Marcos de trabajo ágiles

Renato Flores, 201709244

25 de octubre de 2020

1. Feature Driven Development (FDD)

Es un método de desarrollo de ciclos cortos que se concentra en la fase de diseño y construcción. En la primera fase, el modelo global de dominio es elaborado por expertos del dominio y desarrolladores; el modelo de dominio consiste en diagramas de clases con clases, relaciones, métodos y atributos. Los métodos no reflejan conveniencias de programación sino rasgos funcionales.

1.1. Principios

- Se requiere un sistema para construir sistemas si se pretende escalar a proyectos grandes.
- Un proceso simple y bien definido trabaja mejor.
- Los pasos de un proceso deben ser logicos y su merito inmediatamenteobvio para cada miembro del equipo.
- Vanagloriarse del proceso puede impedir el trabajo real.
- Los buenos procesos van hasta el fondo del asunto, de modo que los miembros del equipo se puedan concentrar en los resultados.
- Los ciclos cortos, iterativos, orientados por rasgos (features) son mejores.

1.2. Fases

1.2.1. Desarrollo de un modelo general

Cuando comienza este desarrollo, los expertos de dominio ya están al tanto de la visión, el contexto y los requerimientos del sistema a construir. A esta altura se espera que existan requerimientos tales como casos de uso o especificaciones funcionales. FDD, sin embargo, no cubre este aspecto. Los expertos de dominio presentan un ensayo (walkthrough) en el que los miembros del equipo y el arquitecto principal se informan de la descripción de alto nivel del sistema. El dominio general se subdivide en áreas más específicas y se define un ensayo más detallado para cada uno de los miembros del dominio. Luego de cada ensayo, un equipo de desarrollo trabaja en pequeños grupos para producir modelos de objeto de cada área de dominio. Simultáneamente, se construye un gran modelo general para todo el sistema.

1.2.2. Construcción de la lista de rasgos

Los ensayos, modelos de objeto y documentación de requerimientos proporcionan la base para construir una amplia lista de rasgos. Los rasgos son pequeños ítems útiles a los ojos del cliente. Son similares a las tarjetas de historias de XP y se escriben en un lenguaje que todas las partes puedan entender. Las funciones se agrupan conforme a diversas actividades en áreas de dominio específicas. La lista de rasgos es revisada por los usuarios y patrocinadores para asegurar su validez y exhaustividad. Los rasgos que requieran más de diez días se descomponen en otros más pequeños.

1.2.3. Planeamiento por rasgo

Incluye la creación de un plan de alto nivel, en el que los conjuntos de rasgos se ponen en secuencia conforme a su prioridad y dependencia, y se asigna a los programadores jefes. Las listas se priorizan en secciones que se llaman paquetes de diseño. Luego se asignan las clases definidas en la selección del modelo general a programadores individuales, o sea propietarios de clases. Se pone fecha para los conjuntos de rasgos.

1.2.4. Diseño por rasgo y construcción por rasgo

Se selecciona un pequeño conjunto de rasgos del conjunto y los propietarios de clases seleccionan los correspondientes equipos dispuestos por rasgos. Se procede luego iterativamente hasta que se producen los rasgos seleccionados. Una iteración puede tomar de unos pocos días a un máximo de dos semanas. Puede haber varios grupos trabajando en paralelo. El proceso iterativo incluye inspección de diseño, codificación, prueba de unidad, integración e inspección de código. Luego de una iteración exitosa, los rasgos completos se promueven al build principal. Este proceso puede demorar una o dos semanas en implementarse.

2. Extreme Programming(XP)

2.1. Valores

- Comunicación
- Simplicidad
- Feedback
- Coraje

2.2. Fases

2.2.1. Juego de Planeamiento

Busca determinar rápidamente el alcance de la versión siguiente, combinando prioridades de negocio definidas por el cliente con las estimaciones técnicas de los programadores. Éstos estiman el esfuerzo necesario para implementar las historias del cliente y éste decide sobre el alcance y la agenda de las entregas. Las historias se escriben en pequeñas fichas, que algunas veces se tiran. Cuando esto sucede, lo único restante que se parece a un requerimiento es una multitud de pruebas automatizadas, las pruebas de aceptación.

2.2.2. Entregas pequeñas y frecuentes

Se "produccioniza" un pequeño sistema rápidamente, al menos uno cada dos o tres meses. Pueden liberarse nuevas versiones diariamente (como es práctica en Microsoft), pero al menos se debe liberar una cada mes. Se agregan pocos rasgos cada vez.

2.2.3. Metáforas del sistema

El sistema de define a través de una metáfora o un conjunto de metáforas, una "historia compartida" por clientes, managers y programadores que orienta todo el sistema describiendo como funciona. Una metáfora puede interpretarse como una arquitectura simplificada. La concepción de metáfora que se aplica en XP deriva de los estudios de Lakoff y Johnson, bien conocidos en lingüística y psicología cognitiva.

2.2.4. Diseño simple

El énfasis se deposita en diseñar la solución más simple susceptible de implementarse en el momento. Se eliminan complejidades innecesarias y código extra, y se define la menor cantidad de clases posible. No debe duplicarse código. En un oxímoron deliberado, se urge a "decir todo una vez y una sola vez". Nadie en XP llega a prescribir que no haya diseño concreto, pero el diseño se limita a algunas tarjetas elaboradas en sesiones de diseño de 10 a 30 minutos. Esta es la práctica donde se impone el minimalismo de YAGNI: no implementar nada que no se necesite ahora; o bien, nunca implementar algo que vaya a necesitarse más adelante; minimizar diagramas y documentos.

2.2.5. Prueba continua

El desarrollo está orientado por las pruebas. Los clientes ayudan a escribir las pruebas funcionales antes que se escriba el código. Esto es test-driven development. El propósito del código real no es cumplir un requerimiento, sino pasar las pruebas. Las pruebas y el código son escritas por el mismo programador, pero la prueba debería realizarse sin intervención humana, y es a todo o nada. Hay dos clases de prueba: la prueba unitaria, que verifica una sola clase, o un pequeño conjunto de clases; la prueba de aceptación verifica todo el sistema, o una gran parte.

2.2.6. Refactorízación continua

Se refactoriza el sistema eliminando duplicación, mejorando la comunicación y agregando flexibilidad sin cambiar la funcionalidad. El proceso consiste en una serie de pequeñas transformaciones que modifican la estructura interna preservando su conducta aparente. La práctica también se conoce como Mejora Continua de Código o Refactorízación implacable. Se lo ha parafraseado diciendo: "Si funciona bien, arréglelo de todos modos" Se recomiendan herramientas automáticas.

2.2.7. Programación en pares

Todo el código está escrito por pares de programadores. Dos personas escriben código en una computadora, turnándose en el uso del ratón y el teclado. El que no está escribiendo, piensa desde un punto de vista más estratégico y realiza lo que podría llamarse revisión de código en tiempo real. Los roles pueden cambiarse varias veces al día. Esta práctica no es en absoluto nueva.

2.2.8. Integración continua

Cada pieza se integra a la base de código apenas está lista, varias veces al día. Debe correrse la prueba antes y después de la integración. Hay una máquina (solamente) dedicada a este propósito.

2.2.9. Ritmo sostenible, trabajando un máximo de 8 horas por día.

Antes se llamaba a esta práctica Semana de 40 horas. Mientras en RAD las horas extras eran una best practice , en XP todo el mundo debe irse a casa a las cinco de la tarde. Dado que el desarrollo de software se considera un ejercicio creativo, se estima que hay que estar fresco y descansado para hacerlo eficientemente; con ello se motiva a los participantes, se evita la rotación del personal y se mejora la calidad del producto.

2.2.10. Todo el equipo en el mismo lugar

El cliente debe estar presente y disponible a tiempo completo para el equipo. También se llama El Cliente en el Sitio. Como esto parecía no cumplirse (si el cliente era muy junior no servía para gran cosa, y si era muy senior no deseaba estar allí), se especificó que el representante del cliente debe ser preferentemente un analista.

2.2.11. Estándares de codificación

Se deben seguir reglas de codificación y comunicarse a través del código. Según las discusiones en Wiki, algunos practicantes se desconciertan con esta regla, prefiriendo recurrir a la tradición oral. Otros la resuelven poniéndose de acuerdo en estilos de notación, indentación y nomenclatura, así como en un valor apreciado en la práctica, el llamado "código revelador de intenciones" Como en XP rige un cierto purismo de codificación, los comentarios no son bien vistos. Si el código es tan oscuro que necesita comentario, se lo debe reescribir o refactorizar.

3. 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.

3.1. 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.

3.2. Fases

3.2.1. 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.

3.2.2. Elaboración

El objetivo es probar la arquitectura del sistema.

3.2.3. 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.

3.2.4. Transición

El objetivo es validar y desplegar el sistema en el entorno de producción.

4. Dynamic System Development Method (DSDM)

Es un método que provee un framework para el desarrollo ágil de software, apoyado por su continua implicación del usuario en un desarrollo iterativo y creciente que sea sensible a los requerimientos cambiantes, para desarrollar un sistema que reuna las necesidades de la empresa en tiempo y presupuesto. Es uno de un número de métodos de desarrollo ágil de software y forma parte del alianza ágil.

4.1. Fases

1. Fase del pre-proyecto
2. Fase del ciclo de vida del proyecto
   a) Estudio de viabilidad
   b) Estudio de la empresa
   c) Iteración del modelo funcional
   d) Diseño e iteración de la estructura
   e) Implementación
3. Fase del post-proyecto.

4.2. Principios

1. Involucrar al cliente es la clave para llevar un proyecto eficiente y efectivo, donde ambos, cliente y desarrolladores, comparten un entorno de trabajo para que las decisiones puedan ser tomadas con precisión.

2. El equipo del proyecto debe tener el poder para tomar decisiones que son importantes para el progreso del proyecto, sin esperar aprobación de niveles superiores.

3. DSDM se centra en la entrega frecuente de productos, asumiendo que entregar algo temprano es siempre mejor que entregar todo al final. Al entregar el producto frecuentemente desde una etapa temprana del proyecto, el producto puede ser verificado y revisado allí donde la documentación de registro y revisión puede ser tenida en cuenta en la siguiente fase o iteración.

4. El principal criterio de aceptación de entregables en DSDM reside en entregar un sistema que satisface las actuales necesidades de negocio. No está dirigida tanto a proporcionar un sistema perfecto que resuelva todas las necesidades posibles del negocio, si no que centra sus esfuerzos en aquellas funcionalidades críticas para alcanzar las metas establecidas en el proyecto/negocio.

5. El desarrollo es iterativo e incremental, guiado por la realimentación de los usuarios para converger en una solución de negocio precisa.

6. Todos los cambios durante el desarrollo son reversibles.

7. El alcance de alto nivel y los requerimientos deberían ser base-lined antes de que comience el proyecto.

8. Las pruebas son realizadas durante todo el ciclo vital del proyecto. Esto tiene que hacerse para evitar un caro coste extraordinario en arreglos y mantenimiento del sistema después de la entrega.

9. La comunicación y cooperación entre todas las partes interesadas en el proyecto es un prerrequisito importante para llevar un proyecto efectivo y eficiente.