Extreme Programming
(Programación extrema)

Ing. Mauricio Campos
Universidad Tecnológica Nacional
Facultad Regional Buenos Aires
Buenos Aires, Argentina
Julio 2004
mauricio@labint.frba.utn.edu.ar

"Luego que un proyecto termina sabremos suficiente para hacer un buen plan, pero este ya no es útil"

RESUMEN
Los proyectos de desarrollo de software están rodeados de riesgos. La capacidad para lidiar con ellos es esencial para el éxito. Las dos categorías de administración de riesgos son: Anticipación o elasticidad.
Anticipación tiende a identificar y resolver todos los posibles problemas antes de programar. Esta lógica es aceptada por los métodos clásico y es muy costosa en ambientes de constantes cambios.
Por otro lado la **elasticidad** es la capacidad de afrontar con éxito los peligros imprevistos, luego de que se hayan manifestado. Este es el principal criterio de Programación extrema (XP) [Kent Beck].
XP puede ser descripta con tres palabras: Simplicidad–Comunicación-Feedback

Keywords
XP; extreme programming; desarrollo de software; proceso de desarrollo liviano.

1 INTRODUCCION
El ambiente instable que rodea a los proyectos de desarrollo de software hace imposible el uso de las técnicas clásicas, como ser "método de cascada". Los resultados insatisfactorios y la necesidad de tener un modo eficiente que guíe el proceso de desarrollo ha hecho que personas alrededor del mundo comiencen a crear técnicas, mas adaptadas a la realidad. Programación extrema (XP) es una de ellas.
Es una metodología liviana para equipos chicos a medianos que afronten requerimientos vagos o cambiantes.

2 CONCEPTOS GENERALES
El equipo XP incluye desarrolladores, manager y clientes, todos trabajando juntos, haciéndose preguntas, negociando alcances y cronogramas y creando las pruebas de funcionalidad. El grupo de programadores es pequeño, generalmente entre 2 y 12 personas.
El principal objetivo es entregar el software requerido cuando es requerido. Para lograrlo la metodología utiliza herramientas de buenas prácticas de desarrollo de software.

3 LAS MEJORES PRACTICAS
Metáfora de sistema: Cada proyecto tiene una metáfora que provee una convención de nombres fácil de recordar.
Diseño simple: Siempre utiliza el diseño mas simple posible que cumpla el trabajo.
Pruebas continuas: Los equipos de XP se enfocan en la validación del software en todo momento. Los programadores escriben las pruebas antes que el código.
Re-factoring: Re-factor elimina cualquier línea de código duplicada durante la etapa de codificación. Es el proceso de cambiar el código para mejorar la estructura mientras evoluciona.
Programación en pareja: El código es escrito por dos programadores sentados en una máquina. Esencialmente todo el código es revisado mientras es escrito.

Integración continua: Todos los cambios son integrados en el código base por lo menos un vez al día.
Pequeños Releases (lanzamientos): Comenzar con el conjunto de funcionalidades útiles mas pequeño. Agregar continuamente porciones de funcionalidad en cada lanzamiento.
Semana laboral de 40 horas: Los programadores se van a casa a tiempo. Continuas horas extras es considerado un indicador de que el proceso y/o cronograma están mal.
Cliente In-situ : Del equipo de desarrolladores tienen continuo acceso continuo a un cliente, este debe ser alguien que vaya a utilizar el sistema.
Codificación estándar: Todos codifican bajo el mismo estándar Lo ideal sería no poder identificar, al mirar el código, por quien fue escrito.
Código colectivo: A nadie le "pertenece" un módulo. Cualquier desarrollador debe poder trabajar en cualquier parte del código base en cualquier momento.

4 DICCIONARIO SINTETICO

Carta: Término genérico para describir a una *tarea* o *historia.*
Entrenador (Coach)*:* Un desarrollador con experiencia quien enseña y guía a los otros a través de la metodología XP.
Cliente: Es la persona que dirige los requerimientos. Es aconsejable que sea un usuario final o un gerente de producto.
Días/semanas ideales de programación: Se refieren a cuanto toma (en días o semanas) completar una tarea si esto es tu única actividad y no hay distracciones.
Iteración: Unidad de tiempo al cabo de la cual una versión del software es internamente lanzada
(Típicamente dura entre 1 a 3 semanas)
Programación en parejas*:* Dos desarrolladores sentados juntos en una máquina y escribiendo el código operativo.
Lanzamiento: Unidad de tiempo al cabo de la cual una versión del proyecto será lanzada al usuario final(pocas iteraciones)
Pizarra de historias: Pizarra en la cual son desplegadas las tarjetas de historias y tareas divididas en iteraciones y lanzamientos.
Historia: Un Caso de uso simplificado que provee valor agregado al cliente.
Tarea: Tarea del desarrollo que normalmente solo cumple parcialmente la funcionalidad de una historia.
Clima de ayer: Modo simple de conocer cuanto trabajo puede hacerse en una iteración. Se calcula a partir de cuanto trabajo se hizo en la ultima iteración.

5 PIEDRAS ANGULARES

- **Pruebas** – (Unidad de prueba)
  - Las unidades de pruebas son escritas antes que el código y son puestas dentro del mismo repositorio con el código.
  - Código sin pruebas no será aceptado.
  - Requiere que todo el código pase todas las unidades de prueba. Asegurando así que todas las funcionalidades trabajan bien.
- **Doble chequeo** – (Programación en pareja)
  - Todo el código es escrito por dos programadores trabajando en la misma máquina, ellos constantemente chequean el código de cada uno y comparten opiniones.
  - Esta práctica asegura que todo el código es revisado, esto resulta en mejor diseño, mejor prueba, y mejor código.
- **Simpleza** – (Hacerlo fácil)
  - Los métodos de diseño y programación en XP tienden a ser tan simples y confiables como sea posible.
  - "Un diseño simple siempre toma menos tiempo que uno complejo. Hazlo de la manera mas simple que funcione. Mantén las cosas tan simple como sea posible el tiempo que sea posible, mediante

nunca agregar funcionalidades antes de lo previsto"

- **Comunicación** – (Usuario experto)
  - Trabaje con un cliente, quien se convertirá al final del proyecto en un usuario experto.
  - El cliente debe ser alguien que conozca las metas del negocio y las funcionalidades requeridas del software.
  - Durante todas las fases se requiere comunicación con este cliente, preferentemente cara a cara en el lugar.
  - Funciones principales del cliente:
    - Escribir las "Historias de Usuario".
    - Asignar las prioridades.
    - Negociar las "Historias de Usuario" para que sean incluidas en los lanzamientos.
    - Negociar los momentos de cada pequeño lanzamiento.

6 PASO A PASO

El Nuevo proyecto XP comienza recolectando las historias de usuario[7] y llevando a cabo las spike solutions[8.1] para los puntos que parecen ser riesgosos. Esto lleva entre 2 y 4 semanas.
Cuando esto se encuentra terminado ya conoceremos los costos estimados y conoceremos el perfil del sistema. Luego estamos listos para crear el (necesariamente impreciso) cronograma del proyecto, con el cual todos están de acuerdo, esto se llama "Plan del Lanzamiento"[9]. Hasta el primer plan es suficientemente confiable para tomar las decisiones, de todos modos el equipo XP revisa el plan regularmente.
El desarrollo comenzará con una reunión de plan de iteración [10]. Esta reunión es una practica mediante la cual el equipo recibe instrucciones cada par de semanas. El equipo construye software en iteraciones bisemanales, entregando un utilitario funcional al fin de cada iteración.
Durante el planeamiento de la iteración el cliente presenta las funcionalidades deseadas para las próximas dos semanas. Los programadores las quiebran en tareas y estiman su costo (a un nivel mas fino de detalle que en el Plan de Lanzamiento). Basándose en la cantidad de trabajo terminado en la anterior iteración, el equipo acuerda lo que será encarado en la presente iteración.

7 HISTORIAS DE USUARIOS

Las historias de usuario se utilizan en reemplazo de grandes documentos de requerimiento y ellas son escritas por el cliente (quien trabajará con el equipo de desarrollo a lo largo de todo el proyecto) como las cosas que quiere que el sistema haga para El, en sus propias palabras y sin terminologías de sintaxis técnica.
Las historias de usuario están en un formato de aproximadamente tres oraciones (la tarjeta que se utiliza para escribirlas también las limita). Ellas solo deben proveer suficiente detalles para hacer una razonable estimación de cuanto tiempo llevará implementar la historia. Cuando llegue el momento de implementar la historia, los desarrolladores irán con los clientes para recibir una descripción detallada del requerimiento.
El *tiempo ideal* de desarrollo (entre 1 y 3 semanas) es estimado por los desarrolladores y muestra cuanto tiempo llevaría implementar la historia en caso de que no hubiese otras tareas ni distracciones y uno sabe exactamente que hacer. Mas allá de tres semanas significa que la historia debe ser quebrada en mas de una. Menos de una semana significa que debe ser combinada con otra historia. El número óptimo para crear un plan de lanzamiento es entre 100 a 60 historias de usuario.

8 METAFORA

La arquitectura del sistema es construida para guiar la "Integridad Conceptual" (la consideración mas importante en el diseño de sistemas). En cambio de una arquitectura formal XP utiliza un conjunto de metáforas. La metáfora es una simple historia compartida, que muestra como el sistemas funciona. Esta historia típicamente abarca un puñado de clases y paterns que moldean el esqueleto del sistema en construcción.

El tener la habilidad de adivinar como algo se llama resulta en ahorro de tiempo. Elige un sistema de nomenclatura para tus objetos con el cual todos puedan relacionarse sin necesidad de profundos conocimientos sobre el sistema. Por ejemplo, el sistema de planilla de sueldo de la Chrysler fue construido como una línea de producción.
Existen también metáforas del tipo naive que se basan en cosas cotidianas, pero frecuentemente no son útiles.

8.1 Spike solutions

Estos son programas muy simples que permiten explorar posibles soluciones. Ellos son usados cuando tenemos potenciales situaciones de riesgo en el proyecto, también son usadas para probar la validez de la metáfora.
La mayoría no serán parte del producto final, su meta es incrementar la confiabilidad de las estimaciones hechas para las historias de usuarios.

9 PLAN DE LANZAMIENTO

El plan de lanzamiento especifica exactamente que historias de usuarios serán implementadas para cada lanzamiento del sistema y especifica las fechas para dichos lanzamientos. Esto da al cliente un grupo de historias de usuario para incluir en el próximo plan de iteración.
El plan de lanzamiento solía ser llamado el cronograma de compromiso o "el juego planificado". El nombre fue cambiado para describir mas claramente su propósito y que fuese mas consistente con el plan de iteración.

Conceptos:

Los proyectos son cuantificados por cuatro variables relacionadas entre si:

- Alcance: Cuanto se debe hacer.
- Dinero: Cantos recursos (incluyendo personas) están disponibles.
- Tiempo: Cuando el proyecto o lanzamiento tiene que terminarse.
- Cantidad: Cuan bueno y que tan bien probado estará el sistema.

Si se hace variar a cualquiera de estos items esto se reflejará en el resto. En esencia, el cliente fija tres de estas variables y los desarrolladores ajustan la restante. Una de nuestras metas es lograr la mejor combinación de estas variables en términos de satisfacer las necesidades de nuestro cliente, y es por ello que necesitamos el plan.

Reunión de planeamiento del "Release":

Esta reunión es usada para crear el plan del lanzamiento, el cual estima los tiempos del proyecto. Este plan será revisado muchas veces a lo largo del proyecto. La esencia de esta reunión es estimar cada historia de usuario en términos de semanas ideales de programación (esto incluye las pruebas). Luego el cliente decide que historia es la mas importante o tiene la mayor prioridad para ser completada (Rango entre: Vital ; Importante ; Útil).
Juntos los desarrolladores y el cliente mueven las tarjetas de historias de usuarios sobre una gran mesa para crear un grupo de historias para ser implementadas en el primer (o próximo) lanzamiento. El plan debe ser negociado hasta que los desarrolladores, clientes y gerentes estén todos de acuerdo. En la práctica, el orden de las historias debe estar dado por su importancia y riesgo.

Velocidad del proyecto:
Es importante determinar cuantas historias pueden ser implementadas antes de una determinada fecha (tiempo) o cuanto llevará implementar un grupo de historias (alcance).
Cuando se planifica por tiempo se debe multiplicar el numero de iteraciones por la velocidad del proyecto para determinar cuantas historias de usuarios pueden ser completadas. Cuando se planifica por alcance se debe dividir el total de semanas para las historias de usuario estimadas por la velocidad del proyecto para determinar cuantas iteraciones faltan para que el lanzamiento este listo.

Cuando la velocidad del proyecto cambia dramáticamente se debe realizar una nueva reunión de planeamiento del "release" con el cliente para crear un nuevo plan de lanzamiento.

10 ITERACION

Al principio de cada iteración se realiza una reunión para hacer el plan de tareas programadas (cosas a ser hechas) para la iteración, esta se llama reunión de planificación de la iteración.
La planificación "Just-in-time" es un modo simple de mantenerse al día con los cambios de requerimientos del usuario.
Cada iteración no debe ser mas extensa que tres semanas, este es el latido del proyecto y debe mantenérselo fuerte y constante.
Al planificar cada iteración como si fuera la última, uno mantiene los plazos de entrega del producto.
Las historias de usuario[7] son elegidas (por el cliente) del plan de lanzamiento[9] en orden de las mas valiosas para el cliente primero. La cantidad de historias de usuarios depende de la velocidad del proyecto de la ultima iteración.
Luego las historias de usuario y pruebas de fallos son subdivididos en tareas de programación. Las tareas son escritas (en lenguaje técnico) en tarjetas indexadas. Ellas serán el plan detallado para la iteración.
Los desarrolladores que aceptan hacer las tareas estiman cuanto llevará completarlas. Es vital que los mismos desarrolladores acepten y estimen el tiempo que tomarán las tareas.
Cada tarea debe tomar **entre 1 a 3 días ideales de programación.** Las tareas de menos de 1 día deben ser agrupadas con otras y las de mas de 3 días subdivididas.
Ahora la velocidad del proyecto (factor entre el tiempo estimado y el real) es usada nuevamente para determinar si la iteración esta sobrecargada. La totalización de los días ideales de programación de las tareas planificadas no debe superar la velocidad de la iteración previa. Si la iteración esta sobrecargada, el cliente debe elegir historias a dejar fuera hasta otra iteración (**snow plowing**). Si la iteración esta subcargada otras historias pueden ser aceptadas. Se debe mantener la atención en la velocidad de proyecto y el "snow plowing". Es normal re-estimar todas las historias y **renegociar el plan de lanzamiento cada tres o cinco iteraciones**. Las tareas mas importantes para el cliente deben ser implementadas primero para asegurarlas.

El principal criterio es agregar lo que está siendo necesitado hoy, no agregar extra funcionalidades hasta que sean necesarias. El desarrollo iterativo puede agregar agilidad al proceso. El concepto de programación Just-in-time requiere solo planificar la actual iteración.

La fecha límite de la iteración debe ser tomada estrictamente, para ello el progreso también es seguido durante la iteración. Si existen síntomas de retraso se debe llamar a otra reunión de planificación de la iteración para volver a estimar y quitar algunas de las tareas.

Reuniones "de parados"
Estas reuniones se realizan a la misma hora cada día, en ellas participan todos los miembros estando de pie (para que la reunión no se prolongue). Cada miembro reporta cualquier problema, pero las soluciones no son discutidas. De este modo los miembros pueden pensar como resolver los problemas mientras trabajan.

11 UNIDAD DE PRUEBA

Durante la iteración y antes de comenzar a codificar cada tarea, esta debe ser traducida en pruebas de aceptación, esto se llama crear las unidades de prueba. Una tarea puede tener una o muchas pruebas de aceptación, lo que sea necesario para asegurar la funcionalidad. El tiempo que lleva crear las unidades de prueba y el código es similar al que lleva solo codificar sin escribir las pruebas antes. Al crear la unidad de prueba nos ayuda a realmente considerar que se necesita hacer. Luego el código es simple y conciso. Además, otros desarrolladores pueden aprender como funciona el código con solo examinar las pruebas.

Los marcos de trabajo (frameworks) de las unidades de prueba son herramientas de desarrollo que ayudan a formalizar los requerimientos, clarificar la arquitectura, escribir y debugear el código, integrar el código, optimizar y por supuesto, a probar.
JUnit (disponible en *http://www.junit.org*) se esta convirtiendo en el framework de unidades de prueba estándar para Java.

12 JUnit

"JUnit test cases" (casos de prueba) son clases de Java que contienen uno o mas métodos de unidades de prueba, y estas pruebas son agrupadas en suites de prueba. Se pueden correr las pruebas individualmente o la suite completa. Cada prueba en JUnit debe correr rápido y fácilmente, preferentemente con un único comando. La capacidad de correr todas las pruebas mediante un único comando es casi un requerimiento para decir que el proyecto esta haciendo XP.

Aquí examinamos un ejemplo:

```
import junit.framework.TestCase;

public class TestSupplier extends TestCase {

/*@param testMethodName el nombre del método de
prueba a ejecutar*/

public TestSupplier (String testMethodName){
        super(testMethodName);
}
/*Unidad de prueba para verificar el correcto formateo
del nombre.*/

public void testGetFullName(){
    Supplier s =new Supplier("Juan","Perez");
    assertEquals("Juan Perez",s.getFullName());
}
/*Unidad de prueba para verificar que los NULL son
manejados correctamente.*/

public void testNullsInName(){
    Supplier s =new Supplier(null,"Perez");
    assertEquals("?Perez",s.getFullName());
/*este código es solamente ejecutado si el assertEquals
es verdadero*/
    s =new Supplier("Lopez",null);
    assertEquals("Lopez ?",s.getFullName());
    }
}
```

Los métodos de prueba siguen este formato:
public void test<*something*>() [throws **xxxException**]

La convención de nombres sirve para hacer que el código sea mas mantenible y facilitar la automatización. Las pruebas en este ejemplo tienen el prefijo "Test ", resultando en nombre tal como: TestSupplier que corresponde a la clase Supplier. Al escribir un caso de prueba por clase se hace mucho mas fácil el mirar dentro del directorio y saber que pruebas están disponibles. "Test" puede ser usado como prefijo o sufijo.
Las pruebas pueden lanzar cualquier subclase de java.lang.Throwable. Cuando esto pasa, JUnit atrapa la excepción y reporta un error en la prueba. Luego continua ejecutando los siguiente métodos de prueba.
junit.framework.TestCase es la clase base de todos los casos de prueba.

Cada unidad de prueba utiliza varios métodos del tipo **assertXXX()** para realizar las pruebas::

assertEquals("Juan Perez",s.getFullName());

Este método confirma si estos dos argumentos son iguales. Si lo son la prueba es superada. De otro modo un fallo de prueba es reportado y el resto del método de prueba es pasado por alto. JUnit continua ejecutando otro método de prueba.

*"Una **falla** de prueba ocurre cuando una sentencia assertXXX() falla. Un* **error** *de prueba ocurre cuando una unidad de prueba lanza una excepción."*

Resumen del método Assert

assertEquals()	Compara dos valores. La prueba pasa si los valores son iguales.
assertFalse()	Evalúa una expresión booleana. La prueba pasa si la expresión es falsa.
assertNotNull()	Compara un objeto referencia con null . La prueba pasa si la referencia no es null.
assertNotSame()	Compara direcciones de memoria de dos objetos. La prueba pasa si ambos referencian a distintos objetos.
assertNull()	Compara una referencia a un objeto con null . La prueba pasa si la referencia es null .
assertSame()	Compara direcciones de memoria de dos objetos. La prueba pasa si ambas se refieren al mismo objecto.
assertTrue()	Evalúa una expresión booleana. La prueba pasa si la expresión es verdades.
fail()	Provoca que la prueba falle. Esto es común en los manejos de excepciones.

Todos los métodos aceptan un primer argumento opcional. Cuando es especificado, este argumento provee un mensaje descriptivo a ser mostrado cuando la prueba falla.

assertEquals("El método getFullName no esta funcionando bien.","Juan Perez",s.getFullName());

Modo grafico y de texto:

JUnit puede correr pruebas en modo texto (java junit.textui.TestRunner <*test unit name*>) o en modo gráfico (java junit.swingui.TestRunner <*test unit name*>)

El modo texto es mas rápido, es bueno para correr pruebas de partes. Las pruebas graficas son mas interesantes de correr y hacen mas fácil el análisis de los resultados de pruebas masivas.

Un trozo de funcionalidad a la vez:

Cada unidad de prueba debe chequear un trozo específico de funcionalidad. No se debe combinar múltiples pruebas no relacionadas en un único método testXXX(). Si se hace así y el primer assertXXX() falla, el resto de la prueba no será ejecutada y no se sabrá el resultado de las otras pruebas. Un método de prueba puede contener mas de un assertXXX() pero ellos todos deben ser parte de una única funcionalidad.

Métodos setUp y tearDown:

JUnit crea una nueva instancia para cada método en la unidad de prueba.

Luego de hacer esto Junit ejecuta el método **setUp**(), luego el método de prueba es ejecutado y finalmente se ejecuta el método **tearDown**(). Estos dos métodos pueden ser usados para ejecutar instrucciones compartidas que todos los métodos de prueba ejecutan al principio y al final. De esta forma se evita duplicar el código en cada método.

Suites de pruebas (TestSuite):

JUnit permite organizar múltiples pruebas dentro de una suite (junit.framework. TestSuite), todos los cuales se ejecutan a la vez.

13 CODIFICACION

Durante la iteración tendemos a agregar funcionalidad ahora, en cambio de esperar, porque hemos visto de que manera exactamente agregarlas o porque ésta hará al sistema mucho mejor y parece que es mas rápido agregarla ahora. Pero debemos evaluar constantemente si es que la necesitamos ahora.

La extra funcionalidad siempre nos retrasará y malgastará recursos. La concentración tiene que estar en lo que esta planeado para hoy solamente.

Los estándares para nombres y comentarios permitirán que logremos la **propiedad colectiva del código**. Esto permite a cualquier desarrollador fácilmente entender el código escrito por cualquier otro miembro del equipo.

Programación en pareja:
La programación en pareja (Pair programming) es el método de programación recomendado por XP. Es necesario porque compensa la falencia de ciertas practicas en XP: Como el diseño up-front y la documentación permanente. Codificar en pareja puede ser vista como una práctica compensatoria que ayuda a que los programadores (corajudamente) construyan el diseño mientras codifican. Durante la programación en pareja un programador esta pensando si el diseño funcionará, sobre las pruebas, o sobre modos de simplificar el código mientras que el otro escribe el código (codifica). Los roles de los dos programadores pueden cambiar frecuentemente y las parejas tienen que rotarse con frecuencia.

Este **experimento** sobre programación en pareja fue dirigido por Laurie Williams Universidad de Nord-Carolina (North Carolina State University) EE.UU.

*Hicimos un experimento con 42 programadores seniors en la Universidad de Utah. Catorce de ellos trabajaron solos. El resto trabajaron programando en parejas. Todos los estudiantes completaron las mismas tareas. Lo sorprendente es que los que programaron en parejas no costaron el doble! En su primer tarea las parejas costaron **60% mas** que los que programaron solos. En la segunda tarea, ya se habían acostumbrado a esto de programar en pareja. Las parejas costaron solo **20% mas** que los solitarios. Para la tercera tarea, las parejas solo costaron **10% mas** – entonces si un solitario tarda 10 horas en una tarea, la pareja trabaja en ella 5 horas y 15 minutos.*

***En todos los casos, las parejas pasaron 15% mas de los casos de prueba pos-desarrollo.** El 90% dijo disfrutar la programación mas y que se sintieron mas confiables en su trabajo, haciéndolo de a dos.*
Como un pos-experimento, todos los estudiantes trabajaron individualmente para completar una tarea. Un estudiante al volver a trabajar en solitario dijo: "Sin mi pareja, siento que perdí la mitad de mi cerebro".

El problema con este estudio en particular es que se realizo en una universidad, utilizando estudiantes, y no con programadores profesionales.

En los proyectos reales los problemas típicos que surgen son:
• Dinámica social.
• Falta de privacidad.
• Falta de "tiempo tranquilo para pensar".

14 LANZAMIENTOS
Cuando cada iteración termina un pequeño lanzamiento del sistema, este es presentado al cliente por el equipo de desarrollo. Estas funcionalidades, en forma incremental van formando el producto final.

Estos lanzamientos son críticos para obtener valioso feedback a tiempo para que tenga impacto en el desarrollo del sistema. Si toma mucho tiempo introducir cierta característica al sistema se tendrá menos tiempo luego para corregirla.

Refactoring (Refabricación):
El código es reestructurado para mejorar la estructura y la performance. *Refactoring* se hace antes que las capacidades nuevas sean agregadas, no mientras están siendo agregadas, de esta forma nos aseguramos que el *refactoring* no esta rompiendo el código.

15 CONCLUSIONES

Extreme Programming [XP] (programación extrema) es una técnica de desarrollo diferente para lidiar con un medioambiente inestable y riesgoso. Su estructura dinámica nos permite modificar e incluir nuevos diseños al momentos de la codificación.
Sin embargo, muchas prácticas de XP son distintas en la teoría y en la práctica.
Por ejemplo, los aspectos sociales de la programación en pareja pueden ser muy difíciles de manejar.

Comparándolo con RUP/UML:

XP tiene algunas similitudes con el Proceso Unificado de Rational (Rational Unified Process-RUP) y en algún modo puede ser considerado una versión abreviada del RUP con modificaciones.
También ocurre que alguna metodologías de XP pueden ser aplicadas sin aplicar la metodología completa (Ejemplo: "programación en pareja").
Sin embargo, XP tiene muchas diferencias y muchos de sus métodos son interdependientes. Esto significa que si algunas técnicas de XP son dejados fuera, la metodología completa pierde sentido.
Otra importante diferencia entre XP y RUP/UML es la que XP no utiliza representaciones gráficas (diagramas) para el análisis y diseño de los procesos.
Basado en los métodos que utiliza XP, se espera que el código producido sea de alta calidad.
También se basa fuertemente en el concepto de refactoring (refabricar) el código para aumentar su productibilidad.
Generalmente XP introduce constantes correcciones de curso al proyecto, en cambio RUP/UML es planificado con anterioridad.
Algunos expertos dicen que XP trabaja mejor con proyectos de un tamaño pequeño a mediano y es mejor utilizar UML y RUP para proyectos de mayor envergadura. Se dice que XP trabaja mejor con proyectos de alto riesgo.

Criticas comunes a XP:

- Conflictos interpersonales (Programación en parejas)
- Ambiente incómodo de trabajo (Lugar de trabajo comunitario y sin divisiones)
- Pobre documentación del sistema.

Halagos comunes a XP:

- Cambio dinámico de las especificaciones.
- Ahorro en tiempo y dinero (menor costo de diseño y documentación innecesaria)
- Prioriza la calidad (Los casos de prueba son escritos antes que el código).
- Feedback continuo con el cliente.

REFERENCIAS

- Java extreme programming cookbook
By Eric M. Burke and Brian M. Coyner [O'Reilly]
- Extreme Programming Pocket Guide
By Chromatic [O'Reilly]
- Extreme Programming Refactored
By Matt Stephens and Doug Rosenberg [Apress]
- The Costs and Benefits of Pair Programming.
By Alistair Cockburn & Laurie Williams
http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF
- http://www.extremeprogramming.org
- http://www.onlamp.com/
- http://www.serverworldmagazine.com/webpapers/
- http://clabs.org/papers.htm
- http://www.xprogramming.com
- http://pairprogramming.com
- http://www.junit.org