Una explicación de la programación extrema (XP)

V Encuentro usuarios xBase 2003 MADRID

Manuel Calero Solís.
http://www.apolosoftware.com/

Introducción

La mayoría de los programadores tenemos cierta tendencia en embebernos en cuestiones técnicas, hablar de lenguajes de programación, de técnicas de programación, de entornos de desarrollo o de editores de recursos. Pero se nos pasan por alto temas muy importantes que nos afectan tanto o más que las cuestiones mencionadas, como es la ingeniería de software, la menera en que debemos de hacer nuestro software. Alrededor de cómo hacer software hay una gran numero de autores teorías, propuestas, etc, etc., voy a tratar de presentaros una nueva disciplina de desarrollo de software, la programación extrema.

1. ¿Que es la programación extrema (XP)?

XP (eXtreme Programing) nace como nueva disciplina de desarrollo de software hace aproximadamente unos seis años, y ha causado un gran revuelo entre el colectivo de programadores del mundo. Kent Beck, su autor, es un programador que ha trabajado en múltiples empresas y que actualmente lo hace como programador en la conocida empresa automovilística DaimlerChrysler. Con sus teorías ha conseguido el respaldo de gran parte de la industria del software y el rechazo de otra parte.

La programación extrema se basa en la simplicidad, la comunicación y el reciclado continuo de código, para algunos no es mas que aplicar una pura lógica.

2. Problemas del desarrollo de software.

¿ Cuales son los principales problemas a la hora de desarrollar nuestro software ? Seguro que cualquiera de vosotros se ha topado alguna vez con alguno de estos problemas:

- Retrasos en la planificación: llegada la fecha de entregar el software éste no esta disponible.
- Sistemas deteriorados: el software se ha creado pero después de un par de año el coste de su mantenimiento es tan complicado que definitivamente se abandona su producción.
- Tasa de defectos: el software se pone en producción pero los defectos son tantos que nadie lo usa.
- Requisitos mal comprendidos: el software no resuelve los requisitos planificados inicialmente.
- Cambios de negocio: el problema que resolvía nuestro software ha cambiado y nuestro software no se ha adaptado.
- Falsa riqueza: el software hace muchas cosas técnicamente muy interesantes y divertidas, pero no resuelven el problema de nuestro cliente, ni hace que éste gane mas dinero.
- Cambios de personal: después de unos años de trabajo los programadores comienzan a odiar el proyecto y lo abandonan.

XP trata de evitar estos riesgos en nuestro desarrollo de software.

3. ¿ En que consiste XP ? Sus objetivos.

Los objetivos de XP son muy simples: la satisfacción del cliente. Esta metodología trata de dar al cliente el software que él necesita y cuando lo necesita. Por tanto, debemos responder muy rápido a las necesidades del cliente, incluso cuando los cambios sean al final de ciclo de la programación.

El segundo objetivo es potenciar al máximo el trabajo en grupo. Tanto los jefes de proyecto, los clientes y desarrolladores, son parte del equipo y están involucrados en el desarrollo del software.

3.1. Las cuatro variables.

XP define cuatro variables para proyectos de software: coste, tiempo, calidad y ámbito.

Además de estas cuatro variables, Beck propone que sólo tres puedan ser establecidas por las fuerzas externas (jefes de proyecto y clientes), mientras que el valor de la cuarta variable debe ser establecido por los programadores en función de las otras tres.

Pongámonos en un episodio diario de desarrollo.

El jefe de proyecto*: "Quiero estos requisitos realizados para el día 1 de mes próximo, con lo que contáis con el equipo actual. ¡Ah ya sabéis que la calidad es lo primero!"*

Todos sabemos qué es lo primero que salta por la ventana en estos casos: "la calidad", ¿ Por qué ? Porqué nadie es capaz de trabajar bien cuando se le somete a mucha presión.

XP nos propone que juguemos todas las partes implicadas en el proyecto hasta que el valor que alcancen las cuatro variables sea el correcto para todas las partes: *"Si quieres mas calidad en menos tiempo tendrás que aumentar el equipo e incrementar el coste".*

Además con el agravante de que estas cuatro variables no guardan una relación tan directa como en principio pueda parecer. El incremento del número de programadores no repercutirá de manera lineal en el tiempo de desarrollo del proyecto, siendo de todos conocido el dicho: *"nueve mujeres no pueden tener un hijo en un mes".*

Con la calidad suele suceder un fenómeno extraño: frecuentemente un proyecto que tratemos de aumentar la calidad conduce a que el proyecto pueda realizarse en menos tiempo, siempre con unos márgenes obviamente. Es verdad que cuando un equipo de desarrollo se acostumbra a realizar pruebas intensivas, se siguen estándares de codificación, poco a poco se comenzara a andar mas rápido y mas seguro, por tanto mas preparados para futuros cambios, sin estrés y así sucesivamente.

Frente a esto existe la tentación de entregar el trabajo mas rápido, por tanto probar menos, codificar más rápido y peor, sin hacer planteamientos maduros, esto repercutirá en la confianza de nuestros clientes, al entregarle trabajos con fallos. Esta es una apuesta a muy corto plazo y suele ser una invitación al desastre, conduce a la desmoralización del equipo, y con ello a la larga a la ralentización del proyecto y la perdida de tiempo que habríamos conseguido en un principio.

La cuarta variable, el ámbito del proyecto, suele ser conveniente que sea establecida por el equipo de desarrollo. Es una variable muy importante que nos va a decir donde vamos a llegar con nuestro software, que problemas vamos a resolver y cuales vamos a dejar para siguientes versiones. Cuantas veces hemos escuchado *"Los clientes no nos pueden decir lo que quieren. Cuando le damos lo que nos piden no les gusta".* Y es que los requisitos nunca son claros al principio y el mismo desarrollo del software hace cambiar los requisitos. Por tanto el ámbito debe de ser dúctil, podremos jugar con el, si el tiempo para el lanzamiento es limitado, siempre habrá cosas que pudramos diferir para siguientes versiones.

Por tanto implementaremos primero los requisitos mas importantes para el cliente, de forma que si tenemos que dejar algo para después que sea menos importante que las que ya incorpore un sistema.

3.2. El coste del cambio.

Una de las suposiciones establecidas en la industria del software es que el coste de los cambios en un programa crece exponencialmente con el tiempo. XP propone que si el sistema que empleas hace que el coste del software aumente con el tiempo debes de actuar de forma diferente a cómo lo haces. XP propugna que esta curva ha perdido validez y con una combinación de buenas practicas de programación y tecnología es posible lograr que la curva sea la contraria, por tanto se pretende conseguir esto:

Si decidimos aceptar el cambio debemos de desarrollar para basarnos en dicha curva, pero ¿cómo se consigue dicha curva?, no existe una forma mágica desde luego hay varias medidas que nos ayudan a conseguirla.

Diseñar los más sencillo que sea posible, para hacer sólo lo imprescindible en un momento dado, la simplicidad del código y los test continuos hacen que los cambios sean posibles tan a menudo como sea necesario.

La programación orientada a objetos es una tecnología clave para el mantenimiento del software, cada mensaje a un objeto es una oportunidad de cambio sin necesidad de cambiar el código existente, esto no quiere decir que no puedas tener flexibilidad sin programar orientado al objeto y el caso contrario que haya programas orientados a objetos que nadie quería tocar, sólo se dice que el programar orientado a objetos reduce el costo del cambio.

En vez de tomar grandes decisiones al principio y pocas posteriormente, podemos idear una aproximación para desarrollar software en la que se tomen decisiones rápidamente, pero estas decisiones apoyadas por pruebas y que te preparan para mejorar el diseño del software cuando aprendas una mejor manera de diseñarlo.

He oído a muchos programadores (entre los que me incluyo) decir: *"Hasta que no he terminado el programa no lo he entendido ahora lo haría con esta jerarquía y que esta clase herede de esta otra"*, dejemos pues abierta la puesta a esta posibilidad.

3.3. Los cuatro valores.

Una de las cosas que a los programadores nos tiene que quedar muy claro es que en el ciclo de vida del desarrollo de un proyecto software los cambios van a aparecer, cambiarán los requisitos, las reglas de negocio, el personal, la tecnología, todo va a cambiar. Por tanto el problema no es el cambio en si, ya que este va a suceder sino la incapacidad de enfrentarnos a estos cambios.

Como en otra cualquier actividad humana necesitamos valores para desarrollar nuestro trabajo y conseguir los planteamientos iniciales.

Estos cuatro valores son:

- Comunicación
- Sencillez
- Retroalimentación
- Valentía

Comunicación.

Cuantas veces hemos tenido problema en nuestro equipo de desarrollo por falta de comunicación, por no comentar un cambio crítico en el diseño, por no preguntar lo que pensamos al cliente. La mala comunicación no surge por casualidad y hay circunstancias que conducen a la ruptura de la comunicación, como aquel jefe de proyecto que abronca al programador cuando éste lo comunica que hay un fallo en el diseño. XP ayuda mediante sus prácticas a fomentar la comunicación.

Sencillez.

Siempre debemos hacernos esta pregunta ¿ Qué es lo más simple que pueda funcionar ?. Lograr la sencillez no es fácil. Tenemos cierta tendencia a pensar en qué programaremos mañana, la próxima semana y el próximo mes. Cuantos de nosotros no hacemos a veces mas de lo que debemos: "*Ya que estoy tocando esta clase voy a añadirle dos métodos mas para visualizar los mensajes en colores*", cuando eso no está entre los requisitos, "*es que mañana puede que lo necesite*", si mañana esta entre los requisitos, hazlo entonces.

XP nos enseña a apostar, apuesta por hacer una cosa sencilla hoy y pagar un poco mas para mañana, si es necesario, que hacer una cosa complicada hoy y no utilizarla después. La sencillez y la comunicación se complementan, cuanto mas simple es tu sistema menos tienes que comunicar de el.

Retroalimentación.

"No me preguntes a mi, pregúntale al sistema", es la primera clave de la retroalimentación, por medio de pruebas funcionales a nuestro software este nos mantendrá informado del grado de fiabilidad de nuestro sistema, esta información realmente no tiene precio. Los clientes y las personas que escriben pruebas tienen una retroalimentación real de su sistema.

La retroalimentación actúa junto con la sencillez y la comunicación, cuanto mayor retroalimentación más fácil es la comunicación. Cuanto mas simple un sistema mas fácil de probar. Escribir pruebas nos orienta como simplificar un sistema, hasta que las pruebas funcionen, cuando las pruebas funcionen tendrá mucho echo.

Valentía.

Asumir retos, ser valientes antes los problemas y afrontarlos. En mi experiencia como programador estuve realizando un programa de Nominas para la delegación portuguesa de SP. Yo estaba encargado de elaborar unos informes para la seguridad social portuguesa denominados "Balanço Social", nunca se me olvidará. Mientras mas avanzaba en el trabajo mas engorro montaba en el código por tal de que aquello funcionara después de 3 semanas de trabajo casi lo tenia resuelto, pero el código estaba tan súper parcheado que no había nadie que lo entendiera. Realmente hasta ese momento yo no entendía realmente lo que se necesitaba y decidí tirarlo todo y reprogramar todo el sistema, al principio de los cambios todo empezó a fallar, pero ahora avanzaba firme y seguro, a los pocos días todas las pruebas empezaron a funcionar, hasta que finalicé el trabajo. Es una de las acciones valientes que adoptas y después te enorgulleces de ellas, otras veces me he escondido detrás de la complejidad de los cambios y no los he realizado, me acobarde pensando en las consecuencias, en las broncas de mis jefes si les decía que había cambiado el sistema. Cuando no afrontas el problema y parcheas un código que positivamente sabes que esta mal acabas odiando el sistema, y cada mañana cuando vas a la oficina en el coche te entra dolor de barriga. Cuando vayas hacia el trabajo y se te revuelva el estomago piensa en cambiar de trabajo.

Nuestro trabajo se asimila al de un escalador cuando hacemos una cima tenemos que volver a bajar para hacer otra cima y así constantemente, planteándonos hacer sistemas cada vez mas sencillos y fiables.

La valentía junto con la comunicación y la sencillez se convierte en extremadamente valiosa. "Odio este código ¿vamos a ver cuanto podemos cambiar en esta tarde?"

Para continuar tenemos que disponer de unas guías mas concretas que satisfagan y encarnen estos cuatro valores.

3.4. Las cuatro actividades basicas

Ahora que tenemos nuestros cuatro valores estamos preparados para construir una disciplina de desarrollo de software. ¿ Qué tareas debemos de llevar a cabo para desarrollar un buen software ?

Codificar

Es la única actividad de la que no podremos prescindir. Sin código fuente no hay programa, aunque hay gente que cuenta que existe software en producción del que se perdió el código fuente. Por tanto necesitamos codificar y plasmar nuestras ideas a través del código. En una programación en XP en pareja el código expresa tu interpretación del problema, así podemos utilizar el código para comunicar, para hacer mías tus ideas, y por tanto para aprender y mejorar.

Hacer pruebas

Las características del software que no pueden ser demostradas mediante pruebas simplemente no existen. Las pruebas me dan la oportunidad de saber si lo que implementé es lo que en realidad yo pensaba que había implementado. Las pruebas nos indican que nuestro trabajo funciona, cuando no podemos pensar en ninguna prueba que pudiese originar un fallo en nuestro sistema entonces has acabado por completo.

No debemos de escribir tan solo una prueba ver que funciona y salir corriendo, debemos de pensar en todas las posibles pruebas para nuestro código, con el tiempo llegaras a conclusiones sobre las pruebas y podrás pensar que si dos de tus pruebas ya funcionan la tercera prueba no será necesaria escribirla, sin caer en demasiada confianza.

Programar y probar es mas rápido que sólo programar. Puedes ganar media hora de productividad sin hacer pruebas, pero perderás mucho tiempo en la depuración. Tendrás menos errores, tendrás que volver menos veces sobre el código, te costará menos localizar los errores, perderás menos tiempo escuchado como tus clientes te dicen que no funciona.

Las pruebas deben de ser sensatas y valientes. No podemos hacer pruebecillas que no testen a fondo el sistema, esos agujeros que vamos dejando nos esperan para cuando pasemos de nuevo por allí y volveremos a caer dentro.

Escuchar

Los programadores no lo conocemos todo, y sobre todo muchas cosas que las personas de negocios piensan que son interesantes. Si ellos pudieran programarse su propio software ¿ para que nos querrían ?.

Si vamos a hacer pruebas tenemos que preguntar si lo obtenido es lo deseado, y tenemos que preguntar a quien necesita la información. Tenemos que escuchar a nuestros clientes cuales son los problemas de su negocio, debemos de tener una escucha activa explicando lo que es fácil y difícil de obtener, y la realimentación entre ambos nos ayudan a todos a entender los problemas.

Diseñar

El diseño crea una estructura que organiza la lógica del sistema, un buen diseño permite que el sistema crezca con cambios en un solo lugar. Los diseños deben de ser sencillos, si alguna parte del sistema es de desarrollo complejo, divídela en varias. Si hay fallos en el diseño o malos diseños, estos deben de ser corregidos cuanto antes.

Tenemos que codificar porque sin código no hay programas, tenemos que hacer pruebas por que sin pruebas no sabemos si hemos acabado de codificar, tenemos que escuchar, porque si no escuchamos no sabemos que codificar ni probar, y tenemos que diseñar para poder codificar, probar y escuchar indefinidamente.

4. La solución

Hasta ahora sólo hemos montado la escena, conocemos nuestros problemas, hemos adoptado nuestros valores y hemos decidido cuales son las actividades básicas para desarrollar software.
Nuestra intención es que todo lo expuesto funcione y para ello vamos a relacionar una serie de prácticas para que todo llegue a buen puerto:

4.1. Fases de la metodología XP

Existen diversas prácticas inherentes al desarrollo de software.

4.4.1.- Planificación.

XP plantea la planificación como un permanente dialogo entre las partes la empresarial (deseable) y la técnica (posible). Las personas del negocio necesitan determinar:

Ámbito: ¿ Qué es lo que el software debe de resolver para que este genere valor ?
Prioridad: ¿ Qué debe ser hecho en primer lugar ?
Composición de versiones: ¿ Cuánto es necesario hacer para saber si el negocio va mejor con software que sin el ?. En cuanto el software aporte algo al negocio debemos de tener lista las primeras versiones.
Fechas de versiones: ¿ Cuáles son las fechas en la presencia del software o parte del mismo pudiese marcar la diferencia ?

El personal del negocio no puede tomar en vació estas decisiones, y el personal técnico tomará las decisiones técnicas que proporcionan la metería prima para las decisiones del negocio.

Estimaciones: ¿ Cuanto tiempo lleva implementar una característica ?
Consecuencias: Informar sobre las consecuencias de la toma de decisiones por parte del negocio. Por ejemplo el cambiar las bases de datos a Oracle.
Procesos: ¿ Cómo se organiza el trabajo y el equipo ?
Programación detallada: Dentro de una versión ¿ Qué problemas se resolverán primero ?

4.1.2.- Pequeñas versiones.

Cada versión debe de ser tan pequeña como fuera posible, conteniendo los requisitos de negocios más importantes, las versiones tiene que tener sentido como un todo, me explico no puedes implementar media característica y lanzar la versión.

Es mucho mejor planificar para 1 mes o 2 que para seis meses y un año, las compañías que entregan software muy voluminoso no son capaces de hacerlo con mucha frecuencia.

4.2.- Diseño

4.2.1.- Metáfora.

Una metáfora es una historia que todo el mundo puede contar a cerca de cómo funciona el sistema. Algunas veces podremos encontrar metáforas sencillas *"Programa de gestión de compras, ventas, con gestión de cartera y almacén".* Las metáforas ayudan a cualquier persona a entender el objeto del programa.

4.2.2. Diseño sencillo.

El diseño adecuado par el software es aquel que:

1.Funciona con todas las pruebas.
2.No tiene lógica duplicada.
3.Manifiesta cada intención importante para los programadores
4.Tiene el menor número de clases y métodos.

Haz el diseño lo mas simple posible borra todo lo que puedas sin violar las reglas 1,2 y 3. Contrariamente a lo que se pensaba el *"Implementa para hoy, diseña para mañana"*, no es del todo correcto si piensas que el futuro es incierto.

4.3.- Desarrollo.

4.3.1.- Recodificación.

Cuando implementamos nuevas características en nuestros programas nos planteamos la manera de hacerlo lo mas simple posible, después de implementar esta característica, nos preguntamos como hacer el programa mas simple sin perder funcionalidad, este proceso se le denomina recodificar o refactorizar (refactoring). Esto a veces nos puede llevar a hacer mas trabajo del necesario, pero a la vez estaremos preparando nuestro sistema para que en un futuro acepte nuevos cambios y pueda albergar nuevas características. No debemos de recodificar ante especulaciones si no solo cuándo el sistema te lo pida.

4.3.2.- Programación por parejas.

Todo el código de producción lo escriben dos personas frente al ordenador, con un sólo ratón y un sólo teclado. Cada miembro de la pareja juega su papel: uno codifica en el ordenador y piensa la mejor manera de hacerlo, el otro piensa mas estratégicamente, ¿ Va a funcionar ?, ¿ Puede haber pruebas donde no funcione ?, ¿ Hay forma de simplificar el sistema global para que el problema desaparezca ?.

El emparejamiento es dinámico, puedo estar emparejado por la mañana con una persona y por la tarde con otra, si tienes un trabajo sobre un área que no conoces muy bien puedes emparejarte con otra persona que si conozca ese área. Cualquier miembro del equipo se puede emparejar con cualquiera.

4.3.3.- Propiedad colectiva.

Cualquiera que crea que puede aportar valor al código en cualquier parcela puede hacerlo, ningún miembro del equipo es propietario del código. Si alguien quiere hacer cambios en el código puede hacerlo. Si hacemos el código propietario, y necesitamos de su autor para que lo cambie entonces estaremos alejándonos cada vez mas de la comprensión del problema, si necesitamos un cambio sobre una parte del código lo hacemos y punto. XP propone un propiedad colectiva sobre el código nadie conoce cada parte igual de bien pero todos conoce algo sobre cada parte, esto nos preparará para la sustitución no traumática de cada miembro del equipo.

4.3.4.- Integración continúa.

El código se debe integrar como mínimo una vez al día, y realizar las pruebas sobre la totalidad del sistema. Una pareja de programadores se encargara de integrar todo el código en una maquina y realizar todas las pruebas hasta que estas funcionen al 100%.

4.3.5.- 40 Horas semanales.

Si queremos estar frescos y motivados cada mañana y cansado y satisfecho cada noche. El viernes quiero estar cansado y satisfecho para sentir que tengo dos días para pensar en algo distinto y volver el lunes lleno de pasión e ideas. Esto requiere que trabajemos 40 horas a la semana, mucha gente no puede estar más de 35 horas concentrados a la semana, otros pueden llegar hasta 45 pero ninguno puede llegar a 60 horas durante varias semanas y aun seguir fresco, creativo y confiado. Las horas extras son síntoma de serios problemas en el proyecto, la regla de XP dice nunca 2 semanas seguidas realizando horas extras.

4.3.6.- Cliente In-situ.

Un cliente real debe sentarse con el equipo de programadores, estar disponible para responder a sus preguntas, resolver discusiones y fijar las prioridades. Lo difícil es que el cliente nos ceda una persona que conozca el negocio para que se integre en el equipo normalmente estos elementos son muy valiosos, pero debemos de hacerles ver que será mejor para su negocio tener un software pronto en funcionamiento, y esto no implica que el cliente no pueda realizar cualquier otro trabajo.

4.3.7.- Estándares de codificación.

Si los programadores van a estar tocando partes distintas del sistema, intercambiando compañeros, haciendo refactoring, debemos de establecer un estándar de codificación aceptado e implantado por todo el equipo.

4.4.- Pruebas

4.4.1.- Hacer pruebas.

No debe existir ninguna característica en el programa que no haya sido probada, los programadores escriben pruebas para chequear el correcto funcionamiento del programa, los clientes realizan pruebas funcionales. El resultado un programa mas seguro que conforme pasa el tiempo es capaz de aceptar nuevos cambios.

5.- ¿Como hacemos funcionar esto?

Vamos a tratar de explicar como se ponen en marcha todas estas practicas se apoyan entre si y toman valor, veremos como todo esta historia de XP puede funcionar.

5.1.- Planificación

En principio no podriamos comenzar el programa con tan sólo un plan aproximando y no podriamos estar actualizando este plan constantemente a no ser que:

- Los propios clientes hiciesen su planificación con las estimaciones que les pasan los programadores.
- Le diéramos a los clientes un plan para hacerles una idea de lo que seria posible en los próximos meses.
- Hiciéramos versiones pequeñas para que el cliente detecte cualquier error en el plan.
- Tu cliente esté incorporado al equipo, para observar rápidamente los posibles cambios.

5.2.- Versiones reducidas

En principio no podíamos tener una producción después de unos pocos meses, a no ser que:

- La planificación te ayudase a trabajar sobre las historias más valiosas, de tal forma que un pequeño tuviese valor para el negocio.
- Estuvieses integrando constantemente de tal forma que el coste de la versión fuese mínimo.
- Tus pruebas redujesen los defectos lo suficiente, para evitar los largos ciclos de testeo.
- Hiciéramos diseños sencillos necesarios únicamente para esta versión.

5.3.- Metáfora.

No podríamos comenzar a desarrollar con tan solo una metáfora a no ser que:

- Tengas rápidamente retroalimentación a partir del código real y de las pruebas, sobre si esta metáfora esta funcionando en la práctica.
- Tus clientes estén a gusto hablando sobre el sistema en términos de metáfora.
- La Recodificación que hagas refine continuamente vuestro conocimiento de lo que la metáfora significa en la práctica.

5.4.- Diseño sencillo.

En principio no tendríamos bastante diseñado para codificar hoy, a no ser que:

- Utilizáramos la Recodificación, haciendo cambios que no fuesen preocupantes.
- Tuviésemos una metáfora global clara, de tal forma que los cambios del diseño tenderían a seguir caminos convergentes.

- Estuviésemos codificando con un compañero, de tal forma que tuvieses confianza en hacer un diseño sencillo, y no un diseño estupido.

5.5.- Hacer pruebas.

En principio escribir pruebas nos llevaría mucho tiempo, a menos que:

- El diseño sea tan simple como pueda ser de tal forma que escribir pruebas no sea difícil.
- Estemos programando con un compañero, así no puedes pensar en otra prueba pero tu compañero si puede.
- Te sientas bien cuando veas las pruebas funcionando.
- Tus clientes se sientan bien cuando vean todas las pruebas funcionando.

5.6.- Recodificación.

En principio no podríamos hacer Recodificación del sistema durante todo el tiempo, nos llevaría mucho tiempo y sería difícil de controlar, a menos que:

- Estemos habituados a la propiedad colectiva y no tengamos inconvenientes en hacer cambios necesarios.
- Trabajemos sobre estándares de codificación, para que no tengamos que cambiar el formato del código antes de recodificar.
- Codifiquemos por parejas y esto nos de valentía a la hora de afrontar mejoras difíciles en el código.
- Tengamos diseños sencillos donde recodificar sea más fácil.
- Tengamos integración continua para que si accidentalmente dañemos algo lo sepamos en cuestión de horas.
- Estemos descansados y así tengamos más valentía y sea mas improbable que cometamos errores.

5.7.- Programación en parejas.

En principio escribir todo el código por parejas seria más lento, a no ser que:

- Los estándares de codificación reduzcan las disputas.
- Cada uno este fresco y descansado y así evitar las discusiones absurdas.
- Las parejas escribas las pruebas juntas, dando la posibilidad de alinear su comprensión antes de afrontar el meollo de la implementación.
- Las parejas tengan la metáfora para fundamentar sus discusiones sobre los nombres y el diseño básico.
- Las parejas estén trabajando sobre diseños sencillos.

5.8.- Propiedad colectiva.

En principio no podrás dejar a todo el mundo cambiar todo lo que deseen. Las personas estropean cosas a diestro y siniestro, a menos que:

- Integremos después de un corto periodo de tiempo.
- Escribamos y hagamos pruebas, así la posibilidad de dañar las cosas accidentalmente disminuye.
- Programemos por parejas así es menos probable que dañemos el código, y los programadores aprendan mas rápido lo que pueden cambiar con beneficio.
- Te adhieras a los estándares de codificación, así no entras en las espantosas Guerras de los Corchetes

5.9.- Integración continúa.

Posiblemente no podremos integrar tras unas pocas horas de trabajo, a no ser que:

- Podamos ejecutar pruebas rápidamente para saber que no hemos perdido nada.
- Codifiques en parejas, así hay la mitad de cambios a integrar.
- Recodifiques, así hay piezas mas pequeñas, reduciendo la posibilidad de conflicto.

5.10.- 40 Horas semanales.

En principio no podríamos trabajar 40 horas semanales pues no daríamos el suficiente valor a nuestro negocio, a menos que:

- La planificación nos este dando mas trabajo valioso que hacer.
- La combinación de planificación reduzca la frecuencia de malas sorpresas, donde tienes que hacer más de lo que piensas.
- Las practicas como un todo te ayudaran a programar a gran velocidad.

5.11.- Clientes in-situ.

En principio no podremos tener a un cliente in-situ ya que este produce más valor en otra parte, a menos que:

- Puedan producir valor para el proyecto escribiendo pruebas funcionales.
- Puedan producir valor para el proyecto priorizando el a pequeña escala y tomando decisiones junto a los programadores.

5.12. Estándares de codificación.

En principio no podemos pedirle al equipo que codifique bajo un estándar común, los programadores somos individualistas. A menos que:

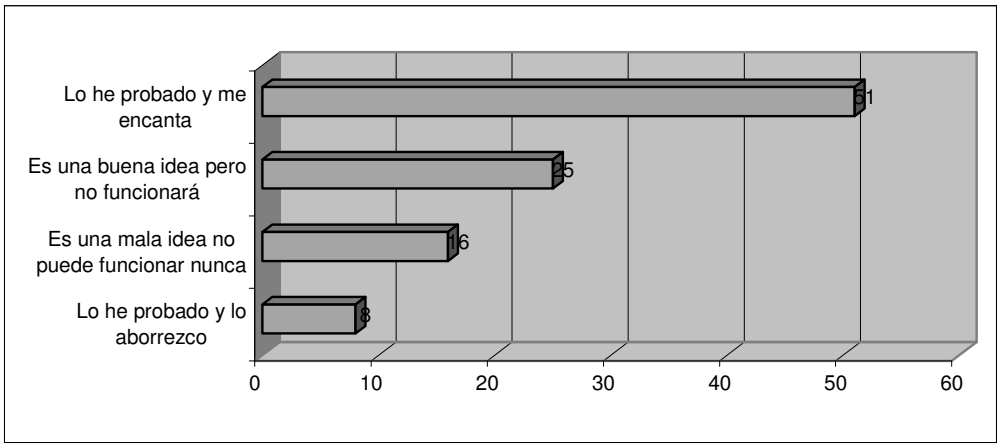
- Toda XP le de la posibilidad de sentirse dentro de un equipo ganador.

Conclusión.

Ninguna práctica funciona bien por si sola (con la excepción del las pruebas). Requieren las otras practicas par equilibrarse.

5.- Comparativa con metodologías tradicionales.

XP ha causado un gran revuelo en la comunidad de la ingeniería del software. Muestro una gráfica llevada a cabo por IBM.



XP tiene muchas criticas especialmente contra la pair programing – sobre todo por parte de los jefes de proyecto- pero también por parte de muchos programadores con gran sentimiento de posesión del código, piensan que ellos son los mejores conocedores de las herramientas y lenguajes que utilizan y que si no lo entiendes es por que no sabes lo suficiente.

También se critica el mito de las 40 horas semanales, y que es un lujo para las exigencias del mercado.

También hay críticas hacia XP que dicen que solo puede funcionar con programadores muy buenos, como Kent Beck, que son capaces de hacer un buen diseño, sencillo y fácilmente extensible.

XP es mas una filosofía de trabajo que una metodología. Por otro lado ninguna de las practicas defendidas por XP son invención de este método, XP lo que hace es ponerlas todos juntas.

XP esta diseñado para grupos de pequeños programadores mas de 10 ya seria muy complicado, y para que estén en el mismo centro de trabajo.

Las metodologías tradicionales imponen un proceso disciplinado para tratar de hacer el trabajo predecible, eficiente y planificado. Estos métodos están orientados a documentos y se vuelven demasiado burocráticas e ineficaces. XP es más liviana y ágil y están orientadas más a las personas que a los procesos.

XP supone:

- Las personas son claves en los procesos de desarrollo.
- Los programadores son profesionales no necesitan supervisión.
- Los procesos se aceptan y se acuerdan, no se imponen.
- Desarrolladores y gerentes comparten el liderazgo del proyecto.
- El trabajo de los desarrolladores con las personas que conocen el negocio es regular, no puntual.

Y conviene recordar que:

"ninguna metodología hace el trabajo por si sola, pero te podrá ayudar".