2019

Resumen

Alumnas:
Salgado Gutiérrez Ingrid
Pacheco Alberto Aylin Tania
grupo: 1101
profesor: Gerardo Hernández Hernández
materia: introducción a la ingeniera en software
Fecha: 16/09/19

SCRUM

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

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

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

Extrem propagamming

Extrema o Xtreme Programming (de ahora en adelante, XP) es una metodología de desarrollo de la ingeniería de software formulada por Kent Beck, autor del primer libro sobre la materia, Extreme Programming Explained: Embrace Change (1999). 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 la 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.

Fenture driven developmant

Metodología FDD (Feature Driven Development). Es una metodología ágil para el desarrollo de sistemas, basado en la calidad del software, que incluye un monitoreo constante del proyecto.

FDD fue desarrollado por Jeff De Luca y Peter Coad a mediados de los años 90. Esta metodología se enfoca en iteraciones cortas que permite entregas tangibles del producto en corto periodo de tiempo que como máximo son de dos semanas.

Características

-No hace énfasis en la obtención de los requerimientos sino en cómo se realizan las fases de diseño y construcción.

-Se preocupa por la calidad, por lo que incluye un monitoreo constante del proyecto.

-Ayuda a contrarrestar situaciones como el exceso en el presupuesto, fallas en el programa o el hecho de entregar menos de lo deseado.

-Propone tener etapas de cierre cada dos semanas.

-Se obtienen resultados periódicos y tangibles.

-Se basa en un proceso iterativo con iteraciones cortas que producen un software funcional que el cliente y la dirección de la empresa pueden ver y monitorear.

-Define claramente entregas tangibles y formas de evaluación del progreso del proyecto.

Procesos

El FDD tiene cinco procesos. Los primeros tres se hacen al principio del proyecto.

Desarrollar un modelo global: Al inicio del desarrollo se construye un modelo teniendo en cuenta la visión, el contesto y los requisitos que debe tener el sistema a construir. Este modelo se divide en áreas que se analizan detalladamente. Se construye un diagrama de clases por cada área.

Construir una lista de los rasgos: Se elabora una lista que resuma las funcionalidades que debe tener el sistema, cuya lista es evaluada por el cliente. Cada funcionalidad de la lista se divide en funcionalidades más pequeñas para un mejor entendimiento del sistema.

Planear por rasgo: Se procede a ordenar los conjuntos de funcionalidades conforme a su prioridad y dependencia, y se asigna a los programadores jefes.

Diseñar por rasgo: Se selecciona un conjunto de funcionalidades de la lista. Se procede a diseñar y construir la funcionalidad mediante un proceso iterativo, decidiendo que funcionalidad se van a realizar en cada iteración. Este proceso

iterativo incluye inspección de diseño, codificación, pruebas unitarias, integración e inspección de código.

Construir por Rasgo: se procede a la construcción total del proyecto.

Roles y responsabilidades

El equipo de trabajo está estructurado en jerarquías, siempre debe haber un jefe de proyecto, y aunque es un proceso considerado ligero también incluye documentación (la mínima necesaria para que algún nuevo integrante pueda entender el desarrollo de inmediato).

Director del Proyecto: Líder administrativo y financiero del proyecto. Protege al equipo de situaciones externas.

Arquitecto jefe: Realiza el diseño global del sistema. Ejecución de todas las etapas.

Director de desarrollo: Lleva diariamente las actividades de desarrollo. Resuelve conflictos en el equipo. Resuelve problemas referentes a recursos.

Programador Jefe: Analiza los requerimientos. Diseña el proyecto. Selecciona las funcionalidades a desarrollar de la última fase del FDD.

Propietario de clases: Responsable del desarrollo de las clases que se le asignaron como propias. Participa en la decisión de que clase será incluida en la lista de funcionalidades de la próxima iteración.

Expertos de dominio: Puede ser un usuario, un cliente, analista o una mezcla de estos. Poseen el conocimiento de los requerimientos del sistema. Pasa el conocimiento a los desarrolladores para que se asegure la entrega de un sistema completo.

Comparación

Puesto que todos los procesos se centran en la producción de software es deseable una comparación, no en su conjunto, sino según los medios que emplean y sus resultados. Realizamos una comparación entre FDD, RUP y XP.

Tamaño de los equipos: RUP esta pensado para proyectos y equipos grandes, en cuanto a tamaño y duración. FDD y XP se implementan mejor para proyectos cortos y equipos más pequeños, siendo quizás FDD más escalable que XP.

Obtención de requisitos: RUP y XP crean como base UseCases y UserStories, por lo contrario FDD no define explícitamente esa parte del proyecto sobre la adquisición de requisitos.

Evaluación del estado del proyecto: FDD es posiblemente el proceso más adecuado para definir métricas que definan el estado del proyecto, puesto que al dividirlos en unidades pequeñas es bastante sencillo hacer un seguimiento de las mismas. XP también define esos componentes pequeños. RUP por su parte, es tan grande y

complejo en este sentido como en el resto, por lo que manejar el volumen de información que puede generar requiere mucho tiempo.

Carga de trabajo: XP es un proceso ligero, esto es, que los creadores del proceso han tenido cuidado de no poner demasiadas tareas organizativas sobre los desarrolladores. RUP es un proceso pesado, basado mucho en la documentación, en la que no son deseables todos esos cambios volátiles. FDD es por su parte un proceso intermedio, en el sentido de que genera más documentación que XP pero menos que RUP.

Relación con el cliente: Con RUP se presentarán al cliente los artefactos del final de una fase, en contrapartida, la aseguración de la calidad en XP y FDD no se basa en formalismos en la documentación, si no en controles propios y una comunicación fluida con el cliente.

Conocimiento sobre la arquitectura: En RUP se intentará reducir la complejidad del software a producir a través de una planificación intensiva. En XP se conseguirá a través de la programación a pares que ya en la creación del código se puedan evitar errores y malos diseños. En FDD sin embargo se usan las sesiones de trabajo conjuntas en fase de diseño para conseguir una arquitectura sencilla y sin errores.

Test driven developmant

TDD es una criatura extraña; es simple de definir, pero su definición parece ir en contra del sentido común; es sencilla de explicar, pero difícil de llevar a cabo.

TDD es una disciplina que promueve el desarrollo de software con altos niveles de calidad, simplicidad de diseño y productividad del programador, mediante la utilización de una amplia gama de tipos de pruebas automáticas a lo largo de todo el ciclo de vida del software. El principio fundamental es que las pruebas se escriben antes que el software de producción y estas constituyen la especificación objetiva del mismo.

La primera parte de la definición suena todo miel sobre hojuelas. ¿Quién no quiere software confiable, bien diseñado y producido rápidamente?. Sin embargo, todo esto no viene gratuitamente; la palabra clave aquí es disciplina.

Disciplina: Doctrina, instrucción de una persona, especialmente en lo moral. Observancia de las leyes y ordenamientos de la profesión o instituto.

Esto nos lleva a la conclusión de que si TDD es en efecto una disciplina, entonces no es algo que aplicamos "según nos vayamos sintiendo", sino que es algo que debe formar parte integral de nuestra profesión o arte.

Las pruebas primero

La segunda parte de la definición de TDD viene con el primer "¡¿qué demonios?!" para muchos: Las pruebas se deben escribir antes que el software mismo.

Para entender que esto no es algo del otro mundo, recordemos cuando aprendimos a programar. Probablemente aprendimos con un lenguaje interpretado y normalmente comenzamos con cosas simples como por ejemplo, sumar 2 y 3:

```
>>> 2 + 3
5
```

Intuitivamente pensamos "debe dar cinco", incluso antes de oprimir la tecla Enter; y normalmente funciona; Si no, entonces hay algo definitivamente mal con el lenguaje o nuestro entendimiento del mismo. Posteriormente pasamos a cosas más complejas y/o sofisticadas, como por ejemplo:

```
>>> def sum(a, b):
...     return a + b
...
>>> sum(2, 3)
5
```

Cada que vemos aparecer en la pantalla el resultado que esperamos, aumenta nuestra autoconfianza. Esto nos motiva a seguir aprendiendo, a seguir programando. Este podría tal vez ser el ejemplo más básico de TDD.

Una vez que tomamos mayor confianza, comenzamos a escribir cantidades cada vez mayores de código entre una comprobación y la siguiente del resultado. Como creemos que nuestro código "está bien", comenzamos a "optimizar el tiempo" escribiendo más y más código de un jalón. Al poco tiempo, nos olvidamos de estas primeras experiencias, incluso tachándolas como "cosas de novatos".

Llegamos al presente y nos encontramos a nosotros mismos tratando de aprender TDD. Nos conseguimos una copia de JUnit, NUnit, o el framework de moda para nuestro lenguaje de elección y comenzamos a seguir un tutorial que encontramos por ahí. A partir de aquí, estamos en la parte sencilla de nuestra curva de aprendizaje. Comenzamos a producir grandes cantidades de pruebas y no tardamos en sentirnos cómodos con el proceso.

Conforme comenzamos a escribir pruebas para proyectos más complejos nos topamos con varios obstáculos en el camino:

Las pruebas se tornan difíciles de escribir, por lo que sentimos una desaceleración importante.

Las pruebas tardan en ejecutarse, por lo que nos volvemos renuentes a ejecutarlas frecuentemente.

Los cambios aparentemente sin importancia en el código provocan que un montón de pruebas fallen. Mantenerlas en forma y funcionando se vuelve complejo y consume tiempo.

Es en este punto donde la mayoría de las personas y equipos se da por vencido con TDD. Estamos en la parte más pronunciada de nuestra curva de aprendizaje. Estamos produciendo pruebas y obteniendo valor de las mismas, pero el esfuerzo para escribir/mantener estas mismas parece desproporcionado.

Al igual que con cualquier otra habilidad que vale la pena adquirir, si en lugar de rendirnos seguimos adelante, eventualmente aprenderemos a cruzar a la parte de nuestra gráfica donde la pendiente de la curva se invierte y comenzamos a escribir pruebas más efectivas con un menor esfuerzo y a cosechar los beneficios de nuestra perseverancia.

Aprender a escribir bien y de mantener las pruebas toma tiempo y práctica. En el resto de este artículo comparto algunos tips para ayudar a acelerar un poco este proceso.