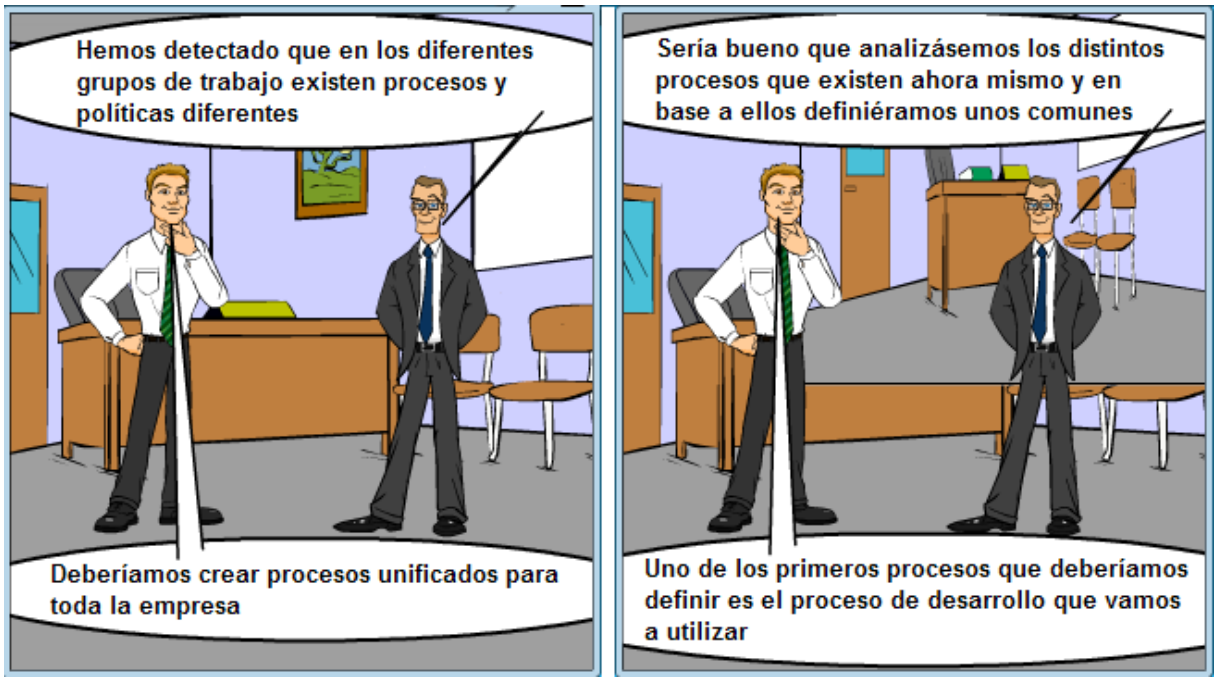

Figura 1. Escenario de apertura I

En parte de las empresas que ahora forman parte de COMPASS se estaba usando un modelo en cascada como modelo de desarrollo, pero se ha observado que con este modelo había muchas cosas que fallaban.

Introducción

El software juega un papel significativo en la vida de las personas. Se puede usar tanto en una aplicación en un ordenador personal como parte embebida de un robot industrial.

Desde que se empezó con el desarrollo de software han ido surgiendo numerosos métodos, paradigmas y modelos de proceso para manejar los esfuerzos complejos del desarrollo. Algunos de los métodos de desarrollo se han convertido en métodos orientados a documentación o con la expectativa de que los desarrolladores sigan ciertos procesos. A estos métodos se les suele conocer como **métodos tradicionales o pesados**.

El desarrollo de software ha estado plagado de problemas. Afortunadamente, al mismo tiempo se están haciendo continuamente innovaciones en técnicas de programación para entregar software de calidad que cumpla los requisitos de los clientes dentro del presupuesto y la planificación.

Probablemente el cambio más notable en los últimos años en el proceso de software ha sido la aparición de la palabra ágil. Se habla de **métodos de software ágiles**, de cómo introducir agilidad en un equipo de desarrollo, o de cómo resistir a la tormenta inminente de “agilistas” decididos a cambiar prácticas bien establecidas.

Este nuevo movimiento creció de los esfuerzos de varias personas que trataban con procesos software en los 90. La mayoría de las ideas no eran nuevas, es más, mucha gente creía que se había construido software exitoso de esta forma desde hacía tiempo. Había, sin embargo, una vista de que esas ideas se habían contenido y no se habían tratado de forma suficientemente seria, particularmente por la gente interesada en procesos de software.

Muchas personas se preguntan qué es el desarrollo ágil y siempre consiguen diferentes definiciones dependiendo de a quién pregunten. Mucha gente dirá correctamente que el desarrollo de software ágil cumple con los valores y principios del Manifiesto Ágil. El desarrollo de software ágil disciplinado es en enfoque iterativo e incremental (evolutivo) de desarrollo de software que se realiza de una forma colaborativa mediante una organización de los equipos propia dentro de un marco de trabajo de gobierno efectivo con la ceremonia justa que produce software de alta calidad con un coste efectivo y en el tiempo apropiado que cumple con las necesidades cambiantes de las personas involucradas en el negocio.

Modelos iterativos e incrementales

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* o 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, que se irán viendo a lo largo del curso.

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 2. 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.

Figura 3. Elección de un modelo iterativo e incremental

La idea básica

La idea básica detrás de la mejora iterativa es desarrollar un sistema software de forma incremental, 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.

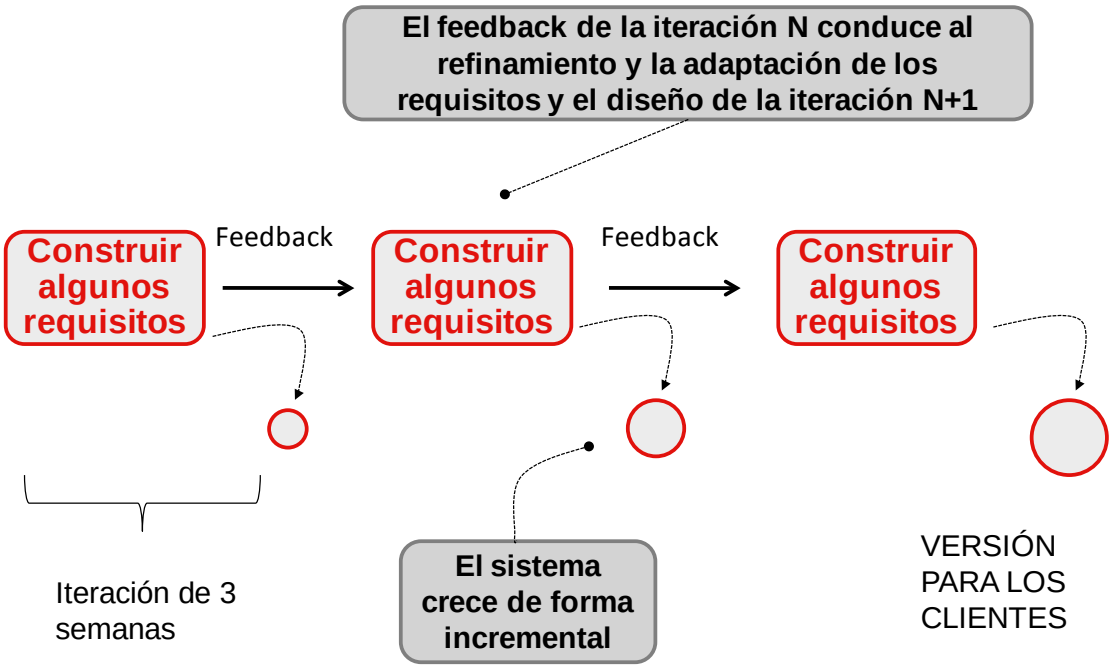

Figura 4. Desarrollo iterativo e incremental

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 interactivo. 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 técnicos de pruebas.

Debilidades en los modelos

Entre las debilidades que se han encontrado en este tipo de modelos aparecen las que se mencionan a continuación:

- 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.

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:

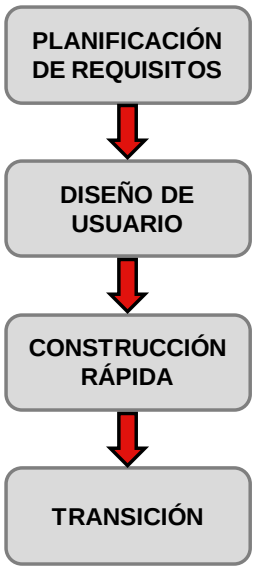

Figura 5. Flujo de proceso de RAD

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.

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 quipo solía diseñar software; tenía un modelo orientado a objetos subyacente, usando UML (*Unified Modeling Language*)

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.

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

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.

Desarrollo ágil

El desarrollo de software ágil hace referencia a un grupo de metodologías de desarrollo de software que se basan en principios y valores similares recogidos en el Manifiesto Ágil. Generalmente promocionan:

- 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.

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 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.

Entre los métodos ágiles tempranos destacan *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.

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 para tratar el tema de la unificación de sus metodologías. A raíz de esto 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 La Alianza Ágil, 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”.

El Manifiesto para el desarrollo de software ágil trata de poner en duda la situación del momento respecto al desarrollo de software, ya que fue creado para contrarrestar los métodos pesados orientados a procesos y a documentación e intenta sugerir nuevos enfoques de desarrollo de software.

El Manifiesto Ágil es una colección de valores y principios que se pueden encontrar en la mayoría de los métodos ágiles. La Alianza Ágil describe sus intenciones de la siguiente forma:

> *El movimiento ágil no es un anti-método, de hecho, muchos de nosotros queremos restablecer la credibilidad de la palabra método. Queremos restablecer un equilibrio. Estamos de acuerdo con modelar pero no para guardar algunos diagramas en un repositorio corporativo polvoriento. Estamos de acuerdo con la documentación pero no con cientos de páginas que no se*

mantienen y rara vez se usan. Nosotros planificamos pero reconocemos los límites de planificar en un entorno turbulento.

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.

  Los métodos tradicionales con frecuencia intentar desarrollar un proceso que distingue los recursos humanos como roles, tales como jefe de proyecto, analista o programador. Tales procesos enfatizan los roles, no los individuos llevando a cabo tareas relacionadas con un rol particular. Con frecuencia los managers quieren pensar que las personas son intercambiables. Los managers creen que para mantener los proyectos desarrollándose un rol se puede cubrir por cualquiera si un individuo decide abandonar el proyecto. En algunos entornos de producción este pensamiento puede ser conveniente, pero en trabajo creativo el individualismo y la competencia personal se resaltan y por lo tanto es difícil reemplazar a un individuo.

  Los métodos ágiles rechazan tales suposiciones de que la gente involucrada en el desarrollo de software son partes reemplazables. Aunque las descripciones de proceso y los gráficos de organización son necesarios para conseguir que el proyecto arranque, la Alianza Ágil quiere hacer énfasis en los individuos más que en los roles y fomentan la interacción entre los individuos.

  Este valor también quiere abandonar el uso de procesos estrictos. Sin embargo, los procesos pueden ser beneficiales hasta cierto punto, por lo que se debería evitar un abandono total de los mismos.

- **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.

De acuerdo con la Alianza Ágil, los documentos que contienen requisitos, análisis o diseño pueden ser muy útiles para guiar el trabajo de los desarrolladores y ayudan a predecir el futuro. Pero código que funciona, que ha sido probado y depurado revela información sobre el equipo de desarrollo, el proceso de desarrollo y la naturaleza de los problemas a solucionar. La Alianza Ágil afirma que la ejecución de un programa es la única medida fiable de la velocidad y las deficiencias del equipo y vislumbra lo que el equipo debería construir en realidad. En ocasiones los métodos ágiles aparecen menos orientados a planes que lo que de verdad son. Los documentos y planes se usan en métodos ágiles hasta cierto punto.

- **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.

  Este valor enfatiza las relaciones cercanas entre el equipo de desarrollo de software y el cliente. El supuesto básico detrás de este valor es la satisfacción del cliente en general, que es el driver principal en el desarrollo de software ágil.

  La colaboración y dedicación del cliente son críticas para todos los proyectos de desarrollo de software.

- **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.

  Los requisitos de un software nuevo no se van a conocer completamente hasta que los usuarios lo hayan usado. La incertidumbre es inherente e inevitable en los procesos y productos de desarrollo de software y no es posible especificar completamente un sistema. Los requisitos cambian constantemente a causa de la incertidumbre y la naturaleza interactiva del software y a causa de los negocios y los entornos tecnológicos fluctuantes. Estos cambios deberían tenerse en cuenta en el

desarrollo de software. La Alianza Ágil admite que los planes son útiles y que la planificación está realmente incluida en los métodos ágiles que también tienen mecanismos para tratar con los requisitos cambiantes. Sin embargo, en vez de seguir un plan de forma rigurosa, los equipos de desarrollo deberían reflejar de forma constante en el plan la situación actual y cambiar de forma flexible de acuerdo a ella.

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.

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 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 que no hay que hacer, 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.

Características

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.

Figura 6. Métodos ágiles

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 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.

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 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).

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.

Comparación 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 desarrollo ágil tiene poco en común con el modelo en cascada.

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 puede 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 (método de desarrollo ágil que se explicará en apartados sucesivos) 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.

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.

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 se muestran algunos de los elementos que pueden caracterizar a las 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 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

Empezar a usar un método ágil

El uso de un método ágil no es para todo el mundo. Hay un número de aspectos que hay que tener en mente si se decide seguir este camino. Por otro lado, estas metodologías se consideran ampliamente aplicables.

Para alguien nuevo en métodos ágiles, la pregunta es por dónde empezar. Como con cualquier nueva tecnología o proceso, es necesario hacer nuestra propia evaluación de ello. Esto permite ver cómo encaja en un entorno.

El primer paso es encontrar proyectos adecuados para probar los métodos ágiles. Debido a que los métodos ágiles están fundamentalmente orientados a las personas, es esencial que se empiece con un equipo que quiera probar y trabajar de forma ágil.

Es valioso también tener clientes (aquellos que necesitan el software) que quieran trabajar de forma colaborativa. Si los clientes no colaboran, no se verán las ventajas completas de un proceso adaptativo.

Mucha gente afirma que los métodos ágiles no se pueden usar en proyectos grandes. Se ha tenido éxito con proyectos ágiles con unas 100 personas y múltiples continentes. A pesar de ello, se podría empezar con algo más pequeño. Los proyectos más grandes son indudablemente más complicados, así que es mejor aprender con proyectos de tamaño más manejable.

Algunas personas aconsejan escoger un proyecto con poco impacto en el negocio para empezar, de tal forma que si algo va mal haya menos daños. Sin embargo, un proyecto que no es importante puede ser una prueba pobre ya que nadie se preocupa mucho del resultado.

Quizás una de las cosas más importantes que se pueden hacer es encontrar a alguien con experiencia en métodos ágiles para ayudar. Y una vez que se haya encontrado un buen mentor, seguir su consejo.

¿Por qué no se debería usar un método ágil? Si las personas involucradas no están interesadas en el tipo de colaboración intensa que requiere el trabajo ágil, va a ser una lucha importante conseguir que trabajen así.

Escenario

La empresa COMPASS quiere lleva a cabo su idea de implantar un modelo iterativo e incremental, en este momento están decidiendo cuál de ellos escogen.

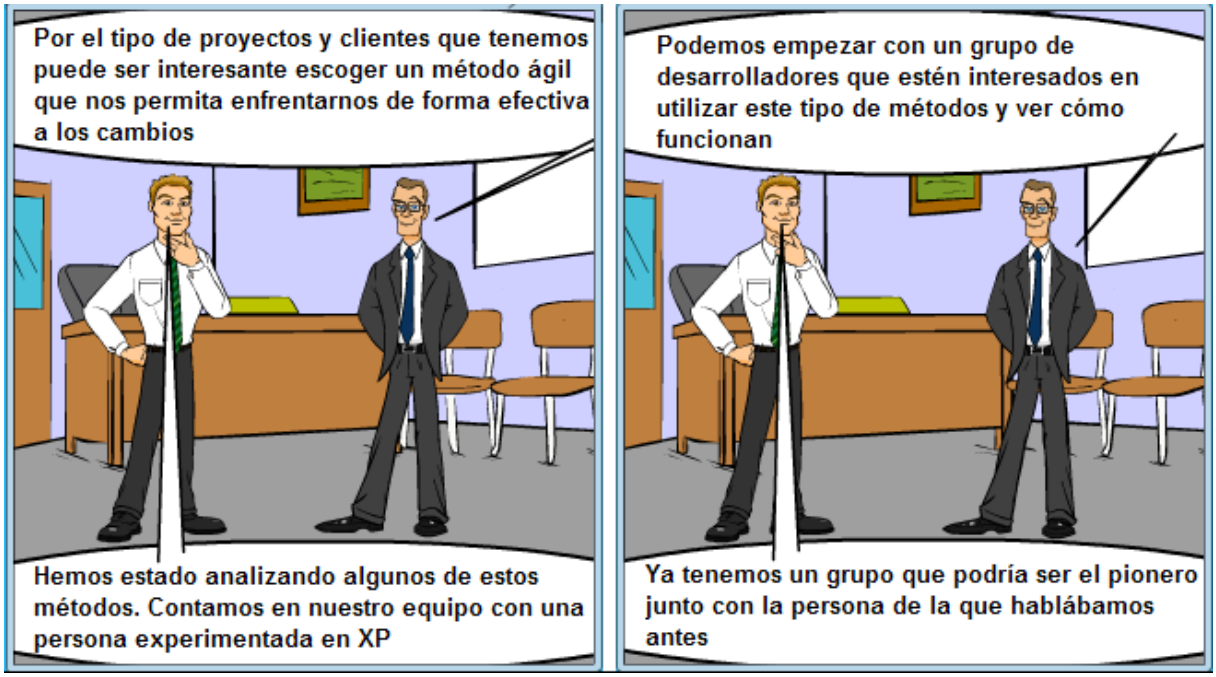

Figura 7. Elección de un modelo iterativo e incremental I

Figura 8. Elección de un modelo iterativo e incremental I

Es importante contar con un equipo motivado y con un cliente colaborador.

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 preceptivos. 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 (método ágil que se explicará en este apartado) hace la necesidad de la adaptación de métodos explícita. Una de las ideas fundamentales de XP es que no un proceso 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 también considerada en varias ocasiones.

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.

Existen herramientas de gestión de proyectos dirigidas para el desarrollo ágil. Están diseñadas para planificar, hacer seguimiento, analizar e integrar el 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 para medir objetivamente el éxito del proyecto.

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.

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.

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 nombres que actúen como vocabulario para hablar sobre el dominio del problema, ayudando a

la nomenclatura de clases y métodos del sistema). Las metáforas se usan para dar una visión general y un entendimiento común del sistema y sus funciones y de cómo se debería construir. Ayudan a conseguir una visión común y a transformar el conocimiento tácito en conocimiento explícito.

- 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 con más 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 con más 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 de 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.

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 cuanto 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 lo que es más importante. Pueden derivarse problemas de tener ciclos muy largos, como meses de trabajo inútiles 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.

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 necesidad 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 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.

Escenario

El gerente de la organización se reúne con parte del esquipo de desarrollo que está llevando a cabo el piloto de utilización de una método ágil, en este caso XP.

Figura 9. Beneficios XP I

Figura 10. Beneficios XP II

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.

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 los 90, 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 Development with Scrum*”.

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 él 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 organización propia 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.

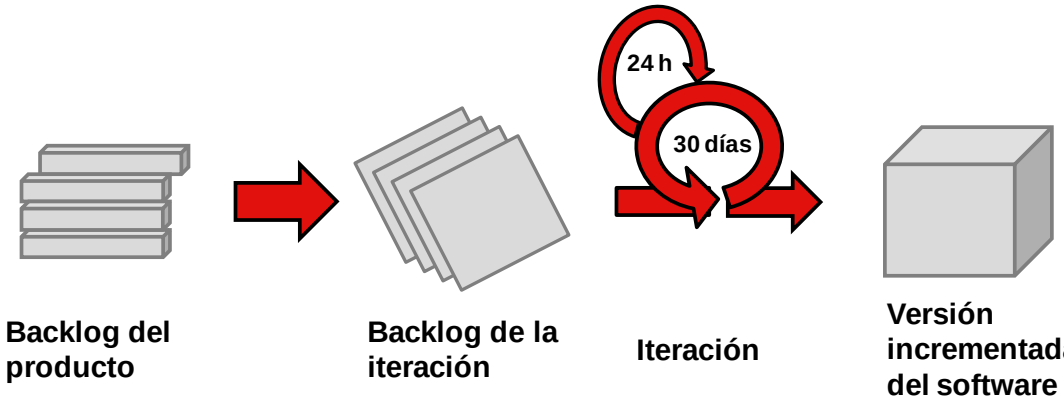

Figura 11. Flujo de proceso de SCRUM

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 debajo de la alfombra. Nadie es penalizado por reconocer o describir un problema imprevisto.

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.

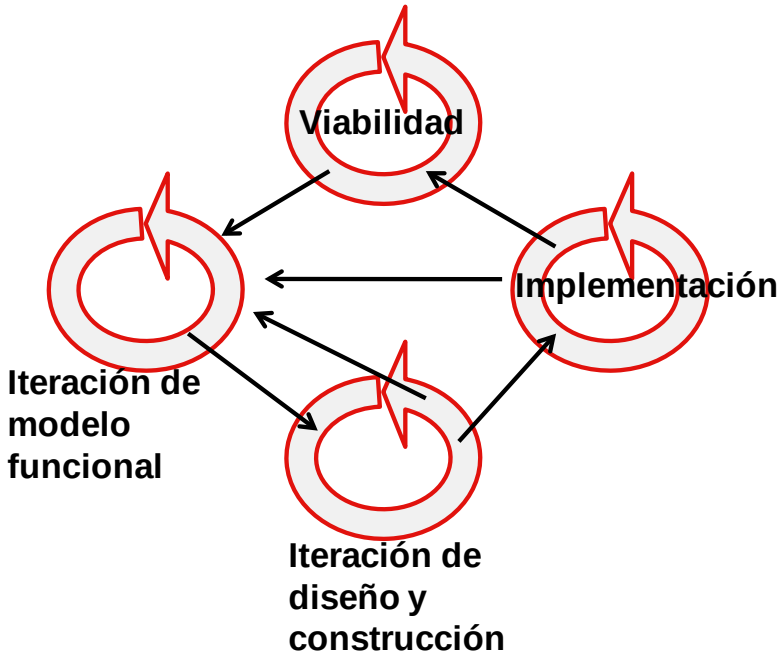

Figura 12. Ciclo de desarrollo de DSDM

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.

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 ha completarse hasta que empiece el siguiente paso. Esto permite que la nueva iteración del proyecto comience sin retrasos innecesarios. 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.

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 la motivación se mantienen a lo largo del proyecto.
- El compromiso de la gestión de asegurar la participación del 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 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.

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.

Otros métodos ágiles

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

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.

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.

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.

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:

- Desarrollo orientado a pruebas (TDD)
- Integración continua
- *Pair programming*

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.

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.

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.

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.

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.

Pero hay que tener en cuenta que el repositorio se puede convertir en algo tan diferente a la línea base del desarrollador que puede caer en lo que se suele llamar, “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.

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 a menudo 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 conlleva un 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.
- Facilitar 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.

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 son 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.

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.

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.

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, etc. 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 disminución de 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 otros 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.

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.

Críticas al desarrollo ágil

Los rumores iniciales y los principios controvertidos de *Extreme 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 difíciles de plasmar como historias de usuario.

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.

Tabla 1. Comparativa entre metodologías tradicionales y desarrollo ágil

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

Escenario de clausura

EL equipo de desarrollo que llevó a cabo el piloto obtuvo grandes beneficios con el uso de XP como modelo de desarrollo.

Figura 13. Escenario de clausura I

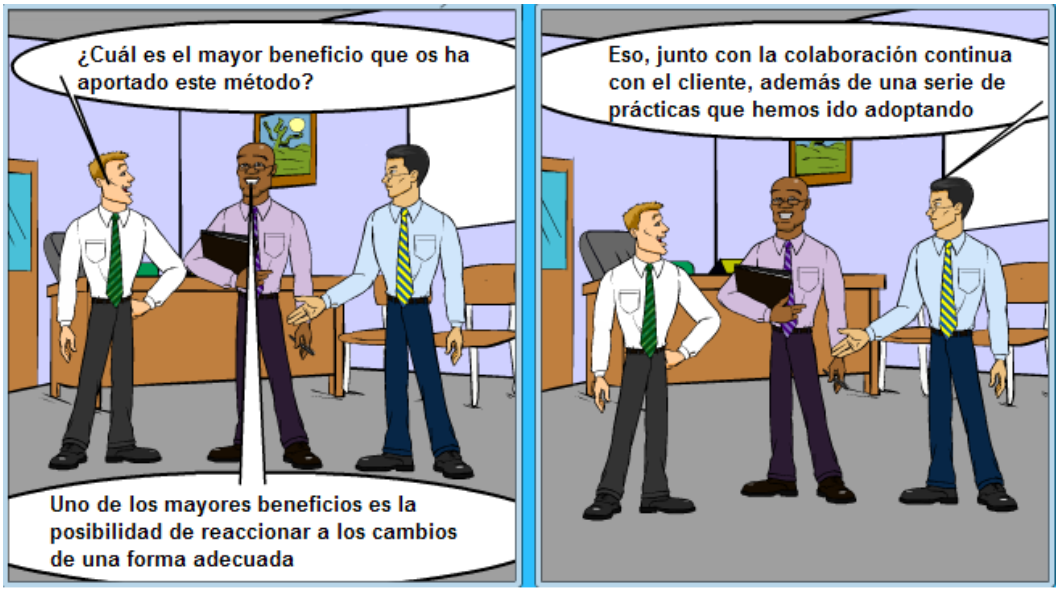

Figura 14. Escenario de clausura II

El gerente de COMPASS felicita a todo el equipo por el buen trabajo realizado y le reporta al director el resultado exitoso del uso de este método de desarrollo.

Enlaces

Comunidad sobre métodos ágiles en lengua castellana, Agile Spain: www.agile-spain.com

Portal de la Alianza ágil: www.agilealliance.org

Portal de la Alianza de Scrum: http://www.scrumalliance.org/

Portal del Consorcio de DSDM: http://www.dsdm.org/

Contenido del Manifiesto ágil: www.agilemanifesto.org

Portal con información acerca de Extreme Programming: http://www.extremeprogramming.org/

Información de RUP:http://www-01.ibm.com/software/awdtools/rup/

Glosario

- **Ciclo de vida:** conjunto de etapas que comprenden todas las actividades, desde el momento en que surge la idea de crear un nuevo producto software, hasta aquel en que el producto deja definitivamente de ser utilizado por el último de sus usuarios
- **Escalabilidad**: capacidad de un software o de un hardware de crecer, adaptándose a nuevos requisitos conforme cambian las necesidades del negocio
- **Framework**: conjunto de APIs y herramientas destinadas a la construcción de un determinado tipo de aplicaciones de manera generalista
- **Proceso**: secuencia de pasos para realizar alguna actividad e incluye la descripción de entradas, salidas, procedimientos, herramientas, responsabilidades y criterios de salida
- **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.