Programación eXtrema y Software Libre

Gregorio Robles
Universidad Rey Juan Carlos

grex@scouts-es.org
Jorge Ferrer
Universidad Politécnica de Madrid

jferrer@jorgeferrer.com

Historial de revisiones
Revisión 2.0 - versión V Congreso Hi-10 de octubre de 2002
spalinux, Octubre 2002

La programación extrema es una metodología de desarrollo ligera basada en una serie de valores y una docena de prácticas de, llamémoslas así, buenas maneras que propician un aumento en la productividad a la hora de generar software. Por otro lado, el software libre es un movimiento nacido de la idea de que los usuarios tienen una serie de derechos sobre el software que permiten modificarlo, adaptarlo y redistribuirlo. Estas características han hecho que el desarrollo de software libre haya desembocado en unos métodos de desarrollo informales similares a los que se pregonan en la programación extrema y que serán presentados, estudiados y comparados en este artículo. Se hará especial énfasis en las diferencias que hay entre los dos métodos y lo que puede aprender el software libre de la programación extrema.

Tabla de contenidos

1. Sobre este documento

En este artículo se hará una introducción tanto de la programación extrema como del software libre. Aunque serán dos presentaciones bastante amplias, no se pretende llegar a un gran nivel de detalle. Existen multitud de artículos y libros sobre programación extrema y software libre que tratan ambos temas de una manera mucho más extensa; alguno de los artículos y libros se pueden encontrar en el apartado dedicado a las referencias y direcciones de interés al final de este documento.

Después de dar a conocer en qué consisten la programación extrema y el software libre se procederá a compararlos, a ver qué prácticas son comunes, así como en qué aspectos difieren o serían necesarias modificaciones para poder llegar a compaginarlas. La comparación desembocará en un último punto en el que se muestran las conclusiones que el autor ha sacado de la elaboración del estudio.

2. La Programación Extrema

La programación extrema se basa en una serie de reglas y principios que se han ido gestando a lo largo de toda la historia de la ingeniería del software. Usadas conjuntamente proporcionan una nueva metodología de desarrollo software que se puede englobar dentro de las metodologías ligeras, que son aquéllas en la que se da prioridad a las tareas que dan resultados directos y que reducen la burocracia que hay alrededor tanto como sea posible (pero no más) [Fowler]. La programación extrema, dentro de las metodologías ágiles, se puede clasificar dentro de las evolutivas [Harrison].

Una de las características de eXtreme Programming es que muchos de, si no todos, sus ingredientes son de sobra conocidos dentro de la rama de la ingeniería del software desde hace tiempo, incluso desde sus comienzos. Los autores de han seleccionado los que han considerados como los mejores y han profundizado en sus relaciones y en cómo se refuerzan unos a otros. El resultado ha sido una metodología única y compacta. Por eso, aunque se pueda

alegar que la programación extrema no se base en principios nada nuevos, se ha de aclarar que, en conjunto, es una nueva forma de ver el desarrollo de software.

Aunque, como ya se ha comentado, la programación extrema se basa en unos valores, unos principios fundamentales y unas prácticas, en este artículo no se van a enumerar así de primeras, ya que el autor considera que no es la mejor forma de presentarlos. Los principios y prácticas no se han hecho a priori o porque sí, sino que tienen un porqué a partir de una forma global de desarrollar software que, al menos en teoría, parece ser más eficiente.

Por tanto, en este artículo se presentará la programación extrema desde un punto de vista práctico para luego dar paso a enunciar los valores y principios que se han extraído y las prácticas que hacen que se lleven a buen fin. La idea es seguir que el lector pueda seguir en los siguientes párrafos un proceso de desarrollo extremo tal y como debería darse en un equipo de desarrollo que siguiera la metodología XP. De esta forma se irán detallando y explicando las diferentes técnicas utilizadas, así como su razón de ser.

Una vez que hayamos visto el proceso de desarrollo extremo, los valores, principios y prácticas serán evidentes y no requerirán mucho detenimiento.

2.1. El proceso de desarrollo extremo

La programación extrema parte del caso habitual de una compañía que desarrolla software, generalmente software a medida, en la que hay diferentes roles: un equipo de gestión, un equipo de desarrolladores y los clientes. La relación con el cliente es totalmente diferente a lo que se ha venido haciendo en las metodologías tradicionales que se basan fundamentalmente en una fase de captura de requisitos previa al desarrollo y una fase de validación posterior al mismo.

2.1.1. Interacción con el cliente

En la programación extrema al cliente no sólo se le pide que apoye al equipo de desarrollo, en realidad podríamos decir que es parte de él. Su importancia es capital a la hora de abordar las historias de los usuarios y las reuniones de planificación, como veremos más adelante. Además, será tarea suya realimentar al equipo de desarrolladores después de cada iteración con los problemas con los que se ha encontrado, mostrando sus prioridades, expresando sus sensaciones... Existirán métodos como pruebas de aceptación que ayudarán a que la labor del cliente sea lo más fructífera posible.

En resumen, el cliente se encuentra mucho más cercano al proceso de desarrollo. Se elimina la fase inicial de captura de requisitos y se permite que éstos se vayan definiendo de una forma ordenada durante el tiempo que dura el proyecto. El cliente puede cambiar de opinión sobre la marcha y a cambio debe encontrarse siempre disponible para resolver dudas del equipo de desarrollo y para detallar los requisitos especificados cuando sea necesario.

El proceso de captura de requisitos de XP gira entorno a una lista de características que el cliente desea que existan en el sistema final. Cada una de estas características recibe el nombre de historias de usuarios y su definición consta de dos fases:

En la primera fase el cliente describe con sus propias palabras las características y el responsable del equipo de desarrollo le informa de la dificultad técnica de cada una de ellas y por lo tanto de su coste. A través del diálogo resultante el cliente deja por escrito un conjunto de historias y las ordena en función de la prioridad que tienen para él. En este momento ya es posible definir unos hitos y unas fechas aproximadas para ellos.

La segunda fase consiste en coger las primeras historias que serán implementadas (primera iteración) y dividirlas en las tareas necesarias para llevarlas a cabo. El cliente también participa, pero hay más peso del equipo de desarrollo, que dará como resultado una planificación más exacta. En cada iteración se repetirá esta segunda fase para las historias planificadas para ella.

Este proceso es una de las principales diferencias con las metodologías tradicionales. Aunque las historias de usuarios guardan cierta relación con otras técnicas como los casos de uso de UML, su proceso de creación es muy diferente. En lo que al cliente se refiere no se le exige que especifique exactamente lo que quiere al principio con un documento de requisitos de usuario. La parte que se mantiene con este documento es que es el cliente el que tiene que escribir lo que quiere, no se permite que alguien del equipo de desarrolladores lo escriba por él.

Como se ha comentado, son los desarrolladores los que se encargan de catalogar las historias de los usuarios y asignarles una duración. Para ello se sigue una norma simple: las historias de usuarios deberían poder ser abordables en un espacio de tiempo de entre una y tres semanas de programación ideal. Historias de los usuarios que requieran menos tiempo de implementación son agrupadas, mientras que aquéllas que necesiten más tiempo deben ser modificadas o divididas. Una semana de programación ideal es una semana (cinco días de trabajo) de desarrollo por parte de un desarrollador sin interferencias de otras partes del proyecto. Al hacer la planificación se aplica un factor de corrección medido de proyectos anteriores para ajustar este tiempo ideal al real.

Las historias de los usuarios se plasmarán en tarjetas, lo que facilitará que el cliente pueda especificar la importancia relativa entre las diferentes historias de usuario, así como la tarea de los desarrolladores que podrán catalogarlas convenientemente. El formato de tarjeta además es muy provechoso a la hora de realizar pruebas de aceptación.

2.1.2. Planificación del proyecto

Es probablemente en este punto donde nos debamos enfrentar a la planificación de entregas (release planning) donde planificaremos las distintas iteraciones. Para ello existen una serie de reglas que hay que seguir para que las tres partes implicadas en este proceso (equipo de gestión, equipo de desarrollo y cliente) tengan voz y se sientan parte de la decisión tomada, que al fin y al cabo debe contentar a todos.

La planificación debe de seguir unas ciertas premisas. La primordial es que las entregas se hagan cuanto antes y que con cada iteración el cliente reciba una nueva versión. Cuanto más tiempo se tarde en introducir una parte esencial, menos tiempo habrá para trabajar en ella posteriormente. Se aconsejan muchas entregas y muy frecuentes. De esta forma, un error en una parte esencial del sistema se encontrará pronto y, por tanto, se podrá arreglar antes.

Sin embargo, los requisitos anteriores en cuanto a la planificación no deben suponer horas extra para el equipo de desarrollo. El argumento que se esboza es que lo que se trabaja de más un día, se deja de trabajar al siguiente.

Diversas prácticas como las pruebas unitarias, la integración continua o el juego de la planificación permiten eliminar los principales motivos por los que suele ser necesario trabajar muchas horas extra.

Pero lo mejor de todo es que a la hora de planificar uno se puede equivocar. Es más, todos sabemos que lo común es equivocarse y por ello la metodología ya tiene previsto mecanismos de revisión. Por tanto, es normal que cada 3 a 5 iteraciones se tengan que revisar las historias de los usuarios y renegociar nuevamente la planificación.

Hemos visto que al principio del proyecto se hace una planificación en iteraciones que debe ser retocada al cabo de unas cuantas iteraciones. A esto hay que añadir que en cada iteración también hay que realizar la planificación de la misma, lo que ha venido a llamarse planificación iterativa. En la planificación iterativa se especifican las historias de los usuarios cuya implementación se considera primordial y se añaden aquéllas que no han pasado las pruebas de aceptación de anteriores iteraciones. La planificación de una iteración también hace uso de tarjetas en las que se escribirán tareas, que durarán entre uno y tres días (la duración la deben decidir los propios desarrolladores).

Es por eso, que el diseño que seguimos se puede calificar de continuo. Como vemos añade agilidad al proceso de desarrollo y evita mirar demasiado adelante e implementar tareas que no estén programadas (algo que se ha venido a llamar programación just-in-time). También es cierto que no hay nada que pueda evitar que los retrasos se acumulen y como ya decía Brooks en esos casos añadir gente a un proyecto retrasado, sólo lo retrasa más.

A raíz de lo anterior, podemos entender el siguiente consejo: optimiza al final. El eslogan subyacente es "make it work, make it right and then make it fast" (haz que funcione, hazlo bien y entonces haz que sea rápido). Y es que nunca se sabe a priori dónde puede estar el verdadero cuello de botella, así que lo mejor es no añadir funcionalidad demasiado temprano y concentrarnos completamente en lo que es necesario hoy. Para la optimización siempre habrá tiempo después cuando sea prioritaria, si es que de verdad llega a serlo.

La planificación en iteraciones y el diseño iterativo dan pie a una práctica poco común en el desarrollo tradicional que son las discusiones diarias de pie. De esta forma, se fomenta la comunicación, ya que los desarrolladores cuentan con tiempo para hablar de los problemas a los que se enfrentan y cómo van con su(s) tarea(s), a la vez que su carácter informal las hacen agradables y, sobre todo, no se alargan.

2.1.3. Diseño, desarrollo y pruebas

El desarrollo es la pieza clave de todo el proceso de programación extrema. Todas las tareas tienen como objetivo que se desarrollo a la máxima velocidad, sin interrupciones y siempre en la dirección correcta.

También se otorga una gran importancia al diseño y establece que éste debe ser revisado y mejorado de forma continua según se van añadiendo funcionalidades al sistema. Esto se contrapone a la práctica conocida como "Gran diseño previo" habitual en otras metodologías. Los autores de XP opinan que este enfoque es incorrecto dado que a priori no se tiene toda la información suficiente para diseñar todo el sistema y se limita la posibilidad del cliente de cambiar de opinión respecto a las funcionalidades deseadas. Como veremos a continuación a cambio se establecen los mecanismos para ir remodelando el diseño de forma flexible durante todo el desarrollo.

La clave del proceso de desarrollo de XP es la comunicación. La gran mayoría de los problemas en los proyectos de desarrollo son provocados por falta de comunicación en el equipo, así que se pone un gran énfasis en facilitar que la información fluya lo más eficientemente posible.

Es en este punto donde entra uno de los términos estrella de la programación extrema: la metáfora. El principal objetivo de la metáfora es mejorar la comunicación entre los todos integrantes del equipo al crear una visión global y común del sistema que se pretende desarrollar. La metáfora debe estar expresada en términos conocidos para los integrantes del grupo, por ejemplo comparando lo que se va a desarrollar con algo que se puede encontrar en la vida real. Aunque también se incluye información sobre las principales clases y patrones que se usarán en el sistema.

Un apoyo a la metáfora a lo largo del proyecto es una correcta elección y comunicación de los nombres que se escojan durante el proyecto para los módulos, sistemas, clases, métodos, etc. Nombres bien puestos implican claridad, reusabilidad y simplicidad... tres conceptos a los que XP otorga una gran importancia.

Aunque en general el diseño es realizado por los propios desarrolladores en ocasiones se reúnen aquellos con más experiencia o incluso se involucra al cliente para diseñar las partes más complejas. En estas reuniones se emplean un tipo de tarjetas denominadas CRC (Class, Responsabilities and Collaboration - Clases, Responsabilidades y Colaboración) cuyo objetivo es facilitar la comunicación y documentar los resultados. Para cada clase identificada se rellenará una tarjeta de este tipo y se especificará su finalidad así como otras clases con las que interaccione. Las tarjetas CRC son una buena forma de cambiar de la programación estructurada a una filosofía orientada a objetos. Aunque los grandes gurús de la programación extrema sostienen que bien hechas suelen hacer el diseño obvio, recomiendan hacer sesiones CRC en caso de que el sistema que se pretenda crear tenga un grado de complejidad grande. Este tipo de sesiones es una simulación, tarjetas CRC en mano, de las interacciones entre los diferentes objetos que puede realizar el equipo de desarrollo.

Como ya hemos visto con anterioridad, uno de los principios de la programación extrema es la simplicidad. El diseño debe ser lo más simple posible, pero no más simple. El paradigma KISS ("Keep It Small and Simple" para unos o "Keep it Simple, Stupid" para otros) se lleva hasta las últimas consecuencias. Por ejemplo, se hace énfasis en no añadir funcionalidad nunca antes de lo necesario, por las sencillas razones de que probablemente ahora mismo no sea lo más prioritario o porque quizás nunca llegue a ser necesaria.

Supongamos que ya hemos planificado y dividido en tareas, como se ha comentado en los párrafos anteriores. Lo lógico sería empezar ya a codificar. Pues no. Nos encontramos con otro de los puntos clave de la programación extrema (y que sí es innovador en ella): las pruebas unitarias se implementan a la vez hay que el código de producción. De hecho cada vez que se va a implementar una pequeña parte se escribe una prueba sencilla y luego el código suficiente para que la pase. Cuando la haya pasado se repite el proceso con la siguiente parte. Aunque intuitivamente esto parezca contraproducente, a la larga hará que la generación de código se acelere. Los creadores de la programación extrema argumentan que encontrar un error puede llegar a ser cien veces más caro que realizar las pruebas unitarias. La idea, en definitiva, se resumen en la siguiente frase: "Todo código que pueda fallar debe tener una prueba". Además, hay que tener en cuenta que se hacen una vez y luego se pueden reutilizar multitud de veces, incluso por otros desarrolladores que desconocen los entresijos de esa parte o de todo el sistema, por lo que permiten compartir código (otra de las prácticas que permiten acelerar el desarrollo tal y como se verá más adelante).

Esta forma de usar las pruebas unitarias ayuda a priorizar y comprobar la evolución del desarrollo y que ofrecen realimentación inmediata. Ya no hay imprescindibles dos equipos diferenciados que desarrollan y prueban cada uno por su cuenta. Ahora el ciclo se basa en implementar una prueba unitaria, codificar la solución y pasar la prueba, con lo que se consigue un código simple y funcional de manera bastante rápida. Por eso es importante que las pruebas se pasen siempre al 100% [Jeffries].

Hay mucha literatura sobre las pruebas unitarias [Beck2][Gamma][Gamma2]. La mayoría de los autores están de acuerdo en que cuanto más difícil sea implementar una prueba, más necesarias son. Algunos incluso dicen que entonces quizás sea porque lo que se intenta probar no es lo suficientemente sencillo y ha de rediseñarse. En cuanto a herramientas para realizar tests unitarios, existen varias para los diferentes lenguajes, lo que hace que su ejecución sea simple y, sobre todo, automática [Impl].

Las pruebas unitarias no se han de confundir con las pruebas de aceptación que han sido mencionadas con anterioridad. Éstas últimas son pruebas realizadas por el cliente o por el usuario final para constatar que el sistema hace realmente lo que él quiere. En caso de que existan fallos, debe especificar la prioridad en que deben ser solucionados los diferentes problemas encontrados. Este tipo de pruebas son pruebas de caja negra y se hacen contra las historias de los usuarios. Se suele tender a que sean parcialmente automáticos y que los resultados sean públicos.

Es hora entonces de ampliar el ciclo de creación de pruebas unitarias, codificación, paso de las pruebas y añadirle un paso más: la integración. La programación extrema viene a perseguir lo que se ha venido a llamar integración continua. De esta forma, haciéndolo cada vez con pequeños fragmentos de código, se evita la gran integración final. Las ventajas de este enfoque es que permite la realización de pruebas completas y la pronta detección de problemas de incompatibilidad. Además, ya no será necesario un equipo independiente de integración que haga uso del mágico pegamento al enfrentarse a problemas de divergencias y fragmentación de código.

En todo desarrollo de programación extrema debería existir, por tanto, una versión siempre integrada (incluso se puede asegurar su existencia mediante cerrojos - locks). La sincronización por parte de los desarrolladores con el repositorio central debe darse como mínimo una vez al día, de manera que los cambios siempre se realicen sobre la última versión. De esta forma nos podemos asegurar de que las modificaciones que hacemos no se estén haciendo sobre una versión obsoleta.

Quizás el lector se haya sorprendido con la última afirmación y pensará que la probabilidad de encontrarse una versión de código obsoleta (más antigua que el código actual) es muy baja. En cierto modo, esto es cierto en las metodologías tradicionales, pero en la programación extrema no: nos topamos con la importancia de refactorizar [Fowler3]. Refactorizar consiste básicamente en quitar redundancia, eliminar funcionalidad que no se usa o "rejuvenecer" diseños viejos. Tiene su justificación principal en que el código no sólo tiene porque funcionar, también debe ser simple. Esto hace que a la larga refactorizar ahorre mucho tiempo y suponga un incremento de calidad. Por cierto, tal es el énfasis que se pone en la refactorización que de la misma no se libran ni las pruebas unitarias.

Como uno de los objetivos de la programación extrema es que cualquier miembro del equipo de desarrollo puede mejorar cualquier parte del sistema, llegamos fácilmente a la conclusión de que se busca que el código sea de todos. Cualquier desarrollador puede realizar cambios, corregir erratas o refactorizar en cualquier momento. Para eso, entre otras cosas, tenemos el colchón de las pruebas unitarias por si nos equivocamos. Además, es una forma coherente

de plasmar que todo el equipo es responsable del sistema en su conjunto y de que no haya feudos personales. En consecuencia, un desarrollador que deje el proyecto (algo habitual, por otra parte) no tiene por qué convertirse en un hecho catastrófico. El mejor método para conseguir que el código sea de todos es seguir unos estándares de codificación consistentes, de manera que la lectura (y refactorización) por parte del resto del equipo de desarrollo se facilite al máximo.

Para terminar esta, ya extensa, descripción de un proceso de desarrollo de programación extrema, he dejado una de sus joyas para el final. El proceso de desarrollo no lo va a hacer un desarrollador en solitario, sino siempre con otra persona, algo que se ha venido a llamar programación por parejas. Una pareja de desarrolladores debe compartir ordenador, teclado y ratón. El principal objetivo es realizar de forma continua y sin parar el desarrollo una revisión de diseño y de código. Las parejas deben ir rotando de forma periódica para hacer que el conocimiento del sistema se vaya difundiendo por el equipo (facilitándose que el código sea de todos), a la vez que se fomentan el entrenamiento cruzado[Jeffries2]. Existen estudios que concluyen que esta práctica es eficaz en la práctica justificándola con aspectos psicológicos y sociológicos [Cockburn]. Con este apoyo los gurús de la programación extrema no dudan en afirmar que dos personas trabajando conjuntamente en pareja generan en cantidad el mismo código (o mejor dicho, la misma funcionalidad) que dos personas por separado, pero de mayor calidad. Sin embargo esta es la práctica que más reticencias provoca por parte de jefes y de los propios programadores.

2.2. Valores de la programación extrema

El proceso de desarrollo descrito en la sección anterior está fundamentado en una serie de valores y principios que lo guían. Los valores representan aquellos aspectos que los autores de XP han considerado como fundamentales para garantizar el éxito de un proyecto de desarrollo de software. Los cuatro valores de XP son:

1. comunicación,
2. simplicidad,
3. realimentación y
4. coraje

Los partidarios de la programación extrema dicen que son los necesarios para conseguir diseños y códigos simples, métodos eficientes de desarrollo software y clientes contentos. Los valores deben ser intrínsecos al equipo de desarrollo.

De los cuatro valores, quizás el que llame más la atención es el de coraje. Detrás de este valor encontramos el lema "si funciona, mejóralo", que choca con la práctica habitual de no tocar algo que funciona, por si acaso. Aunque también es cierto que tenemos las pruebas unitarias, de modo que no se pide a los desarrolladores una heroicidad, sino sólo coraje.

Algunas voces, además, añaden un quinto valor: la humildad. Y es que con la que compartición de código, la refactorización y el trabajo en equipo tan estrecho una buena dosis de humildad siempre será de agradecer.

2.3. Principios de la programación extrema

Los principios fundamentales se apoyan en los valores y también son cuatro. Se busca

1. realimentación veloz,
2. modificaciones incrementales,
3. trabajo de calidad y
4. asunción de simplicidad.

Los principios suponen un puente entre los valores (algo intrínseco al equipo de desarrollo) y las prácticas, que se verán a continuación, y que están más ligadas a las técnicas que se han de seguir.

2.4. Prácticas de la programación extrema

Por su parte, las prácticas son las siguientes:

1. El juego de la planificación (the planning game)
2. Pequeñas entregas (small releases)
3. Metáfora (metaphor)
4. Diseño simple (simple design)
5. Pruebas (testing)

6. Refactorización (refactoring)
7. Programación por parejas (pair programming)
8. Propiedad colectiva (collective ownership)
9. Integración continua (continous integration)
10. 40 horas semanales (40-hour week)
11. Cliente en casa (on-site costumer)
12. Estándares de codificación (coding standards)

No es fácil aplicar una nueva metodología en un equipo de desarrollo ya que obliga a aprender una nueva forma de trabajar. También obliga a abandonar cómo se hacían las cosas antes, que aunque no fuera la mejor forma posible ya se conocía. XP ha sido adoptado por un gran número de equipos en los últimos años y de sus experiencias se ha extraído una conclusión sencilla: es mejor empezar a hacer XP gradualmente.

El proceso que recomiendan los autores de XP es el siguiente: identifica el principal problema del proceso de desarrollo actual. Escoge la práctica que ayuda a resolver ese problema y aplícala. Cuando ese haya dejado de ser un problema, escoge el siguiente. En realidad se recomienda que se apliquen las prácticas de dos en dos. El objetivo es que las prácticas de XP se apoyan unas a otras y por tanto dos prácticas aportan más que la suma de ambas y por tanto es más fácil comprobar los resultados.

El objetivo final debe ser aplicar todas las prácticas, ya que representan un conjunto completo, "si no las aplicas todas no estás haciendo eXtreme Programming".

2.5. Conclusión

Según Kent Beck, "la programación extrema es una forma ligera, eficiente, flexible, predecible, científica y divertida de generar software" (Kent Beck, Extreme Programming Explained). Ahí es nada. Esta metodología ha surgido desde la experiencia, como una forma de resolver los problemas encontrados en los procesos de desarrollo software en los que se han visto involucrados sus autores. Este tipo de desarrollos eran en general de creación de software a la medida del cliente y hay numerosas opiniones que relatan el éxito de esta metodología en este ámbito. Queda por ver si es posible aplicar sus ideas también en procesos de desarrollo muy diferentes, como el seguido por la comunidad del software libre.

3. El Software Libre

En realidad, la definición de software libre es un concepto legalista, ya que la única diferencia entre el software propietario y el software libre es precisamente su licencia [Stallman]. Esto no es del todo cierto, porque se ha venido

constatando en la última década que esta distinción tiene unos efectos secundarios que afectan a la manera en la que se entiende y en la que se genera el propio software. Podemos ver que la licencia condiciona la manera de buscar una forma eficiente de generación y difusión del software.

3.1. El modelo de desarrollo de software libre

Eric S. Raymond, gurú del software libre, se dio cuenta de ello en 1997 y en uno de sus famosos escritos catalogó el modelo de desarrollo de algunos (no todos) proyectos de software libre como el modelo de bazar [Raymond][Bezroukov]. Para Raymond la metodología tradicional se podía comparar con la construcción de catedrales donde existía un gran arquitecto que hacía el diseño y el reparto de tareas, para que posteriormente un conjunto de operarios y peones realizaran las operaciones pertinentes. En el modelo de bazar, por el contrario, no existe ese orden tan estricto, sino que se asemeja más bien al caos que se forma en un bazar oriental. La manera de interactuar entre los diferentes actores en el caso del bazar no está controlada por ningún tipo de personas ni entidades, sino que existe una enorme cantidad de intereses y de intercambios de diferentes tipos (lo que los economistas que han estudiado el software libre han llamado transacciones [Ghosh][Lerner]).

Eric Raymond achacó la creación de proyectos de software libre a necesidades e intereses particulares de los propios desarrolladores, que lanzan una aplicación para resolver su problema y el de otras personas en idéntica situación. No es difícil imaginar entonces que en cada proyecto exista la figura del líder, normalmente asociada a su fundador o a uno de sus principales promotores. El líder del proyecto no necesita ser una persona dotada meramente de conocimientos técnicos, sino que debe tener habilidad para coordinar y motivar a todo aquél que esté interesado en unirse al proyecto realizando aportaciones. Esto es así, porque el origen del rediseño suele situarse en los propios usuarios, que se revelan como los que hacen el trabajo de depuración, probablemente el proceso más tedioso en la generación de software. Se da el hecho de que una cantidad grande de usuarios permite procesos de depuración en paralelo y que, en caso de existir errores, su corrección puede darse de forma distribuida, ya que el usuario que encuentra una errata no tiene por qué ser el usuario/desarrollador que la corrige.

3.2. Herramientas de desarrollo

Para poder llevar a la práctica de manera eficaz el que los proyectos sean lo más abiertos que se pueda, el modelo de bazar ha supuesto la elaboración y perfección de numerosas herramientas, incluso de sitios centralizados que intentan englobar todo el proceso de desarrollo, como son SourceForge y sus clones (BerliOS o Savannah). Las herramientas son, con mucha probabilidad, la mejor forma de ver y entender el desarrollo que sigue el software libre.

Uno de los principios básicos enunciados por Raymond, aunque no expresado precisamente de esta forma, es que el proceso de desarrollo debe ser lo más abierto posible. Una de las consecuencias es que cuantos más ojos hay mirando el código mayor es la probabilidad de encontrar fallos antes [Raymond]. La herramienta que mejor se adapta a esta necesidad es un sistema de control de versiones abierto al público. La última versión (y todas las anteriores) del código estará disponible para todo aquél que lo desee. El sistema de control de versiones, al igual que todas las demás herramientas utilizadas, son herramientas que permiten la colaboración de manera distribuida. Cualquier desarrollador en cualquier lugar del mundo podrá descargarse el código y corregir una errata que haya encontrado.

En realidad, la idea de Raymond va más allá, ya que la "comunidad" que se crea alrededor de un proyecto no se forma por sí sola. Son necesarios una serie de acciones y mecanismos para que el proyecto sea conocido. De esta forma, se consiguen usuarios que son el primer paso para tener desarrolladores. Esta tarea, aunque no necesariamente técnica, implica gran cantidad de trabajo. Uno de los métodos evocados por Raymond y proclamado incluso como principio por algunos es el famoso "release early, release often" (entrega pronto, entrega frecuentemente), que está pensado para captar la atención de una comunidad de usuarios (y desarrolladores) y mantenerla satisfecha con la evolución de una aplicación que satisfaga sus necesidades [Raymond].

La proliferación del acceso a Internet ha hecho que el ciclo de desarrollo se sustente fuertemente en el uso de sus servicios, de manera que se consigue un grupo potencial de usuarios (y desarrolladores) mucho mayor. Las formas de comunicación habituales son las listas de correo, abiertas a la suscripción de cualquiera que así lo desee. Los mensajes de las lista de correo son almacenados en los llamados "archivos", lo que permite poder tener acceso a todas las decisiones y discusiones de la lista. De esta manera, todo queda plasmado por escrito. También existen otros medios de comunicación como pueden ser los canales de IRC.

SourceForge y otros portales especializados en sustentar la creación de software libre lo que hacen es realizar las tareas de instalación, configuración y gestión de herramientas que permiten este tipo de desarrollos, de manera que un desarrollador que quiera crear un nuevo proyecto de software libre no se tenga que preocupar de tener que instalarse su propio software de control de versiones ni gestores de listas de correo-e. En definitiva, los desarrolladores de software libre pueden desarrollar y gestionar sus proyectos de manera que se aproveche la sinergia producida por un entorno de desarrollo lo más abierto posible.

3.3. Conclusión

Podemos concluir que el software libre per se no está asociado a ninguna forma de desarrollo específica, sino que se caracteriza por una serie de condiciones (legales) que un programa debe cumplir para ser considerado como tal. Sin embargo, la experiencia ha demostrado formas y métodos por los cuales proyectos de software libre tienen más éxito y evolucionan más rápido y mejor que otros, probablemente incluso mejor que el software propietario. La idea principal es aprovechar las características impuestas por las licencias para abrir el proceso de desarrollo al mayor número de personas y aprovechar mecanismos de difusión predefinidos (independientes de los desarrolladores) para llegar a tener un amplio grupo de usuarios. Además, la existencia de herramientas de colaboración distribuida permiten aprovechar al máximo efectos sinergéticos.

4. Software Libre y Programación Extrema

A lo largo de las introducciones a la programación extrema y al software libre hemos visto como algunas de las prácticas de la programación extrema son intrínsecas al desarrollo del software libre. Por contra, existe otra serie de prácticas que son de difícil implantación o que generan serios interrogantes. Este apartado va a tratar sobre todo esto.

Antes de todo, es importante recordar que para Beck se está haciendo programación extrema si y sólo si se siguen las doce prácticas, ninguna más ni ninguna menos [Beck][Jeffries3]. Fowler es más flexible en este sentido y prefiere

hablar entonces de procesos influenciados por la programación extrema [Fowler2]. Vemos que mientras el software libre es muy flexible en cuanto a su modelo de desarrollo (como ya hemos visto con anterioridad, en realidad es un concepto legal), la programación extrema consta de un conjunto conocido y cerrado de prácticas. Esta característica hará que consideremos invariante la programación extrema y veamos qué cosas son las que pueden ser interesantes para el desarrollo de software libre.

Como veremos, la programación extrema no puede ser implantada en su práctica totalidad por el desarrollo de software libre. Por eso, retomando las palabras de Fowler, veremos hasta qué punto se puede influenciar el software libre por los procesos de programación extrema.

4.1. Características intrínsecas de programación extrema en el software libre

En un párrafo anterior se ha comentado que el software libre sigue de por sí algunas prácticas de la programación extrema. Como son características ya adoptadas y ampliamente conocidas, no nos detendremos mucho en ellas.

La más evidente de las prácticas comunes es probablemente la propiedad colectiva del código, característica esencial para la libertad del software. Al igual que en la programación extrema, para facilitar el trabajo conjunto, se siguen uno estándares de codificación que permiten una lectura rápida y simple del código, a la vez que independiza el código de su autor. Un buen ejemplo en cuanto a estándares de codificación es el proyecto PEAR, un repositorio de clases PHP [Jansen]. En muchos proyectos, las aportaciones de terceros no se aceptan, ni siquiera revisan, hasta que sigan los estándares establecidos. Que el código sea comunitario en el software libre, como se ha dicho antes, no implica en muchas ocasiones que haya compartición de conocimiento total como se recomienda en la programación extrema.

El paradigma "entrega pronto, entrega frecuentemente" del software libre encaja perfectamente con la idea de tener entregas frecuentes de la programación extrema en busca de realimentación. El que se haga hincapié en hacer una primera entrega pronto encaja, por su parte, perfectamente con la idea de que se empiece por lo primordial y se dejen funcionalidades avanzadas para el futuro. A partir de ahí el proceso iterativo de probar, codificar, pasar pruebas y refactorizar de la programación extrema se cumple parcialmente en el software libre. Mientras las pruebas unitarias todavía se utilizan poco asiduamente, el proceso de tener pequeñas iteraciones se cumple casi al pie de la letra, llegando incluso a tener proyectos con entregas diarias. La refactorización en sí no ocurre como tal, siendo lo más frecuente una simple corrección de erratas. Lo más frecuente es que se de el caso de que los usuarios hagan depuración (paralela) y que ofrezcan realimentación, ya sea con la solución del problema o como un aviso para que el desarrollador principal o cualquier otra persona pueda corregirlo.

4.2. Prácticas de difícil adaptación

Al igual que existen ciertas prácticas de la programación extrema muy arraigadas y en concordancia con el proceso de desarrollo mayoritario de software libre, existen otras cuyo seguimiento se puede prácticamente descartar.

4.2.1. 40 horas semanales

La que se puede eliminar, sin lugar a dudas, es el requisito de que el equipo de trabajo deba tener una carga de trabajo razonable (40 horas). Los estudios realizados sobre desarrolladores de software libre demuestran que una gran mayoría de los desarrolladores de software libre colaboran en proyectos en su tiempo libre. Se da la circunstancia de que un 80% le dedica menos de 15 horas semanales. Sin embargo, aunque esta práctica está dirigida a un entorno laboral, también es cierto que se basa en el hecho de que la capacidad creativa no se puede alargar más allá de un cierto tiempo. El desarrollador necesita una serie de condiciones para poder ejercerla en su máxima expresión y una de ellas es que el ambiente laboral sea agradable.

En el caso de los colaboradores del software libre asumir esto es de perogrullo, ya que su colaboración es en cualquier caso de manera voluntaria.

En conclusión, esta práctica es de difícil adaptación en el mundo del software libre, pero por otra parte tampoco es necesaria, porque los objetivos que persigue ya son alcanzados por otros medios de por sí.

4.2.2. Cliente en casa

El segundo punto es ya un poco más complejo de solucionar: la cliente en casa. Y lo es por partida doble, ya que no es que no podamos asegurar que esté en casa, es que en realidad no tenemos la certeza de tener clientes. De hecho lo que se tiene son usuarios. Sin embargo, la figura del usuario difiere en bastantes aspectos de la de cliente.

En común tienen que, tanto en el software libre como en la programación extrema, se intente integrar al usuario/cliente en el equipo de desarrollo. En la programación extrema esto se hace para agilizar la realimentación, mientras que el software libre va más allá y también se ve la posibilidad de que el propio usuario termine formando parte del equipo de desarrollo, con el objetivo de que colabore también en tareas de depuración, corrección y, con suerte, integración.

Se diferencian en que en la programación extrema, todo el desarrollo gira en torno a las historias de los usuarios. El cliente es la fuente de ingresos y merece, por consiguiente, todas las atenciones. El desarrollo de software libre es más caótico en este sentido. El usuario recibirá la funcionalidad que precisa si existe algún desarrollador que está interesado en implementarla. El centro de decisión se ha trasladado en este caso hacia el desarrollador. Sin embargo, también es cierto que un usuario siempre tiene la posibilidad de pagar a un desarrollador para que aporte la funcionalidad al proyecto que satisfaga sus necesidades. Esto lo puede hacer incluso yendo en contra de la estrategia general del proyecto. Y es que una de las consecuencias económicas de las licencias de software libre es que imposibilitan el vendor lock-in, esa situación donde es el fabricante de software (aquél que tiene la propiedad intelectual) el que decide de manera unilateral el rumbo que debe tomar el software, así como lo que se puede hacer con él [Barahona].

También es cierto que en los grandes proyectos de software libre se hace un gran esfuerzo por conocer las necesidades de los usuarios para satisfacerles en próximas entregas, aunque este proceso no se encuentre formalizado como en la programación extrema mediante el uso de tarjetas CRC, cuya adopción sea probablemente de gran interés. El mecanismo habitual que tienen los usuarios para hacer peticiones a los desarrolladores suelen ser las listas de correo electrónico del proyecto. La cercanía del usuario es parcial.

Resumiendo, hemos visto que el software libre suele carecer de cliente como tal, aunque cuenta con la figura de usuario. Hemos tratado las diferencias existentes entre clientes y usuarios y las consecuencias que acarrean en el

proceso de desarrollo. También sabemos que a día de hoy, en el software libre no existen métodos formales para capturar necesidades de los usuarios y comprobar que han sido satisfechas.

4.2.3. El juego de planificación

Como hemos visto en el apartado anterior, la falta de cliente hará que sea muy difícil que se pueda llevar a cabo el juego de planificación que propone la programación extrema. En realidad, podemos asegurar que tal y como se viene desarrollando el software libre en este aspecto es todavía más extremo que el modelo de la programación extrema, ya que prescinde según los casos parcial o totalmente de este paso.

Los nuevos proyectos suelen crearse para satisfacer necesidades personales por parte de un desarrollador o un grupo de desarrolladores. En este miniestadio, el propio desarrollador suele ser a la vez el propio usuario, por lo que no necesita un proceso de formalización de historias de los usuarios. Según va creciendo el proyecto, es probable que el juego de planificación se haga más y más necesario, sobre todo si el número de usuarios crece y el equipo de desarrolladores está dispuesto a satisfacer las necesidades de los usuarios llegando incluso a realizar una planificación temporal del mismo. Como en muchos de los puntos anteriores vemos que no existen herramientas ni mecanismos de software libre que permitan formalizar esto convenientemente.

La mejor aproximación es quizás Bugzilla que permite ser utilizado además de para la gestión de la generación de informes de defectos y posterior corrección de los mismos, para organizar y repartir tareas entre el equipo de desarrollo. Otro método ampliamente difundido es el uso de wikis, aplicaciones con interfaz web abiertas a la contribución de todo el que las visite, aunque ciertamente es una solución bastante primitiva. Herramientas como MrProject, un clon de la aplicación para gestión de proyectos MS Project, pueden favorecer que esto mejore en un futuro próximo.

Vemos que en lo que al juego de planificación se refiere, el software libre es todavía más extremo que la propia programación extrema. Su ejecución se retrasa hasta estadios en los que es verdaderamente necesaria, siguiendo la idea que aparece en otras partes de la programación extrema: "si no lo necesitas ahora, no lo hagas".

4.2.4. Programación por parejas

La programación por parejas es una de las piezas clave de la programación extrema y que, sin embargo, parece que no tiene cabida en el mundo del software libre. Sin embargo, es posible que la programación por parejas no sea tan útil en el software libre como lo puede ser en un proyecto en una empresa.

La revisión entre iguales, tanto de código como de diseño, que propicia el trabajo en parejas, es una práctica que en muchos proyectos se da de por sí. El más conocido es el caso de Linux, donde se carece incluso de un repositorio CVS. Linus Torvalds quiere que todas las aportaciones de código se manden a la lista de correo. No como adjunto al mensaje, sino incluso en el cuerpo del mensaje. De esta forma está a la vista de todos y no es una práctica poco habitual que los parches sean analizados y comentados por terceras personas en la propia lista.

Existen otros mecanismos no formalizados de aceptación de colaboraciones de código. Generalmente lo más común es que los desarrolladores principales sean los únicos que tengan acceso de escritura al repositorio del sistema de

control de versiones. Cualquier persona es capaz de descargarse la última versión del mismo, pero tendrá que enviar a los desarrolladores principales el parche para que éste sea incluido. Será tarea de estos últimos revisar la contribución y verificar que sigue los parámetros de codificación y diseño pertinentes.

Otra ventaja adicional en el mundo de la empresa de tener dos programadores realizando tareas conjuntamente es por la alta movilidad laboral. Es bien sabido, que muchos proyectos pierden un tiempo valiosísimo cuando uno de los integrantes del equipo deja el trabajo (una situación que, por cierto, se ha dado con una pasmosa frecuencia en los últimos tiempos). La programación por parejas intenta mitigar este efecto, ya que de los dos programadores se quedará uno que tendrá el conocimiento suficiente sobre lo que estaba haciendo la pareja como para poder seguir adelante. Sin embargo, también en este caso, las condiciones que se dan en el desarrollo de software libre hacen que esto no sea tan problemático.

Vemos en definitiva, que aún cuando la programación por parejas se suple por otros medios, no existe un mecanismo formal que permita comprobar que los objetivos promulgados por la misma se cumplen.

4.3. Prácticas interesantes

En los siguientes puntos vamos a comentar las prácticas cuya adopción por parte del método de desarrollo del software libre puede ser interesante. Es verdad que estas prácticas ya se encuentran más o menos arraigadas en diferentes proyectos, pero en opinión del autor esta situación no es suficiente y, por tanto, se debe fomentar su uso.

Las prácticas que se incluyen en este apartado no chocan frontalmente con el modelo actual de desarrollo de software libre, por lo que su adopción se facilita. En la mayoría de los casos suponen más un cambio de costumbre en la forma de programar y utilizar la información que se maneja en un proyecto. Por contra, las consecuencias de este ligero cambio pueden suponer un sustancial aumento de calidad de las aplicaciones de software libre.

4.3.1. Pruebas unitarias y de aceptación

Las pruebas unitarias se deberían implementar con mayor frecuencia. Aunque se puede considerar que es una técnica ya existente en los años 70, la programación extrema puede hacer que se incremente su uso en los proyectos de software libre y, probablemente, que sea su mayor aportación a la misma. Las pruebas unitarias no tienen por qué hacerse estrictamente antes de codificar; al fin y al cabo es mejor tener pruebas, aunque sea con posterioridad, que no tenerlas.

Cuando un programador tiene que modificar código que ha sido desarrollado por otra persona se corre un alto riesgo de que introduzca alguna errata (bug). Las pruebas unitarias reducen este riesgo dado que avisarán al instante si algo deja de funcionar. Naturalmente para ello debe adquirirse el hábito de ejecutar todas las pruebas del sistema tras hacer un cambio. En los proyectos de software libre, la reducción de este factor de riesgo es muy útil, dado que es muy habitual modificar código escrito por otros.

Es probable que la introducción de las pruebas unitarias conlleve un cambio de filosofía en el mundo del software libre. De la búsqueda heroica de erratas en la que se hace tanto énfasis, se debe pasar a la implantación de buenas

maneras que redunden en código menos proclive a tenerlos. En definitiva, se debe tener a desarrolladores con talento creando código (que es probablemente lo que mejor hagan y más les estimule) en vez de repasando código erróneo de otros.

En un futuro, esperemos que este tipo de pruebas puedan estar integradas en sistemas de control de versiones como el CVS. Quizás haya que esperar a que Subversion, la herramienta de siguiente generación de control de versiones que todavía se encuentra en fase de desarrollo, las implemente. De esta forma, las pruebas unitarias formarían parte del sistema de control de versiones.

No nos debemos olvidar por otro lado de las pruebas de aceptación. La mayoría de los proyectos de software libre carecen en sus páginas web de información de hacia dónde va el proyecto: las funcionalidades que serán añadidas en un futuro próximo, los cambios que se van a hacer, etc. Una buena forma de resolver esto y abrir el desarrollo a más desarrolladores podría ser la creación de pruebas de aceptación previas a la implementación de la siguiente operación que podrán ser comprobadas al final de la misma, tal y como ocurren en los ciclos iterativos de la programación extrema. Esto también puede utilizarse para saber las funcionalidades que son prioritarias para los usuarios.

4.3.2. Metáfora

El objetivo de la metáfora del sistema es proporcionar a todo el equipo una misma visión del fin del sistema y de su arquitectura general. Con ello se facilita que todos los desarrolladores hablen un mismo idioma y que nuevos desarrolladores lo adquieran más rápido y integrarse en el proyecto sin dificultades. Este segundo aspecto es el que puede hacerla interesante para el software libre. La posibilidad de conseguir el código en muchas ocasiones no rebaja suficientemente la barrera de entrada que existe para nuevas incorporaciones. De hecho, estoy seguro de que la información que se puede plasmar de manera concisa en la metáfora se encuentra dispersada en el código y las listas de correo.

Plasmar la metáfora por escrito puede suponer, por tanto, una forma de revisar el propio diseño del sistema por parte de los desarrolladores. Asimismo, ya que estamos hablando de un nivel de abstracción superior al de codificación, puede alentar a la utilización de patrones de diseño, una práctica bastante desconocida en el mundo del software libre.

El uso de la metáfora junto con formas de documentación de código, del estilo de javadoc y similares, puede paliar en parte la falta de documentación que existe en el mundo de software libre. Estas prácticas podrían incluso llegarse a automatizarse parcialmente (de hecho javadoc es en sí una herramienta más que una forma de documentación). Muchos adeptos a la programación extrema se muestran contrarios a la utilización de estas maneras de crear documentación del código, ya que argumentan que el código debe ser lo suficientemente simple como para poder ser entendido por sí mismo. Sin embargo, habrá que tener en cuenta que en muchas ocasiones para refrendar las interfaces de una clase o biblioteca puede ser muy útil, ya que el desarrollador puede no estar interesado en esa clase o biblioteca, sino en desarrollar una aplicación que la utilice.

4.3.3. Refactorización

El principal objetivo de la refactorización es que en vez de corregir código erróneo se reescriba, una idea contrapuesta a la búsqueda y caza de erratas que tanto han proclamado grandes gurús del software libre. El nuevo código, por su parte, debe tener un mejor diseño: tiene que ser más simple y estar mejor estructurado que el anterior. Detrás de esta idea, está el convencimiento de que con la inclusión de pruebas unitarias, la corrección de erratas será muchas veces mucho más costosa que la propia reescritura.

Si se mantiene un sistema lo más simple posible se facilita que nuevos desarrollares entren en el proyecto y realicen aportaciones. Esta idea es la contraria al enfoque según el cual lo primero es crear una gran infraestructura para que luego sea más fácil añadir funcionalidad. El principal problema es que en este caso no es necesario conocer esa infraestructura para poder aportar . Más grave aún es que no es posible crear una infraestructura que prevea todas las funcionalidades futuras, por lo que si no se refactoriza acabará siendo un impedimento.

Con refactorización continua se podría evitar que aplicaciones enteras sean reescritas, porque los errores de diseño, la calidad del código y otros parámetros hacen imposible mantener la antigua versión.

4.4. Desarrollo distribuido y programación extrema

El hecho de que el software libre se produzca mayoritariamente de forma distribuida hace indispensable un análisis un poco más exhaustivo de la implicación que tiene la distribución en las técnicas de programación extrema. Para ello vamos a retomar algunos aspectos que ya han sido comentados, aunque poniendo ahora el énfasis en su carácter distribuido.

La generación distribuida de software también es interesante desde un punto de vista netamente empresarial. La globalización de la economía mundial ha propiciado que exista un número creciente de proyectos cuyo equipo de desarrollo, por razones económicas o personales, se encuentra distribuido geográficamente.

La experiencia con los métodos de desarrollo tradicionales ha demostrado que el esfuerzo dedicado a documentar pormenorizadamente o crear documentación extra no ha incrementado la eficiencia comunicativa del equipo que trabaja de manera distribuida. Es más, en realidad, se ha visto que es contraproducente.

Por el contrario, nos encontramos con un dilema al intentar llevar a cabo algunas de las prácticas de la programación extrema, ya que varias de ellas asumen proximidad física. Es el caso de las siguientes prácticas:

1. Programación por parejas
2. Integración continua
3. Juego de planificación
4. Cliente en casa

Esto ha dado paso a lo que se ha venido a llamar la programación extrema distribuida (DXP - Distributed eXtreme Programming), que permite a los miembros del equipo estar en lugares geográficos diferentes e incluso gozar de una gran movilidad. El método se basa en aplicar los valores y principios de la programación extrema, pero adapta las prácticas a un entorno de trabajo distribuido, de manera que se relaja la asunción de proximidad física.

Es verdad que las tecnologías Internet han ayudado a que el trabajo distribuido sea posible, pero también es cierto que esas tecnologías que se usan (páginas web, IRC, listas de correo, bitácora de cambios y documentación -- ya sea generada manual o semiautomáticamente) existían ya en los años 70. Aunque es constatable que estas tecnologías no pueden reemplazar la presencia en persona, el salto a las nuevas (y tan prometidas) formas de comunicación como la videoconferencia no son accesibles al público en general.

4.5. Interrogantes y retos

El software libre se ha caracterizado en los últimos años porque su proceso de desarrollo es flexible. El hecho de que queramos introducir varias técnicas nuevas para acentuar la calidad de las aplicaciones generadas, da pie a una serie de interrogantes y retos que habrá que ir solucionando.

4.5.1. Compatibilidad hacia atrás y dependencias

El diseño iterativo que promulga la programación extrema choca frontalmente en muchos proyectos con la compatibilidad hacia atrás. Quizás esto también sea una de las debilidades de la programación extrema, ya que invita a realizar frecuentes entregas y diseño iterativo pero no aborda la cuestión de mantener la compatibilidad hacia atrás o de la migración de datos de versiones anteriores. La programación extrema está orientada a satisfacer las necesidades de un cliente conocido donde este problema está más controlado. Sin embargo dentro de la industria del software, y por supuesto con mayor frecuencia en el software libre, no es lo general ya que se dirigen a una gran multitud de usuarios finales. Es, por tanto, muy importante poder asegurar al usuario que el trabajo realizado con anterioridad con la aplicación pueda ser manipulado por las nuevas versiones de la aplicaciones y en el caso de las librerías que se mantengan las APIs durante al menos un tiempo de migración.

Sin embargo, el mayor problema para utilizar los métodos de la programación extrema en el software libre son las dependencias. La programación extrema asume que el proyecto se encuentra aislado en el mundo, una asunción que en demasiadas ocasiones no es cierta. Muchos desarrollos se basan en otros paquetes que a la vez se encuentran en fase de desarrollo y cuyas características cambian constantemente. En esta carrera es muy difícil mantenerse independiente y aislado del mundo exterior. Las pruebas de regresión y las pruebas unitarias pueden ser dos maneras de mitigar los efectos de las dependencias dentro del software libre.

4.5.2. Interrogantes económicos y psicológicos

Una de las críticas a la programación extrema es la dificultad de estimar cuánto va a costar un proyecto, un problema asociado, por otra parte, también a las metodologías tradicionales. Es verdad que esto afecta en parte al software libre, ya que muchos proyectos no están ubicados en empresas ni tienen que generar una estimación de costes. Por otro

lado, sería interesante tener alguna forma de hacerlo de manera más o menos exacta para empresas que realizaran desarrollos de software libre.

Los efectos que puede causar la refactorización no están del todo claros. El hecho de tener menos líneas de código después de refactorizar tiene un efecto psicológico negativo importante, además de que siempre es doloroso tirar líneas de código.

A favor del software libre está el hecho de que los desarrolladores se suelen adaptar rápidamente a nuevas ideas, ya que ven el desarrollo de software libre como un buen laboratorio para técnicas novedosas. Esto no suele ser cierto en el mundo empresarial, mucho más inerte y pesado ante nuevos cambios, donde existe frecuentemente el llamado síndrome de "no lo hemos inventado aquí".

Por supuesto, hay que tener en cuenta que gran parte del éxito en la adopción de técnicas de programación extrema dependerá de que se creen nuevas herramientas que las soporten o que las existentes integren los nuevos métodos.

En general, podemos concluir que todavía no existe la experiencia suficiente en proyectos de software libre para poder hablar de los efectos secundarios que provocaría la adopción de las técnicas de programación extrema.

5. Conclusiones

A lo largo de este documento hemos visto cómo el método de desarrollo seguido mayoritariamente en el software libre y la programación extrema tienen muchas características comunes. Por otro lado, existen una serie de prácticas (pruebas, metáfora y refactorización) que sería muy interesantes adoptar en el software libre, ya que conllevarían un aumento de la calidad. Las pruebas permitirán a los desarrolladores realizar modificaciones con la seguridad de que no rompen nada en otro lugar, la metáfora proporciona una visión intuitiva del sistema y la refactorización está pensada para mejorar la calidad del código. Estas tres iniciativas tienen como objetivo principal transformar el modelo de desarrollo que actualmente está centrado en demasía en encontrar y corregir erratas a una nueva forma de desarrollar en la que el diseño iterativo y la codificación tomen mayor relevancia. Dado que lo segundo es más divertido que lo primero no dudo que será muy provechoso en un entorno donde la mayoría de los contribuidores lo hacen en su tiempo libre y de manera altruista.

A la vista de lo que se ha ido exponiendo parece ser que la mejor solución para la adopción de técnicas de programación extrema en el software libre es la existencia de un núcleo que asegure las prácticas de la programación extrema como pequeñas entregas y que permitan tener un diseño simple y controlado (un ejemplo de esto podría ser Linux, donde de facto esto ya existe más o menos de esta manera). Este grupo puede funcionar de por sí como un equipo de desarrollo distribuido, teniendo asignada entre otras tareas la de comprobar que las aportaciones exteriores cumplen los requisitos de calidad necesarios para ser integradas en el proyecto.

Otro aspecto interesante es que la programación extrema ofrece métodos formales para plasmar información que no existen de tal forma en el software libre. Sería interesante contar con dichos métodos para que la barrera de entrada a nuevos desarrolladores sea más baja. Un ejemplo interesante de estos métodos son las tarjetas CRC que permiten tener una idea intuitiva del proyecto en un espacio de tiempo muy corto sin tener que analizar concienzudamente la

organización de ficheros y código. Yendo más allá, se podría utilizar UML además de las tarjetas CRC para especificar el diseño de la aplicación [Wills]. Es probable que esto supongo una redefinición del uso de las tarjetas CRC, ya que en la programación extrema sirven sólo para comunicarse en una reunión, careciendo después de importancia como documentación porque se asume que se quedarán obsoletas enseguida.

En definitiva, se ha podido ver cómo la programación extrema puede aportar nuevas formas en busca de optimizar el modelo de desarrollo de software libre. El tiempo dirá si estas prácticas son asumidas por los diferentes proyectos de software libre y, en su caso, si realmente harán gozar de las ventajas que se han presentado.

6. Referencias (en orden alfabético)

1. [Barahona] Jesús María González Barahona "Software libre, monopolios y otras yerbas", http://sinetgy.org/~jgb/articulos/soft-libre-monopolios/

2. [Beck] Kent Beck "Extreme Programming Explained: Embrace Change", Addison-Wesley Pub Co; ISBN: 0201616416, 1a edición octubre 1999

3. [Beck2] Kent Beck & Erich Gamma "JUnit Cookbook", http://junit.sourceforge.net/doc/cookbook/cookbook.htm

4. [Bezroukov] Nikolai Bezroukov "A Second Look at the Cathedral and the Bazaar", http://www.firstmonday.dk/issues/issue4_12/bezr

5. [Cockburn] Alistair Cockburn, Laurie Williams "Costs and Benefits of Pair Programming", http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF

6. [Fowler] Martin Fowler "Is Design Dead?", http://www.martinfowler.com/articles/designDead.html

7. [Fowler2] Martin Fowler "Variations on a Theme of XP", http://www.martinfowler.com/articles/xpVariation.html

8. [Fowler3] Martin Fowler "Refactoring: Improve the Design of Existing Code", Addison-Wesley Pub Co, ISBN: 0201485672, 1a edición junio 1999

9. [Gamma] Erich Gamma & Kent Beck "JUnit A Cook's Tour", http://junit.sourceforge.net/doc/cookstour/cookstour.htm

10. [Gamma2] Erich Gamma & Kent Beck "JUnitTest Infected: Programmers Love Writing Tests", http://junit.sourceforge.net/doc/testinfected/testing.htm

11. [Ghosh] Rishab Aiyer Ghosh "Cooking pot markets: an economic model for the trade in free goods and services on the Internet", http://www.firstmonday.dk/issues/issue3_3/ghosh/

12. [Harrison] Peter Harrison "Evolutionary Programming", http://www.devcentre.org/research/evoprogramming.htm

13. [Impl] Implementaciones de tests unitarios para diferentes lenguajes y plataformas "Testing Frameworks", http://www.xprogramming.com/software.htm

14. [Jansen] Martin Jansen, Tomás V.V. Cox & Alexander Merz "PEAR Coding Standards", http://pear.php.net/manual/en/standards.php

15. [Jeffries] Ron Jeffries "Essential XP: Unit Tests at 100", http://www.xprogramming.com/xpmag/expUnitTestsAt100.htm

16. [Jeffries2] Ron Jeffries "Essential XP: Junior / Senior Pairing", http://www.xprogramming.com/xpmag/pairing.htm

17. [Jeffries3] Ron Jeffries "They're called Practices for a reason", http://www.xprogramming.com/xpmag/PracticesForaReason.htm

18. [Lerner] Josh Lerner & Jean Tirole "The Simple Economics of Open Source", http://www.idei.asso.fr/Commun/Articles/Tirole/simp July-24-2001.pdf

19. [Raymond] Eric S. Raymond "The Cathedral and the Bazaar", http://www.tuxedo.org/~esr/writings/cathedral-bazaar/

20. [Stallman] Richard Stallman "The Free Software Definition", http://www.fsf.org/philosophy/free-sw.html

21. [Wills] Alan Cameron Wills "UML meets XP", http://www.trireme.com/whitepapers/process/xp-uml/paper.htm

7. Bibliografía y otras direcciones de interés

1. Relación de libros sobre Ingeniería del Software, patrones (patterns) y Programación Extrema (Extreme Programming, XP), http://www.ctv.es/USERS/pagullo/biblio/ingenieria_del_software.htm

2. Donnovan Wells "Extreme Programming: A gentle introduction", http://www.extremprogramming.org

3. Allison Pearce Wilson "Extreme Programming", http://www-106.ibm.com/developerworks/library/it-aprcc01/?dwzone=ibm

4. chromatic "An Introduction to Extreme Programming", http://linux.oreillynet.com/lpt/a//linux/2001/05/04/xp_intro.html

5. Andrew McKinlay "Extreme Programming and Open Source Software", http://www.advogato.org/article/202.html

6. Ron Jeffries "Manuals in Extreme Programming", http://www.xprogramming.com/xpmag/manualsInXp.htm

7. Kent Beck "Simple Smalltalk Testing: With Patterns", http://www.xprogramming.com/testfram.htm

8. Leigh Dodds "XP Meets XML", http://www.xml.com/pub/a/2001/04/04/xp.html

9. Varios autores (Wiki) "Combining Open Source And Xp", http://c2.com/cgi/wiki?CombiningOpenSourceAndXp

10. Armin Roehrl & Stefan Schmiedel "Absolut extrem!!! Extreme Programming und Open Source Entwicklung" (alemán), http://www.entwickler.com/le/ausgaben/2001/10/artikel/2/online.shtml

11. Proyecto Gestión Libre de Hispalinux, http://gestion-libre.hispalinux.es

12. Jorge Ferrer "Extreme Programming y Software Libre - Aprendiendo una metodología de software 'tradicional'", http://www.jorgeferrer.com/doctorado/xp-y-sw-libre/

13. Anónimo (Visión sarcástica de la XP) "RiP: Programmer Invents New Methodology", http://www.bad-managers.com/rumours/story021.shtml