**Agilísmo**

Con el primer computador que logro correr exitosamente un programa, se tuvo la necesidad de encontrar una forma en el que este pudiera ejecutar mas programas, ser mas rápido. Así que se fueron creando componentes hardware que permitían hacer miles de procesos mucho mas rápido.

Pero existió una necesidad de desarrollar sistemas complejos, esta nueva disciplina se basa en la planificación, ejecución y seguimientos atreves de procesos sistemáticos y repetibles.

Estos esfuerzos producen tres áreas de conocimiento que son: la ingeniería de software , la gestión predictiva de proyectos y la producción basada en procesos .

Al principio los proyectos tenían una vida de largo, lo cual generada ganancias a la empresa pero con el tiempo la vida de estos era mas corta ya que el mercado mostraba novedades, las cuales no estaban en lo proyectos. El objetivo es ver que el agilísmo es la respuesta a una necesidad.

**Modelo en cascada**

Este modelo es el realizar un proyecto siguiendo una serie de fases las cuales son:

Ingeniería y análisis del sistema, Análisis de los requisitos del software, Diseño, Codificación ,Prueba ,Mantenimiento. Pero se tiene que tener en cuenta que los proyectos no siempre siguen el flujo ya que pueden haber iteraciones y problemas, el cliente no sabe con exactitud todos los requerimientos y el cliente debe ser paciente hasta llegar a las etapas finales.

Existe un informe en donde se encuentran las diez causas principales de los fracasos de un software:

Escasa participación de los usuarios, requerimientos incompletos, cambios frecuentes en los requerimientos, falta de soporte, incompetencia tecnológica,

Falta de recursos, expectativas no realista, objetivos poco claros, cronogramas irreales y nuevas tecnologías.

**¿En qué consiste el agilísmo?**

El agilísmo es una respuesta a los fracasos y las frustraciones del modelo en cascada. No es perfección, es el hecho de reconocer que el software es propenso a errores y lo que hace es reducir sus efectos malos desde un comienzo. La finalidad de los distintos métodos que componen el agilismo es reducir los problemas clásicos de los programas de ordenador, a la par que dar más valor a las personas que componen el

quipo de desarrollo del proyecto, satisfaciendo al cliente, al desarrollador y al analista de negocio.

El utilizar el agilismo no significa hacer lo que se les viene en gana, como quieren y cuando quieren o que no se documenta, analiza o planifica pero esto no es cierto ya que las decisiones de arquitectura se toman en equipo.

En un software de proyectos los papeles mas comunes son:

Dueño de producto, cliente, analista de negocio, desarrolladores, arquitectos, administradores de sistemas.

Si el agilismo tiene tantas ventajas, ¿Por qué no lo está practicando ya todo el mundo? La resistencia al cambio es uno de los motivos fundamentales. ¿Estamos preparados para darle una oportunidad a la innovación o nos quedamos con lo de “toda la vida”, aunque sólo tenga una vida de medio siglo? Hemos aceptado una determinada forma de trabajar (en el mundo del software) y nos parece inmutable aún cuando esta industria todavía está en pañales.

Existen fuertes razones históricas para ser tan reticentes al cambio pero los que trabajamos con tecnología podemos dar un paso al frente en favor del desarrollo.

Si aceptamos que el software siempre se puede mejorar, el siguiente paso es admitir que es positivo mantener un cierto aire inconformista en nuestra actitud profesional.

El juicio sano sobre nuestro trabajo nos guía en la búsqueda de mejoras estrategias y nos ayuda a superar la pereza que nos produce la idea del cambio.

**¿Qué es el Desarrollo Dirigido por Test? (TDD)**

El desarrollo Dirigido por tests(Test Driven Developers), es una técnica de diseño e implementación de software. El cual se centra el tres pilares: La implementación de las funciones justas que el cliente necesita, la minimización del numero de defectos que llegan al software en la fase de producción y la producción de software modular, reutilizable y preparado para el cambio.

Esta técnica no trata se escribir código a la loca, sino se poder diseñar un software adecuado según los requerimientos que nos hallan dado.

En TDD dejamos que la propia implementación de pequeños ejemplos, en constantes iteraciones, haga emerger la arquitectura que necesitamos usar. Ni más ni menos. lógicamente tendremos que elegir unas herramientas de desarrollo conformes a las exigencias del guión.

La esencia de TDD es sencilla pero ponerla en práctica correctamente es cuestión de entrenamiento, como tantas otras cosas. El algoritmo TDD sólo tiene tres pasos:

Escribir la especificación del requisito (el ejemplo, el test), Implementar el código según dicho ejemplo, Refactorizar para eliminar duplicidad y hacer mejoras.

La diferencia entre el desarrollador experimentado que se sienta a hacer TDD y el junior, es cómo enfocan los tests, es decir, qué tests escriben; más allá del código que escriben.

Debe tenerse en cuenta que se supone que el principiante no está solo, sino que en un contexto XP, hay desarrolladores de más experiencia que supervisarán y habrá momentos en los que se programe en parejas.

La figura de los líderes es importante en XP al igual que en otras metodologías, con la gran diferencia de que el líder ágil está para responder preguntas y ayudar a los demás y no para darles látigo.

Un experto es capaz de anticipar futuros casos de uso y futuros problemas y será más cuidadoso diseñando la API test tras test, aplicando las buenas prácticas que conoce.

La primera vez que usamos una determinada tecnología o incluso una nueva librería, es complicado que podamos escribir la especificación antes que el SUT, porque no sabemos las limitaciones y fortalezas que ofrece la nueva herramienta. En estos casos, XP habla de spikes (disculpen que no lo traduzca, no sabría como) .Un spike es un pequeño programa que se escribe para indagar en la herramienta, explorando su funcionalidad

Para los nuevos requisitos de la aplicación, es decir, aquello que todavía falta por implementar, podremos aplicar eso de escribir el test primero y luego el código (¡y después refactorizar!).

La pregunta de los asistentes aparece antes o después: ¿no se puede aplicar TDD en un proyecto que ya está parcialmente implementado? Claro que se puede, aunque con más con- sideraciones en juego.

Es probable que el nuevo SUT colabore con partes legadas que no permiten la inyección de dependencias y que no cumplen una única res- ponsabilidad12; código legado que nos dificulta su reutilización.

Mi recomendación, antes de ponerse a reescribir partes de código legado, es crear tests de sistema (y cuando el código lo permita, tests unitarios) que minimicen los posibles efectos colaterales de la reescritura.

Además de tener dos API podemos sobrecargar métodos para intentar que el código legado y su nueva versión convivan, si es que la API antigua nos sigue sirviendo.