

Programación extrema y software libre

Diego Berrueta (diego@berrueta.net)*

22 de enero de 2006

Resumen

Este artículo está dividido en tres partes. La primera hace un repaso de la programación extrema (*Extreme Programming*, XP). La segunda analiza los puntos de coincidencia y de divergencia entre la programación extrema y el modelo más habitual de desarrollo de software libre (bazar). La última parte examina las herramientas *opensource* que dan soporte a las prácticas y al proceso de la programación extrema.

1. Programación extrema¹

La programación extrema es una metodología para el desarrollo de proyectos informáticos que trata de dar solución a los problemas de la ingeniería del software desde un enfoque completamente distinto al que ha venido siendo habitual. Los estudios demuestran que la mayoría de proyectos de software fracasan, porque exceden sus plazos, superan su presupuesto, no se ajustan a las auténticas necesidades del cliente, presentan una calidad deficiente o, en muchos casos, son abortados. Las metodologías tradicionales han tratado de poner coto a esta situación mediante un control intensivo del proceso. Al hacerlo, se está ignorando que las necesidades del cliente y sus expectativas realmente cambian durante el desarrollo del proyecto.

*This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.5 Spain License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

¹Para la elaboración de esta sección se ha utilizado extensamente el material disponible en [2, 3, 10].

Como respuesta, ha surgido una nueva familia de metodologías denominadas *ágiles*, cuyo rasgo principal consiste en contemplar y dar respuesta a las necesidades dinámicas del cliente. De entre las metodologías ágiles, la que goza de mayor popularidad es la programación extrema, propuesta en 1999 por Kent Beck, en un libro titulado precisamente “abrazo el cambio”. La programación extrema recibe este calificativo precisamente porque defiende un enfoque radical. Reconoce las bondades de las prácticas de las metodologías tradicionales (diseño, pruebas, revisiones de código, etc.) y propone llevarlas hasta su extremo: “si diseñar es bueno, diseñemos todo el tiempo”, “si las pruebas son buenas, probemos todo el tiempo”, etc.

1.1. Cuatro valores

La programación extrema, lejos de ser un proceso incontrolado, es una metodología muy disciplinada y que se apoya en cuatro valores fundamentales:

- **Comunicación.** Se hace énfasis en que la comunicación, para ser efectiva, debe involucrar a todos los participantes en el proyecto, y debe ser libre y sincera.
- **Simplicidad.** Nunca debe perderse de vista que el objetivo de un proyecto es proporcionar valor al cliente; no es demostrar la pericia técnica del equipo ni construir una aplicación que resuelva más problemas que los del cliente.
- **Realimentación.** No se puede dirigir adecuadamente un proceso si no se dispone de realimentación permanente sobre su progreso. La reali-

mentación puede provenir del cliente, de los programadores, de herramientas automáticas, etc.

- Coraje. A veces, hacer lo que es correcto requiere valor. Por ejemplo, hay que tener coraje para reescribir código que funciona pero ha alcanzado su límite de escalabilidad.

Estos cuatro valores dan origen a cinco principios básicos: conseguir realimentación rápida, no complicar las cosas con suposiciones (asumir que las cosas son simples), realizar cambios incrementales, abrazar el cambio y generar productos de calidad. Los cinco principios se manifiestan a través de las prácticas de la programación extrema.

1.2. Doce prácticas

Las prácticas son las actividades que el equipo de un proyecto lleva a cabo cada día. Las doce prácticas de la programación extrema tienen su origen en prácticas bien conocidas en la ingeniería del software y en el sentido común, y por tanto, no pueden resultar extrañas. Sin embargo, lo que caracteriza a este conjunto es la cohesión de todos los elementos, y que cada práctica ha sido llevada al extremo:

- El juego de la planificación. Esta práctica busca dividir la funcionalidad de un proyecto en pequeños fragmentos autocontenidos, cada uno de los cuales se denomina *historia de usuario* (*user story*)[5]. El cliente y el equipo del proyecto dialogan para decidir qué historias son más importantes (siempre se hacen las historias más importantes primero) y estimar cuánto puede tardar el equipo en completar cada historia.
- Entregas frecuentes. Se trata de publicar una nueva versión en cuanto sea posible aportar algún nuevo valor al cliente. De esta forma, se maximiza la realimentación y se controla el proyecto más fácilmente.
- Diseño simple. El sistema debe ser el más simple posible que cumpla las especificaciones (pruebas de aceptación). En un entorno donde los requisitos del cliente y sus prioridades cambian continuamente, no tiene sentido realizar un diseño

sofisticado que contemple hipotéticas necesidades futuras. La mejor forma de obtener una idea de los futuros requisitos de un sistema es proporcionar cuanto antes un prototipo al cliente y obtener realimentación; y la mejor forma de obtener un diseño simple que funcione es recurrir a los patrones de diseño[7].

- Pruebas automáticas. ¿Cómo puede saber un programador que el código que ha escrito funciona realmente? ¿Cómo puede saber que seguirá funcionando siempre, incluso aunque lo refactorice? La única manera de asegurarlo con cierta confianza es escribiendo pruebas automáticas con las que pueda comprobar el código en cualquier momento y sin esfuerzo. Las pruebas no pueden dejarse para el final, sino que deben escribirse al mismo tiempo que el código, o incluso antes[1].
- Integración continua. Nuevamente se lleva al extremo una práctica convencional de la ingeniería del software. Si la integración es una fase crucial, en la que pueden aparecer errores, ¿por qué dejarla para el final, cuando el riesgo es mayor? Resulta más conveniente realizar integración continua (cada hora, cada día). Para poder hacerlo, es imprescindible que el proceso de integración esté automatizado y pueda verificarse mediante pruebas.
- Refactorización. La refactorización es un proceso disciplinado por el cual se modifica el diseño de un módulo sin afectar a su comportamiento externo[6, 11]. Gracias a la refactorización, es posible compatibilizar el diseño simple con la flexibilidad. El coraje para refactorizar proviene de la disponibilidad de pruebas automáticas.
- Programación por parejas. Una vez más, se lleva al extremo una práctica habitual: las revisiones formales de código. Si revisar el código es bueno, ¿por qué no revisarlo continuamente, incluso desde el mismo momento en el que se escribe por primera vez? En la programación por parejas, dos programadores comparten un único ordenador y colaboran para escribir el código o las pruebas.

De esta forma, se estimula la comunicación y la transmisión de conocimiento, se detectan antes los errores y se produce código de más calidad.

- Propiedad colectiva del código. En el transcurso de una refactorización, o mientras se corrige un defecto, algún programador va a tener que modificar líneas de código escritas por otro programador. La metodología XP invita a llevar a cabo esa modificación con coraje, y el coraje procede de las pruebas automáticas. Esta práctica permite que funcionen bien los equipos dinámicos, cuya composición puede variar durante el proyecto.
- Semana de 40 horas. Los programadores cansados se equivocan más. Si las semanas de más de 40 horas son la norma, algo no funciona bien en el proyecto o en la empresa.
- Cliente en el equipo. Para lograr una realimentación ágil, el cliente no puede estar muy lejos del equipo; en una situación ideal, el cliente debe estar *dentro* del equipo. De esta forma, puede ayudar a los programadores a escribir las pruebas de aceptación.
- Metáfora. El objetivo de esta práctica es encontrar una metáfora que ayude al equipo del proyecto y al cliente a hablar en los mismos términos, compartiendo una visión común del sistema.
- Estándares de codificación. Utilizar estándares de codificación es una necesidad cuando se trata de escribir código que otros programadores puedan entender y modificar.

2. Programación extrema y desarrollo de software libre²

Antes de trazar paralelismos entre la programación extrema y el desarrollo del software libre, debe plantearse una cuestión. ¿Cuál es la metodología, si es que

²Para esta sección se ha usado extensamente el material disponible en [4, 13, 15, 16].

existe alguna, que siguen los proyectos de software libre? Obviamente, no es posible dar una respuesta en términos absolutos, teniendo en cuenta los centenares de miles de proyectos de software libre que existen. No obstante, observando los proyectos más representativos, se pueden identificar algunos rasgos comunes. La mejor representación del proceso de desarrollo de software libre es el *modelo bazar*[14] (en oposición al *modelo catedral*, más habitual en el mundo de software propietario).

Se trata de una analogía entre el proceso de desarrollo de software y el funcionamiento de un bazar. En un bazar, los tenderos acuden, plantan sus puestos y dialogan con los clientes. No existe una autoridad central que organice el funcionamiento, pero eso no impide que tanto los clientes como los proveedores consigan sus objetivos. En el otro lado, la catedral es diseñada por un arquitecto, ejecutada siguiendo un plan minuciosamente trazado, completada y entregada al cliente (aunque, tal vez, realmente sólo quisiera una capilla).

Algunas de las prácticas de la programación extrema son totalmente compartidas por el modelo bazar:

- El software libre aplica la máxima “*release soon, release often*”, por la cual se liberan nuevas versiones muy frecuentemente, incluso desde el mismo comienzo del proyecto. No es extraño ver números de versiones como “0.0.1” ó “linux-2.6.15-rc2”. Esta práctica encaja perfectamente con las entregas frecuentes de la programación extrema.
- También se produce integración continua, gracias a que el código se almacena en repositorios de control de versiones (CVS, Subversion, etc.). En cualquier momento, un nuevo usuario o desarrollador puede descargarse el código del repositorio y compilarlo (y si no lo consigue, se pondrá en contacto con los desarrolladores para notificar el problema). Muchos proyectos ofrecen también versiones empaquetadas automáticamente (*nightly builds*) para los que no quieren acceder directamente al repositorio. Es también muy común encontrar proyectos que mantienen dos ramas de desarrollo, una considerada “*stable*” y otra denominada “*unstable*” o “*devel*”. La

Cuadro 1: Relación entre las prácticas de la programación extrema, las prácticas del modelo bazar y las herramientas de software libre que dan soporte a las prácticas.

Práctica XP	Modelo bazar	Herramientas
Juego de la planificación		
Entregas frecuentes	<i>Release soon, release often</i>	
Diseño simple		
Pruebas automáticas		JUnit, HttpUnit, DbUnit...
Integración continua	Repositorios, <i>nightly builds, unstable</i>	CVS, SVN, Ant, CruiseControl...
Refactorización		Eclipse
Programación por parejas		
Propiedad colectiva del código	GPL y otras licencias libres	CVS, SVN
Semana de 40 horas		
Cliente en el equipo	Realimentación de <i>bugs y feature requests</i>	Bugzilla, listas de correo, foros...
Metáfora		
Estándares de codificación	Guías de estilo	Jalopy, Indent, JCS...

rama *unstable* suele ser el foco de la integración continua. A veces esta distinción se hace mediante un convenio de numeración de versiones (números impares indican versiones en desarrollo, números pares indican versiones estables); otras veces se mantienen explícitamente los nombres de las ramas y se da al usuario la posibilidad de elegir, como por ejemplo en Debian. Precisamente en esta distribución se produce uno de los casos más representativos de integración continua, puesto que los contenidos de la rama *unstable* se están integrando permanentemente de forma automática, usando el demonio *buildd*.

- La propiedad colectiva del código está en la propia naturaleza del software libre, por lo que esta práctica es perfectamente compatible con el modelo bazar.
- Algunos grandes proyectos de software libre disponen de guías de estilo explícitas para la codificación. Incluso en aquellos proyectos que no disponen de ellas, el estilo suele estar implícitamente determinado por el código escrito por los líderes del proyecto.

La práctica de introducir al cliente en el equipo no es perfectamente asimilable, puesto que los proyectos

de software libre se caracterizan por una gran dispersión geográfica y tampoco es sencillo identificar quién es el cliente. Es frecuente que el desarrollador sea el propio cliente, puesto que la motivación para desarrollar software libre es, en muchos casos, egoísta (resolver una necesidad propia). Para facilitar la participación del cliente, la comunidad del software libre ha creado herramientas colaborativas como los gestores de errores (*bugtrackers*), los wikis, las solicitudes de funcionalidades (*feature requests*), los foros, las listas de correo, los canales de IRC, etc. Utilizando estas herramientas, los programadores y los clientes (usuarios) se comunican y se realimentan de una forma ágil, salvando la dispersión geográfica.

Por último, hay un conjunto de prácticas de la programación extrema para las que difícilmente se puede encontrar equivalencia en el modelo bazar:

- El juego de la planificación no encaja en el modelo bazar, que se caracteriza por una escasa planificación: habitualmente no se planea qué funcionalidades incluirá la próxima versión, ni cuándo estará lista (de hecho, cuando se pregunta cuándo saldrá la próxima versión, se suele responder que “saldrá cuando esté lista”). Dada la creciente repercusión comercial del software libre, recientemente algunos proyectos, como GNOME o Ubuntu, han comenzado a introducir cierta pla-

nificación en sus ciclos de desarrollo, fijando las fechas de liberación de nuevas versiones, las funcionalidades que incorporarán, la duración de los procesos de “congelación”, el tiempo durante el que se mantendrá el soporte técnico y de seguridad de las versiones antiguas, etc.

- El diseño simple tampoco es una característica que sea explícitamente promovida por el modelo bazar. Sin embargo, el diseño simple suele manifestarse en los proyectos de software libre como un efecto lateral del entorno: en un ámbito de alta competencia (para cualquier área, pueden encontrarse varios proyectos de software libre que la cubren) se establece un proceso de selección natural. Como los proyectos con un diseño simple son más flexibles y escalables, consiguen atraer más fácilmente a nuevos desarrolladores y sobreviven.
- Al tratarse de equipos muy heterogéneos, la práctica de la refactorización queda frecuentemente a discrección de cada programador.
- La programación por parejas no puede ser llevada a cabo porque los programadores se encuentran geográficamente distantes y no comparten un lugar y horario de trabajo.

3. Programación extrema y herramientas de software libre³

3.1. Herramientas para pruebas automáticas

Para ser realmente efectivas, las pruebas de software tienen que satisfacer varias condiciones:

- Ser fáciles de escribir. En el caso ideal, personas sin formación en programación (como suele ser el caso del cliente) deberían ser capaces de escribir las pruebas. En la práctica, es muy difícil que esto pueda lograrse. Sin embargo, en la programación extrema, las pruebas de aceptación son

³Para esta sección se ha usado extensamente el material disponible en [8, 9, 12, 17].

escritas por el programador y el cliente trabajando en parejas, o bien a partir de las indicaciones del cliente que se capturan en la historia de usuario.

- Ser automatizables, de forma que ejecutarlas y comprobar que funcionan sea muy cómodo, e incluso pueda hacerse de forma desatendida varias veces al día después de realizar cualquier cambio en el código.
- Ser capaces de cubrir todos componentes de una aplicación, en distintos niveles de abstracción (pruebas unitarias, pruebas de integración, pruebas funcionales, etc.).

Existe un *framework* que satisface estos requisitos. Genéricamente, se denomina xUnit, y su implementación más conocida es JUnit (para Java), escrita por Erich Gamma (uno de los autores conocidos como *Gang of Four*) y Kent Beck. Se trata de un pequeño conjunto de clases, fácil de usar y aprender, que permite a los programadores escribir pruebas unitarias (a nivel de método) de su código Java. Este *framework* es extendido por un gran número de bibliotecas que aumentan sus capacidades, ya sea para probar componentes especializados (como EJBs, etiquetas JSP, servlets, objetos de acceso a datos, etc.), facilitar las pruebas de clases individuales mediante *mock objects*, realizar las pruebas dentro del contenedor de aplicaciones (pruebas de integración), o escribir pruebas funcionales capaces de interactuar con interfaces web, por ejemplo. Algunas de estas bibliotecas son StrutsTestCase, DBUnit, EasyMock, MockObjects, MockMaker, DynaMock, Cactus, JWebUnit, XMLUnit y HttpUnit. Al estar integradas bajo el mismo *framework*, su aprendizaje es rápido, y la forma de uso es homogénea.

El *framework* xUnit tiene implementaciones disponibles para muchos otros lenguajes de programación: C# (NUnit), C++ (CppUnit), PHP (PHPUnit), Python (PyUnit) y Haskell (HUnit), por citar algunos.

3.2. Herramientas para integración continua

Un requisito para que sea posible la integración continua es que todos los programadores trabajen simultáneamente sobre la misma base de código. Esto sólo es posible utilizando herramientas de control de versiones y trabajo colaborativo, que se analizan en el siguiente apartado. Asumiendo que todo el código fuente está almacenado en un repositorio, el siguiente paso es automatizar completamente el proceso de integración, compilación, despliegue o instalación y, finalmente, ejecución de las pruebas. Aunque el proyecto GNU dispone de las autotools (autoconf, automake, etc.), su aprendizaje es realmente complejo, y por tanto, se han quedado obsoletas para las aplicaciones empresariales de hoy en día (aunque siguen siendo muy válidas para aplicaciones escritas en lenguajes como C y destinadas a UNIX).

Las plataformas más actuales, como Java/J2EE y .NET, disponen de otras herramientas. Entre ellas, destaca Ant por su popularidad, aunque no es la única (por ejemplo, también existe Maven para Java, y NAnt para .NET). Ant es un completo *framework* que permite automatizar todas las fases de un proceso de integración continua (incluyendo accesos al repositorio, transferencias por red o ejecución de pruebas JUnit), y además dispone de una arquitectura fácilmente extensible. De esta forma, escribiendo un fichero de entrada para Ant, la integración se realiza con sólo una orden.

Pero la integración debe ser continua, y no se puede esperar que los programadores se ocupen de repetir el proceso una y otra vez a cada paso. Afortunadamente, existen otras herramientas que automatizan el ciclo de integración. La más conocida es CruiseControl (para Java y Ant) y su pareja CruiseControl.NET. Estas aplicaciones permanecen a la espera (como un demonio UNIX), monitorizando el estado del repositorio de control de versiones. En el momento de detectar alguna modificación, se despiertan y ejecutan toda la cadena de integración. Finalmente, notifican el resultado a través de correo electrónico u otros medios. De esta forma, a los pocos minutos de subir nuevo código al repositorio, los programadores reciben un mensaje si la modificación produce un

problema de integración o rompe alguna prueba de JUnit.

3.3. Herramientas para trabajo colaborativo

Las herramientas de trabajo colaborativo son fundamentales para la programación ágil. La más importante de todas ellas es el control de versiones, que permite a los programadores trabajar en paralelo sobre el mismo árbol de código fuente. El software libre dispone de excelentes sistemas de control de versiones, desde el popular CVS hasta el moderno Subversion y muchos otros más minoritarios y sofisticados.

Otras aplicaciones también son útiles para coordinar a un equipo ágil. En particular, se antojan especialmente interesantes los gestores de errores o incidencias (como Bugzilla o Mantis) y los wikis. Por este motivo, las herramientas que ofrecen integración de todas estas características resultan muy atractivas. Entre ellas, se pueden señalar GForge y Trac.

3.4. Herramientas para refactorización

La herramienta *opensource* más madura para llevar a cabo refactorizaciones es la que está integrada en el entorno de desarrollo de Java de Eclipse. Permite una gran variedad de refactorizaciones (incluidas algunas complejas), se integra perfectamente con el editor y con el sistema de control de versiones. Por desgracia, sólo sirve para refactorizar código Java.

JEdit es otro popular editor que dispone de un gran número de *plugins*, incluido uno para realizar refactorizaciones llamado JavaRefactor. Nuevamente, sólo se aplica a código Java.

Existen algunas herramientas en desarrollo para refactorizar otros lenguajes, como Python. También cabe mencionar que hay herramientas no libres para refactorizar C++ que se integran en editores libres como Emacs y JEdit.

3.5. Herramientas para seguir estándares de codificación

Cada lenguaje de programación tiene un estilo de codificación característico, y en ocasiones, más de uno. Por estilo de codificación, convenciones de codificación, o directamente, estándares de codificación, se hace referencia a la forma de elegir y disponer físicamente los caracteres que componen la sintaxis del lenguaje; por ejemplo, la profundidad del sangrado, la posición de las llaves de apertura y cierre de bloque, los saltos de línea o el convenio de elección de nombres para los identificadores (variables, clases, funciones, paquetes, etc.). Algunos estilos tienen nombre propio, como las Java Coding Conventions de Sun, el estilo K&R para C (inspirado en los ejemplos del libro correspondiente), el estilo ANSI C, el estilo Win32 definido por Microsoft (notación húngara para los identificadores), etc.

Los proyectos deben elegir un estilo y seguirlo. No se trata de un capricho: utilizar un estilo uniforme facilita la propiedad colectiva del código, simplifica el control de versiones, y en general, aumenta la productividad.

Existen una multitud de herramientas libres para facilitar el seguimiento de estándares de codificación. Cualquier entorno de desarrollo permite definir los parámetros del estilo y formatear automáticamente el código, o bien señalar de alguna forma los puntos del código que no se ajustan. Fuera del editor, herramientas como Indent, Jalopy, JCSC o CheckStyle son extremadamente potentes. Permiten encontrar violaciones del estándar, formatear automáticamente grandes cantidades de ficheros o definir elementos comunes (como insertar un comentario al comienzo de cada fichero haciendo referencia a la licencia del programa).

3.6. Herramientas para dar soporte al proceso

En esta categoría se encuadran las aplicaciones que, en lugar de concentrarse en una práctica concreta, sirven para gestionar el proceso de desarrollo de un proyecto mediante XP. La aplicación más conocida para este menester es XPlanner. Permite gestio-

nar las historias de usuario (con tarjetas virtuales), las iteraciones, los proyectos y las tareas. Genera las métricas que permiten estimar la velocidad del equipo y exporta informes en múltiples formatos.

Sin embargo, este tipo de aplicaciones (generalmente basadas en interfaz web) no son todo lo cómodas y flexibles que cabría desear. Por ejemplo, las tarjetas de cartulina son mucho más expresivas, puesto que se pueden llevar fácilmente a una reunión en un bolsillo, pueden ser colocadas sobre una mesa reflejando distintas agrupaciones, colgadas en un tablón de corcho, retiradas del tablón por los programadores que se ocupan de ellas, intercambiadas entre personas, tachadas, reemplazadas por otras, etc. Esta plasticidad se sacrifica cuando se reemplazan las tarjetas de cartulina por una aplicación web, aunque obviamente se obtienen otras ventajas: las tarjetas virtuales no se pierden, son más fáciles de buscar, mejoran la trazabilidad y pueden ser consultadas desde localizaciones distintas.

4. Conclusiones

La programación extrema y el software libre tienen algunos puntos en común, en los que es posible identificar coincidencias muy significativas, pero también tienen aspectos difícilmente conciliables. La programación extrema se beneficia de la existencia de un gran número de herramientas de software libre que permiten aplicarla con gran productividad. A su vez, el software libre puede inspirarse en algunas de las prácticas de la programación extrema (como el diseño simple, el juego de la planificación y la refactorización) para refinar su modelo de desarrollo y controlar mejor sus proyectos.

Referencias

- [1] Kent Beck. *Test-Driven Development*. Addison-Wesley Professional, 2002.
- [2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.

- [3] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addison-Wesley Professional, 2000.
- [4] C2Wiki. Combining open source and xp. 2005. <http://c2.com/cgi/wiki?CombiningOpenSourceAndXp>.
- [5] Mike Cohn. *User Stories Applied*. Addison-Wesley Professional, 2004.
- [6] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [8] Erik Hatcher and Steve Loughran. *Java Development with Ant*. Manning, 2002.
- [9] Richard Hightower and Nicholas Lesiecki. *Java Tools for eXtreme Programming*. Wiley, 2001.
- [10] Miguel Jaque. Trabajando con programación extrema - experiencias reales. V Jornadas de Software Libre en Asturias, 2005. <http://www.asturlinux.org/jornadas2005/programacion-extrema.php>.
- [11] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Professional, 2004.
- [12] Vincent Massol and Ted Husted. *JUnit in Action*. Manning, 2003.
- [13] Andrew McKinlay. Extreme programming and open source software, November 2000. <http://www.advogato.org/article/202.html>.
- [14] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 1999.
- [15] Gregorio Robles and Jorge Ferrer. Programación extrema y software libre. V Congreso HispaLinux, 2001. <http://es.tldp.org/Presentaciones/200211hispalinux/gregorio2/progm-ext-soft-libre-html/index.html>.
- [16] Gregorio Robles and Jorge Ferrer. Programación extrema, software libre y aplicabilidad, August 2003. <http://www.willydev.net/descargas/Articulos/General/xplibreap.Aspx>.
- [17] Craig Walls and Norman Richards. *XDoclet in Action*. Manning, 2003.