PROMESAS INCUMPLIDAS: SOBRE LOS MÉTODOS ÁGILES

Miguel Ángel Abián
mabian AT aidima DOT es

Resumen: En este artículo se critican las metodologías ágiles, especialmente la programación extrema (XP). Se acusa a estas metodologías de haber contribuido a que el desarrollo de software se haya convertido en un negocio como la moda, donde cuenta más la publicidad y la imagen que la calidad. La mezcla irresponsable de misticismo y propaganda exagerada de los métodos ágiles ha atraído a una generación de jóvenes programadores, sin que hayan probado ser mejores que los métodos anteriores. Su ausencia de resultados contrastables exitosos ha retrasado y retrasa el desarrollo de la ingeniería del software, que precisa más hechos y menos propaganda.
Para el lector no familiarizado con las metodologías ágiles, hay una introducción neutra a ellas en el Apdo. 2, basada en las ideas y declaraciones de sus fundadores. En ese apartado también se comparan con algunos modelos tradicionales de procesos de desarrollo de software (modelo en cascada, en espiral, iterativo, y el análisis y diseño orientado a objetos).

Abstract: In this article the agile methodologies are criticized, specially the eXtreme Programming (XP). These methodologies are accused of having contributed that the development of software has turned into a business as the fashion, where it counts more the publicity and the image than the quality. The irresponsible mixture of mysticism and exaggerated propaganda of the agile methods has attracted a generation of young programmers, without these methods have demonstrated to be better to the previous methods. Their absence of successful contrastable results has delayed -and delays- the development of the software engineering, which needs more facts and less propaganda.
For the reader without experience in the agile methodologies, there is a neutral introduction to them in the Sect., 2 based on the ideas and declarations of their founders. In that section, they are also compared with some traditional software development process models (waterfall model, spiral model, iterative model and object-oriented analysis and design).

Keywords: Extreme programming, XP, agile methodologies, agile methods, C3, Kent Beck

ÍNDICE

1. Introducción. Página 3
2. Panorámica de los métodos ágiles para no iniciados. Página 9
   - 2.1. Antes de los métodos ágiles. Página 9
   - 2.2. Panorámica de los métodos ágiles. Página 14
   - 2.3. La programación extrema. Página 20
   - 2.4. Otros métodos ágiles. Página 40
3. Contra los métodos ágiles. Página 43
   - 3.1. Sobre hechos y otras menudencias. Página 44
   - 3.2. La importancia de los requisitos. Página 47
   - 3.3. Problemas comunes a los métodos ágiles. Página 49
   - 3.4. Limitaciones inherentes a los métodos ágiles. Página 51
   - 3.5. Falta de rumbo: ¿cómo se puede saber si se ha llegado a la meta si se desconoce dónde está? Página 52
   - 3.6. El manifiesto ágil. Página 54
   - 3.7. Hacer una metodología ágil es sencillo: coja prácticas de otras metodologías o derivadas del sentido común y llévelas al extremo. Página 55
   - 3.8. *Extreme Programming Explained: Embrace Change*. Si esto es explicar... Si esto cambia la manera en que se desarrolla el software… Página 56
   - 3.9. La curva de Boehm y la programación extrema. Página 59
   - 3.10. La programación extrema y Smalltalk. Página 61
   - 3.11. El proyecto C3: un Titanic en version extrema Página 62
   - 3.12. La metamorfosis de la metáfora. Página 65
   - 3.13. La programación en parejas. Página 65
   - 3.14. El código fuente no es el diseño. Página 67
   - 3.15. El código fuente no es la documentación. Página 69
   - 3.16. Refactorizar sin ton ni son es diseñar de mala manera. Página 70
   - 3.17. Las pruebas. Página 71
   - 3.18. El papel del cliente “en casa”. Página 71
   - 3.19. Los contratos de alcance parcial. Página 72
   - 3.20. El poder de la palabra. Página 74
   - 3.21. No al *carpe diem*. Página 74
   - 3.22. Elitismo y misticismo: una mala mezcla. Página 77

Soy profesor de método científico, pero tengo un problema: el método científico no existe. Sin embargo, hay algunas reglas generales sencillas que resultan bastante útiles.
Karl Popper

Sobre la mesa de Slothop hay un viejo periódico que parece impreso en español. Las páginas por las que está abierto contienen una caricatura política que muestra a una hilera de hombres de mediana edad con togas y pelucas en una comisaría de policía donde un agente sostiene un bulto blanco..., no, es un bebé, con una etiqueta en el pañal donde se lee La Revolución..., oh, todos reclaman a la pequeña revolución como suya, todos esos políticos discuten y se pelean como si fueran un montón de madres putativas.
El arco iris de gravedad. Thomas Pynchon

1. Introducción.

"*Probar de nuevo, fracasar otra vez, fracasar mejor*." Ése parece ser el lema de la ingeniería del software. Las pruebas y los fracasos mejores corren ahora a cuenta de las metodologías ágiles.

Criticar algo en una época posmoderna no es fácil. Como se rechaza la crítica racional, basada en hechos, puede defenderse casi cualquier postura. Todo vale: las reglas son impuestas por el mercado, la televisión, la moda o las feroces campañas de publicidad. Todo divierte y entretiene.

Las ciencias consideradas *duras* (la física, la química) no se han librado de caer en el posmodernismo. La ingeniería del software también ha sufrido una profunda inmersión en las engañosas aguas del posmodernismo. Las metodologías ágiles reflejan el debilitamiento de los criterios basados en los hechos o en la experiencia. La popularidad de la programación extrema (el método ágil más popular) se debe a varios factores: **publicidad, publicidad y más publicidad**. El carisma de Kent Beck (su creador: mitad ingeniero con talento, mitad altavoz de grandes almacenes) se ha impuesto sobre todas las dificultades de este método, que no son pocas. Incluso los fracasos más estrepitosos se han convertido, contra todo pronóstico, en aparentes triunfos (tal y como mostraré en el Apdo. 3.11).

El esfuerzo que han dedicado los metodólogos ágiles a crear o fundamentar las prácticas que proponen es nulo. Todas han sido cogidas o adaptadas de métodos anteriores, de propuestas de otros autores o de teorías matemáticas, como la de sistemas complejos. Si queremos imaginarnos la originalidad de estas personas y el trabajo duro que han realizado, podemos recordar aquel anuncio de Coca Cola donde el presidente de la empresa tenía que escoger un nuevo envase, de plástico, para el producto. Sus empleados le presentaban varios envases, uno de los cuales era idéntico al de la tradicional botella de cristal, pero de plástico. El presidente elegía ése. El anuncio acababa con la frase "Duro trabajo el del presidente de la Coca Cola".

En general, las metodologías ágiles han comprendido un hecho esencial en cualquier sociedad posmoderna. A saber, que es inútil descalificar a los rivales basándose en hechos y pruebas; para triunfar hay que acudir a la publicidad: hay que proclamar “Venimos de ver el futuro del software y os traemos [nombre en mayúsculas de la metodología]”, “Somos ágiles, somos extremos: paseamos por el afilado borde de la tecnología punta”, “Somos La Revolución... la metodología que acabara con todas las metodologías de software”.

Dicho con otras palabras, más sinceras: “Nos da igual que se desarrollen o no mejores sistemas con nuestras prácticas: queremos cobrar honorarios elevados y gozar de nuestros minutos de fama”. Y han logrado ambos objetivos, desde luego. Hasta el punto de convertir el desarrollo de software en un negocio similar a la moda. Olvidadas las pretensiones de construir sistemas de software robustos y eficaces, la ingeniería del software se ha convertido en una pasarela de modelos donde las metodologías desfilan mostrando las ropas de la última temporada (hoy, programación en parejas; mañana, misticismo de la *new age*; pasado mañana, Mao y la programación extrema). Pasen y vean, señoras y señoras... *And what costume shall the poor girl wear* / *To all tomorrow's parties* / *A hand-me-down dress from who knows where* / *To all tomorrow's parties.*

Figura 1. En esto se ha convertido el desarrollo de software. No es mal negocio...

Existe también cierta similitud entre la proliferación de metodologías ágiles y la de movimientos arquitectónicos durante el siglo XX. No hay, sin embargo,

que forzar la comparación: en arquitectura se discutía por materiales, diseño, etc. Nunca por el objetivo último: construir edificios sólidos, cumpliendo los plazos y presupuestos fijados. Un arquitecto que construya edificios que se caen pronto pasará al olvido o la ignominia, por muy vanguardista que sea o bien publicitado que esté. En los métodos ágiles, el construir software que funcione no parece ser una prioridad *real*, si nos atenemos a los hechos (como veremos en el Apdo. 3.1).

Los métodos ágiles han sabido aprovecharse muy bien de los mecanismos ocultos de la publicidad: "Os llevo tres temporadas de ventaja, y siempre me las apaño para que os sintáis frustrados. […] Os drogo con novedad, y la ventaja de lo nuevo es que nunca lo es durante mucho tiempo. Siempre hay una nueva novedad para lograr que lo anterior envejezca. […] Necesitáis urgentemente un producto, pero inmediatamente después de haberlo adquirido necesitáis otro". Estas frases, escritas por Frédéric Beigbeder en el libro *99 Francs* (*13'99 €uros*, en la edición española), exponen muy bien los artimañas de las que se valen los métodos ágiles y la publicidad.

Resulta sumamente significativo que Scott W. Ambler (creador del *Agile Modeling*) reconociera, en el turno de preguntas de la conferencia *Agile Modeling and Software Quality* (12ª *International Conference on Software Quality*, 2002), que no podía afirmar nada acerca de si "la XP u otros métodos ágiles" tenían en cuenta "la consistencia, la seguridad, la integridad o la privacidad" según "reglas o regulaciones nacionales o internacionales". ¿Se imaginan a un miembro de una corriente arquitectónica diciendo que no puede decir nada sobre si sus edificios pasan las normativas aplicables? ¿O reconociendo que construye edificios sin tener en cuenta los requisitos legales de calidad? En el mejor de los casos, el público tacharía al arquitecto de irresponsable y chapucero. En la ingeniería del software, las cosas discurren de otra forma... Ambler consiguió aplausos por doquier. Incluso firmó autógrafos.

Ante este espectáculo bufo, el cliente anda perdido (muchos desarrolladores también; pero ellos cobran: el cliente es quien paga). Unos se creen lo de ser ágil, pero tampoco quieren ser extremos. Otros piensan que de perdidos al río y apuestan por la programación extrema y lo que haga falta, con la condición de que el presupuesto salga ajustadito. Algunos, más realistas, se limitan a pedir "algo que funcione".

Poco a poco, los clientes van saliendo trasquilados con los métodos ágiles: los proyectos se alargan como siempre, fracasan como siempre y cuestan lo de siempre (o un poco más). Pero no ocurre nada: el programador se ha divertido, que es lo importante. Como el cliente nunca sabe lo que quiere, al menos que se diviertan los programadores...

Este artículo de opinión es una crítica a los métodos ágiles ***actuales*** (nada impide que en el futuro se puedan desarrollar métodos ágiles que funcionen o que se puedan mejorar los que ahora existen). Por motivos de espacio, me centraré especialmente en la programación extrema (***Ex**treme **P**rogramming* o XP).

El artículo se divide en dos partes: en el apartado 2 hago una breve panorámica general de las metodologías ágiles, centrándome en la programación extrema. Esta panorámica es neutra y neutral. Me limito a explicar a grandes rasgos, desde el punto de vista de sus creadores, lo que

propugnan. Puede parecer raro que me dedique a describir las metodologías ágiles para después criticarlas (no soy un caso de desdoblamiento a lo Dr. Jekill y Mr. Hyde, palabra); pero creo que el lector no iniciado en ellas tiene derecho a recibir una visión imparcial. Luego, que él o ella decida. Experimentar, salvo con rayos-X, con nuevos fármacos o con el dinero de uno, suele ser beneficioso.

En el apartado 3 desarrollo una crítica de la XP y de sus prácticas, extensible en parte a todas las metodologías de desarrollo ágiles. Sí que señalo algunos aspectos positivos de los métodos ágiles, pues los tienen, pero son menos numerosos de lo que yo desearía.

Mi relación con los métodos ágiles comenzó en la década de los noventa, durante la cual seguí con atención su surgimiento y difusión. Al principio, cuando aún no los conocía bien, estaba esperanzado. Hacían muchas promesas y todo apuntaba a que iban a cambiar el elemento clave del desarrollo de software: el proceso. Leyendo los textos básicos de la programación extrema y de algunas metodologías ágiles no tan conocidas, comenzaron a surgirme dudas acerca de su eficacia y sus fundamentos. Nunca había sido un converso, desde luego, pero pronto me convertí en un escéptico. Ahora tengo la certeza de que bastantes de los principios de las metodologías ágiles son erróneos o inútiles.

Conforme fui leyendo los textos, me fui dando cuenta de que las metodologías ágiles son como una máquina cuyos engranajes internos, ocultos a la vista o al escrutinio, están entremezclados sin orden ni concierto. Si la máquina funciona bien, nadie mirará dentro. Pero si no, alguien acabará abriéndola y dándose cuenta del caos mecánico en su interior. Hasta ahora, ningún mecánico se ha dedicado a enderezar las arandelas, tuercas y tornillos o a lubricar los rodamientos. Iniciativas como el *Agile Modeling* quizá pongan cada engranaje en su sitio. O quizá no. Quizás los juguetes rotos no se puedan reparar.

Hasta ahora ni siquiera me había planteado escribir nada acerca de las metodologías ágiles: hace tiempo que no las considero relevantes. En realidad, las había olvidado: para mí son una moda que pasará pronto, si no lo está haciendo ya. Su ausencia de un corpus teórico o empírico las destina a un rinconcito modesto y sepulcral dentro de la corta historia de la construcción de software. La gente se divertirá, se entretendrá; los programadores se sentirán mejor tratados... y a por la siguiente hondonada de metodologías. Me he decidido a escribir este texto porque tengo la sensación de que están retrasando más de lo conveniente el desarrollo de la ingeniería del software.

Mientras escribo todo esto tengo muy presentes las frases de Roosevelt acerca de los críticos: “[…] el hombre que realmente cuenta en el mundo es el que hace cosas, no el mero crítico: el hombre que realmente hace el trabajo, incluso aunque sea ruda e imperfectamente, no el hombre que sólo habla o escribe sobre cómo deberían hacerse” y “La crítica es necesaria y útil; a menudo es indispensable; pero nunca puede tomar el lugar de la acción, ni siquiera ser un pobre substituto para ella. La función del mero crítico es de una utilidad muy subordinada. Es el que actúa quien cuenta realmente en la batalla por la vida, y no el hombre que mira y dice cómo debería librarse la lucha, sin compartir él mismo la tensión y el peligro”.

Si critico los métodos ágiles es porque trabajo en proyectos y programo; no soy un teórico ajeno a los problemas prácticos y humanos del desarrollo de software. Me equivoco a veces, desde luego; pero al menos intento evitar los errores que parecen errores desde el principio.

El mundo de las metodologías, ese río permanentemente revuelto y del que muchos pescadores poco comprometidos con los resultados sacan ganancia, genera ingenieros del software y programadores de muchas clases. Hay dos tipos extremos: los escépticos y los eclécticos.

Los primeros desconfían de cualquier palabra que comience con eme, y si aparentan creer ante los demás en las metodologías es por motivos laborales o sociales: no resulta fácil ser ateo en un mundo donde casi todos son creyentes.

Los segundos recopilan prácticas de diversas metodologías, cuantas más mejor, y las atesoran como los que coleccionan mariposas o recetas de cocina. Siguen, pues, una línea de eclecticismo metodológico. En sí misma, la estrategia de seguir prácticas y reglas de distintas metodologías no tiene ninguna ventaja sobre cualquier otra: es la metodología de los que no tienen metodología.

Personalmente, no creo que haya metodologías universales, infalibles o aplicables a cualquier situación o contexto. Sí pienso, sin embargo, que existen métodos más generales –y de mejores resultados– que otros. Saber qué reglas y metodologías aplicar en cada caso es más importante y útil que seguir ciegamente siempre las mismas. La racionalidad no se demuestra siguiendo con rigidez unas reglas o unas recetas.

Cualquier persona que lleve cierto tiempo en el negocio del desarrollo de software sabe lo siguiente: ***en determinadas circunstancias*** conviene practicar el transfugismo metodológico. Esto es, que conviene saltarse reglas de la metodología seguida en el proyecto y adoptar otras (o incluso ninguna en concreto). Existe una especie de consenso para no hacer públicas estas desviaciones; pero son una práctica nada infrecuente, sobre todo en proyectos sin muchos precedentes.

Los métodos no se deben menospreciar o rechazar por ello: son guías, no La Verdad Absoluta. Si las metodologías se siguieran siempre al pie de la letra, no habría avances ni nuevas metodologías. No ocurre nada por saltarse los principios de un método, por sensatos que parezcan, siempre que se sepa ***por qué*** y ***para qué*** se hace.

Desde luego, no tengo ningún ungüento mágico para tapar los agujeros que presenta la construcción de software; esto es un mero artículo de opinión. Mis esperanzas no están puestas en las metodologías ágiles. No obstante, existen aspectos concretos que son de interés, si bien otros los habían explicado antes y de mejor manera.

Algún lector puede pensar que debería formular soluciones, en lugar de críticas. Respeto, pero no comparto, esa opinión. Si fuera médico, criticaría a cualquier curandero que propugnara que la sopa de pollo cura el cáncer hepático terminal, aunque no tuviera ninguna solución para esa enfermedad.

Por si al lector le sirve de algo, diré que generalmente trabajo con UML y con el Proceso Unificado de Rational –ahora, Proceso Unificado de Rational-IBM)– en forma iterativa (esto no quiere decir que recomiende usar siempre el RUP). Como muchos proyectos están relacionados con las necesidades empresariales y la interoperalidad del software, suelo usar también normas

como ENV 12204 (*Constructs for Enterprise Modelling*), ENV 13550 (*Enterprise Model Execution and Integration Services*), ISO 14258 (*Concepts and rules for enterprise models*) e ISO 15704 (*Requirements for enterprise reference architecture and methodologies*). En todos los proyectos se usan cuantificadores de la calidad en cada etapa de desarrollo (métricas, cumplimiento de ciertas pruebas, etc.). Para revisar el código, aparte de las pruebas, me gusta utilizar técnicas extraídas de métodos como el TSP (*Team Software Process*) y el PSP (*Personal Software Process*), que dan muy buen resultado a la hora de detectar fallos en el código.

Creo que usar UML es una buena idea. Los más veteranos recordarán las guerras de notaciones en los años ochenta y noventa, y la proliferación de libros y artículos con distintas notaciones. UML, al convertirse en el estándar del OMG, se ha impuesto como un lenguaje de modelado común, como una base para todos los analistas y diseñadores.

Sin embargo, UML no es la panacea, no es el lenguaje de modelado que solucionará todos los problemas. Veo (y he experimentado) dos problemas importantes en UML. A saber: la complejidad de las versiones 1.x y la necesidad de usar otras herramientas de modelado aparte de UML.

Las versiones 1.x son demasiado densas y prolijas; sus definiciones tampoco resultan siempre tan precisas como cabría esperar. Según iba leyendo las distintas versiones de UML 1.x, recordaba una famosa verdad de la programación: “El 80% del código está escrito para tratar situaciones que se darán el 20% de las veces”. Hubiera sido mucho más útil definir un núcleo del lenguaje que cubriera el 80% de las situaciones más comunes que se plantean al modelar sistemas de software, y que sólo ocupara un 20% del lenguaje. Parte de estos problemas se han solucionado en UML 2.0, pero es una pena no haberlos evitado desde el principio.

Como herramienta de modelado, UML tiene sus limitaciones. En cualquier sistema real de software hay especificaciones imposibles de representar con un lenguaje de modelado visual. A veces, uno recurre a especificaciones escritas o a simulaciones por ordenador. Aunque pocas veces se formula como tal, muchos analistas y diseñadores se hacen esta pregunta: ¿Qué es mejor: un análisis-diseño visual o un análisis-diseño en forma de documento escrito? Dicho de otra manera, ¿vale una imagen más que mil palabras? La respuesta yace más en territorio de la epistemología que en el de la construcción de software.

En cuanto a la XP, no he trabajado directamente en ningún proyecto de programación extrema. Sí he seguido de cerca cuatro proyectos extremos. Tres de ellos fracasaron con evaluaciones que varían entre “Necesita mejorarse” (una aplicación en red que tarda 25 minutos en arrancar y que consume los recursos del servidor como si fuera un vampiro sediento no incita a la alegría) y “Tirar más dinero aquí es inútil” (el equipo de desarrollo reconoció que no sabía cómo continuar y se rescindió bilateralmente el contrato). El cuarto se debate ahora entre la vida y la muerte, a la espera de un pulmón artificial. Por motivos de confidencialidad, no puedo dar más detalles por ahora. En un futuro no muy lejano me gustaría escribir algún artículo sobre ellos.

2. Panorámica de los métodos ágiles para no iniciados.

A continuación se presenta una breve panorámica de los métodos ágiles, situándolos antes en el contexto de los procesos de desarrollo de software. Por motivos de espacio –y de paciencia– me centro sobre todo en la XP, pero casi todas las metodologías ágiles comparten muchas de sus características. Como mínimo, todas subrayan estas ideas: comunicación entre programadores, refactorización, desarrollos incrementales y pruebas.

El lector interesado en comparar la XP con otros procesos de desarrollo (como *Rational Unified Process* o *Feature Driven Development*) puede consultar el recomendable y práctico artículo *Procesos de desarrollo* de Alberto Molpeceres, publicado en **javaHispano**. También puede resultar muy útil visitar la página web www.agile-spain.com. Contiene mucha información interesante y, pese al nombre, está hecha con independencia, buen gusto y honradez.

Las ideas sobre la XP del subapartado 2.3 se basan, en parte, en las primeras ediciones estadounidenses de los libros *Extreme Programming Explained: Embrace Change* (Kent Beck) y *Planning Extreme Programming* (Kent Beck y Martin Fowler). Desconozco si ha habido cambios sustanciales en posteriores ediciones. Las traducciones de todos los textos son mías.

2.1. Antes de los métodos ágiles.

En el software, las metodologías exitosas vienen a ser, aparte de un medio para ganarse la vida, las fases finales del ciclo de vida de las ideas de desarrollo de software antes de pasar a formar parte del conjunto de conocimientos que cualquier programador conoce.

Un proceso de desarrollo de software es un método para desarrollar software, el cual organiza el trabajo en un número de tareas y pasos separados. En el fértil ecosistema de las metodologías de software, las metodologías ágiles tienen antecesores, así como competidores. Algunas de las metodologías contrincantes más destacadas se basan en estos modelos de procesos de desarrollo de software:

- Modelo en cascada.
- Modelo en espiral.
- Desarrollo iterativo.
- Desarrollo iterativo con entregas sucesivas.
- Análisis y diseño orientado a objetos (OOAD).

El **modelo en cascada** es la aproximación más simple y rígida para el desarrollo de software. Consiste en seguir una serie de etapas o fases en cascada, claramente separadas, que se llevan a cabo en orden, sin que pueda pasarse a otra sin haber concluido las anteriores. Las etapas son éstas: recopilación de requisitos, análisis, diseño, implementación del código, pruebas y distribución. La secuencia se representa en la figura 2.

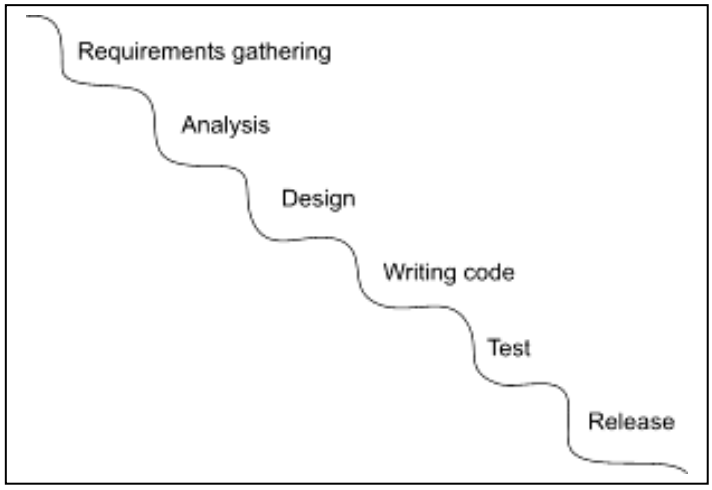

Figura 2. Esquema del modelo en cascada.

El nombre de modelo en cascada resulta irónicamente apropiado: el agua que cae de una cascada no puede (por ella misma) retroceder hacia arriba para volver a caer.

Desde luego, se trata de un modelo demasiado rígido. Lo inevitable en un proyecto real es que aparezcan problemas en cada etapa y que haya que revisar las etapas anteriores. Por otro lado, el cliente no recibe el producto hasta que se acaba la última etapa; lo cual suele provocar que éste no quede satisfecho con los resultados, bien porque éstos han cambiado a lo largo de proyecto, bien porque lo que el cliente pidió no era realmente lo que quería. Algunos autores creen que la captura de requisitos completa y la etapa de análisis lleva muchas veces a “la parálisis del análisis” (típico en proyectos con vastos dominios de problema). Sin embargo, no creo que ese problema sea achacable al modelo en sí, sino al uso (o mal uso) de las técnicas de modelado.

Este modelo suele usarse en la enseñanza por su simplicidad y sencillez; eso no significa que los profesores de ingeniería del software no sepan cómo se trabaja en los proyectos reales. Lo usan porque sirve para explicar muchas técnicas útiles y prácticas, aun cuando no se apliquen en el orden especificado por el modelo en cascada. Resulta similar a estudiar un péndulo; se comienza, primero, por considerarlo como un oscilador armónico simple para pequeñas oscilaciones. Cualquier péndulo real presenta no linealidades, incluso efectos caóticos: explicar el problema completo a alumnos que aún no conocen las técnicas matemáticas para tratarlas (teoría de ecuaciones diferenciales) resultaría absurdo y antipedagógico. Actuando así, se consigue que los alumnos comprendan los movimientos oscilatorios simples, aplicables a muchas situaciones, y una primera aproximación al estudio de una situación real.

El **modelo en espiral** no es tan ingenuo como el anterior: considera que puede haber fallos en cada etapa. Originalmente se propuso así:

1) Se realiza el análisis hasta que los desarrolladores piensan que han acabado.

2) Se realiza el diseño hasta que se piensa que se ha concluido con él. En el caso de que se encuentren fallos o problemas en el análisis, se abordan volviendo al análisis (es decir, se realiza otra iteración del análisis).
3) Se pasa el diseño a código. Si aparecen problemas en la traducción, se vuelve al diseño hasta que se solucionan (es decir, se realiza otra iteración del diseño).
4) Se hace que el código pase las pruebas oportunas. Si aparecen errores o fallos, se comienza otra iteración de la escritura de código.
5) Finalmente, se distribuye la aplicación. Los problemas que aparezcan se enviarán a las etapas anteriores.

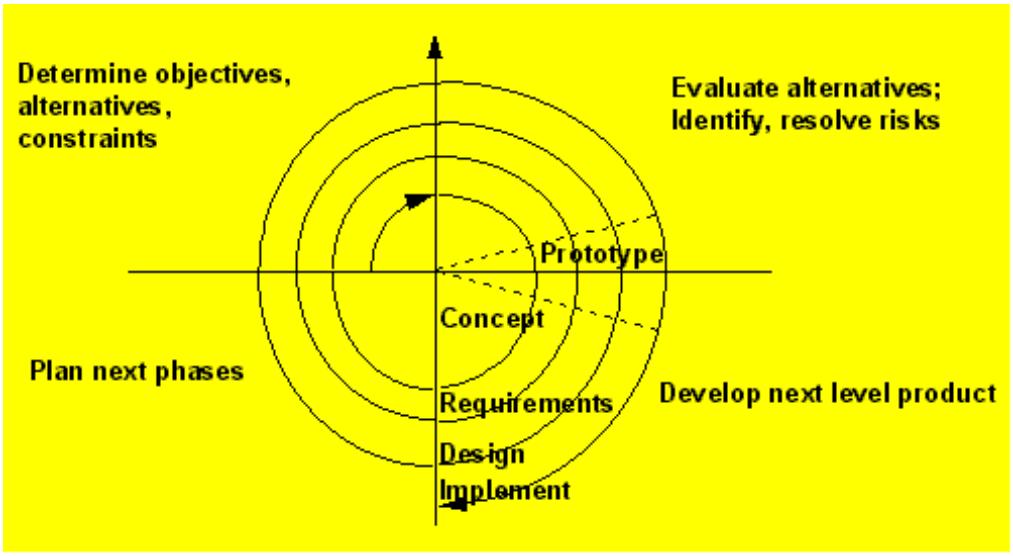

Figura 3. Esquema del modelo en espiral.

Como se aprecia en la figura, las metodologías basadas en el proceso en espiral progresan en el desarrollo de software mediante capas; cada capa o espiral representa una fase en el proceso. No existen fases prefijadas (captura de requisitos, análisis, diseño, etc.): en un proyecto habrá tantas capas como se necesiten. Un prototipo permite a los usuarios determinar si el proyecto va por buen camino, si debería volver a etapas anteriores o si debería concluirse.

El proceso en espiral se introdujo para solucionar los problemas del proceso en cascada, y es la variante de éste más usada en la actualidad. La secuencia de pasos es aún lineal, pero admite retroalimentación.

El modelo en espiral considera que los pasos hacia atrás (las iteraciones posteriores a la primera) son errores. El **modelo de desarrollo iterativo** asume, en cambio, que siempre se van a cometer errores y que, por consiguiente, siempre habrá que efectuar varias iteraciones. Un proyecto basado en este último modelo se construye mediante iteraciones. Cada ciclo o iteración concluye con una versión del sistema en desarrollo que cumple un subconjunto de los requisitos exigibles al sistema final. En cada iteración aparecen todas las fases del modelo en cascada; pero cada iteración sólo está orientada a un subsistema, no al sistema completo.

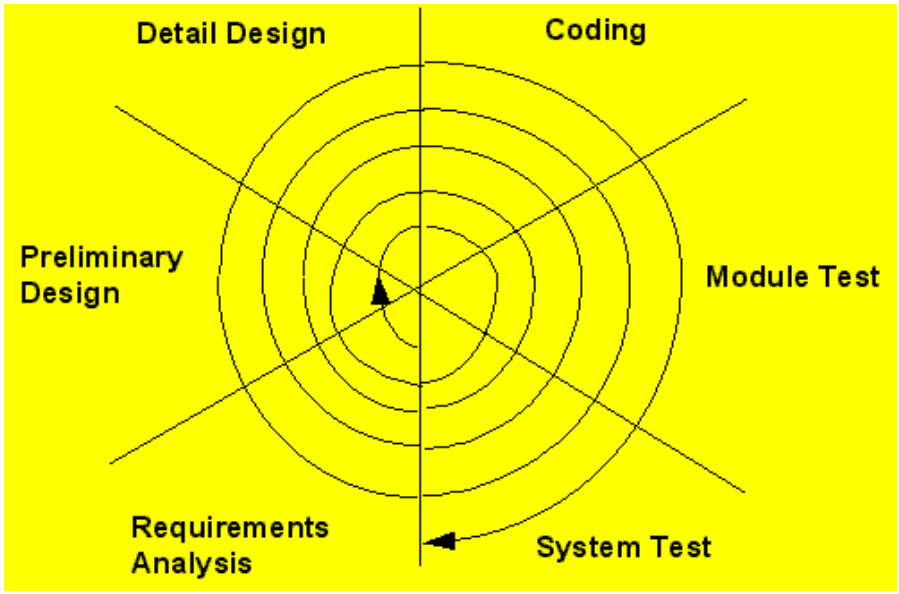

Figura 4. Esquema del modelo de desarrollo iterativo.

Este modelo, al igual que los anteriores, se basa en una sucesión lineal de pasos o etapas. Frente a los otros, presenta algunas ventajas:

1) El desarrollo se divide en iteraciones; en cada una figura un subconjunto de las funciones exigidas al sistema.
2) Pueden abordarse en las primeras iteraciones aquellas funciones críticas o de alta prioridad para el cliente.
3) Los requisitos de las iteraciones aún no implementadas pueden cambiar durante el proyecto, sin que sea preciso modificar el código generado en las iteraciones terminadas.

La evolución de los modelos de desarrollo de software es la historia de la transformación del optimismo en pesimismo. El modelo en cascada considera que no hay errores; el modelo en espiral acepta que puede haberlos; y el modelo iterativo acepta que son habituales.

El **modelo de desarrollo iterativo con entregas incrementales** soluciona un punto débil de los anteriores modelos. A saber: el tiempo transcurrido hasta que el cliente ve el resultado del proyecto (el software no está completamente acabado hasta el final de proyecto). Propugna el lanzamiento de distintas versiones del software durante el desarrollo, de modo que los usuarios pueden probarlo durante el avance del proyecto. En cada lanzamiento se aumenta el número de funciones que incorpora. Así, el cliente no se encuentra con sorpresas al final del proyecto, y puede usar y comprobar el sistema antes de que esté acabado.

El **análisis y diseño orientado a objetos** ha generado una gran cantidad de modelos de procesos. Explicar brevemente cada uno de ellos alargaría este artículo hasta convertirlo en un tomo de respetables proporciones. Así pues, prefiero señalar solamente las tres etapas comunes en todos los modelos de procesos OO:

- Análisis. En esta etapa se modelan las clases básicas y el sistema más elemental que represente los requisitos de los usuarios.
- Diseño. Se refinan las clases básicas para que sean implementadas en un cierto lenguaje, y se sacan a la luz las clases derivadas o secundarias.
- Implementación. Se definen las interfaces de las clases y los métodos de implementación. Finalmente, se escribe el código, se hace que todas las clases pasen sus pruebas unitarias y se comprueba el sistema en conjunto.

Los métodos orientados a objetos no son incompatibles con los basados en los modelos anteriores. Por ejemplo, el proceso de desarrollo unificado de Rational (RUP) es compatible con un modelo en cascada o con un modelo de desarrollo iterativo con entregas incrementales.

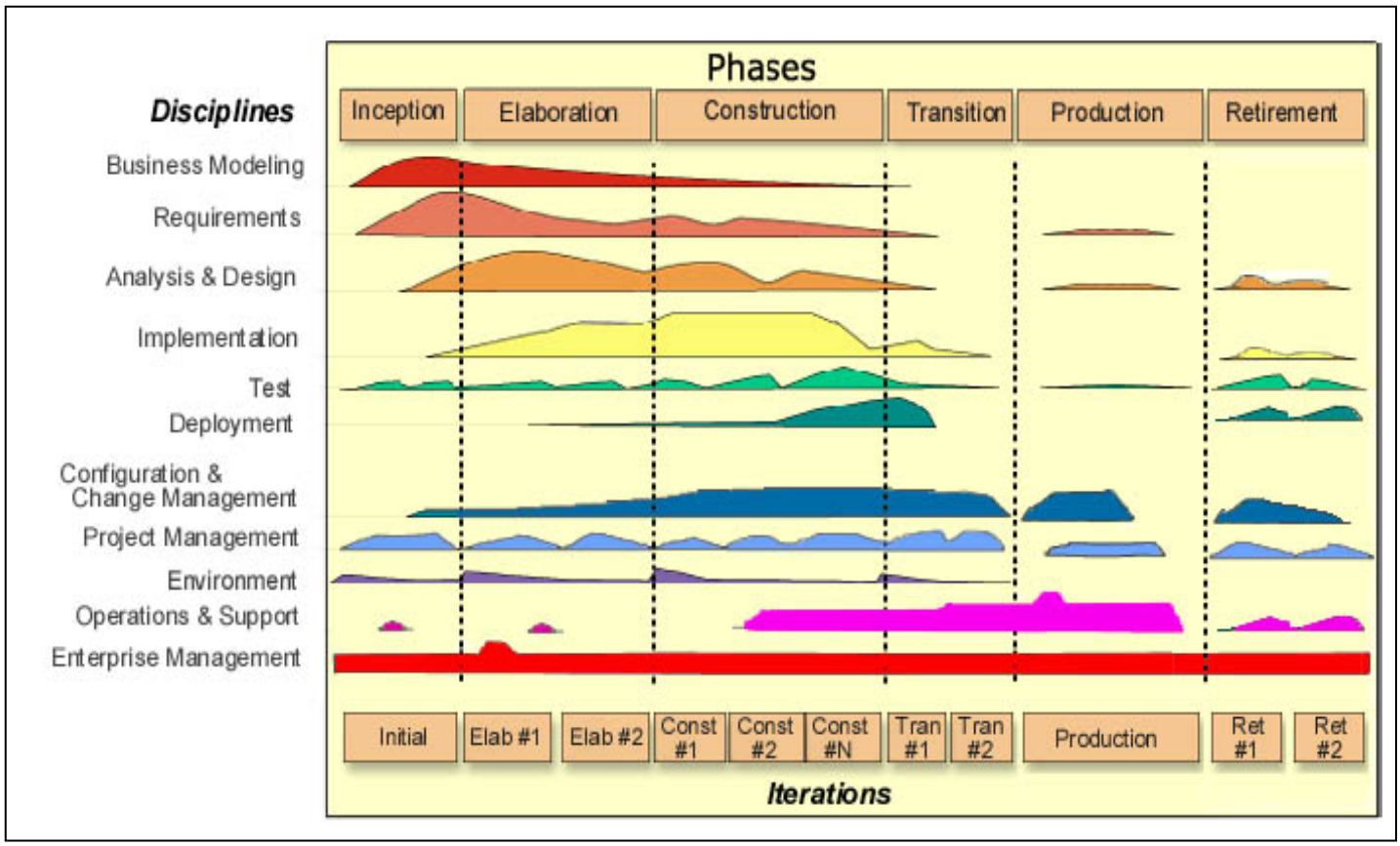

Figura 5. Etapas del Proceso Unificado de Rational. Extraído de la documentación oficial del RUP.

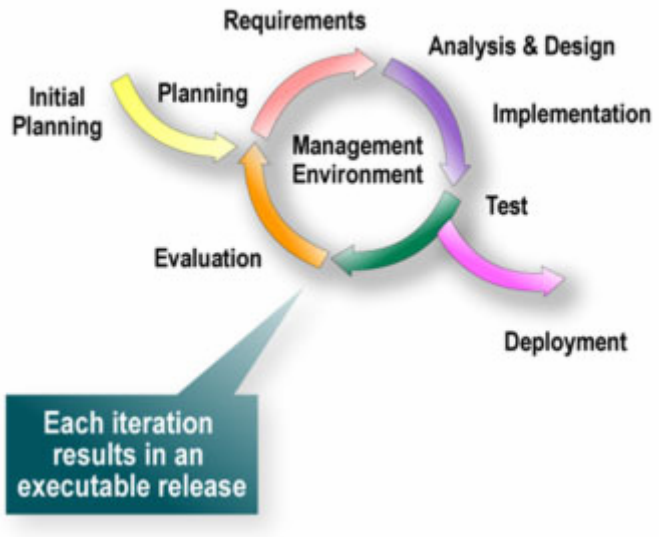

Figura 6. Otra vista de las etapas del Proceso Unificado de Rational. Extraído de la documentación oficial del RUP.

Visto lo anterior, podremos comprender las críticas que hacen los métodos ágiles a las metodologías anteriores, así como sus semejanzas y diferencias con aquéllos.

2.2. Panorámica de los métodos ágiles.

Nota: en este artículo uso ***ágil*** como la capacidad de crear cambio y de responder y reaccionar a él.

Las metodologías ágiles surgen en los años noventa como consecuencia del abaratamiento del hardware y de la existencia de entornos integrados de desarrollo baratos y de rápido manejo. Un programador ágil no hubiera tenido mucho sentido –ni trabajo– en la época de las tarjetas perforadoras o cuando el tiempo de acceso a terminales era muy costoso. En esas circunstancias, los programadores llevaban perfectamente diseñados sus programas antes de empezar a escribir código. Un programador que desperdiciara tiempo de computación preguntándose (o preguntando a sus compañeros) cómo debía programar un algoritmo habría sido visto con muy malos ojos.

La popularización de los lenguajes orientados a objetos y de Internet también ha sido decisiva en la actual avalancha de métodos ágiles. Sin los lenguajes OO, poca reutilización del código podría hacerse. Ante lenguajes rígidos como Fortran o Cobol, los métodos ágiles se baten en retirada. Por otro lado, Internet ha modificado sustancialmente las necesidades informáticas de las empresas, aumentando la necesidad de soluciones adaptables y extensibles a nuevas situaciones.

Entre las metodologías ágiles más conocidas está la programación extrema (la más popular), *SCRUM* (desarrollado por Ken Schwaber), el *Adaptive Software Development* (creado por J. A. Highsmith), la familia de

métodos *Crystal* de Alistair Cockburn (pronúnciese "Coburn", a la manera escocesa), el *Feature Driven Development* (obra de Jeff De Luca y Peter Coad) y la *Pragmatic Programming*. En realidad, hablar de las metodologías ágiles como un todo es como hablar de anarquía bien ordenada: cada metodología tiene sus gurús y sus libros, pero todas coinciden en las características que se verán a continuación. En algunos casos, incluso es dudoso que se aplique con propiedad la palabra *metodología*. Por ejemplo, yo consideraría que el desarrollo adaptable de software (*ASD*) es más una noción de la gestión de proyectos que un método para conseguir software de buena calidad.

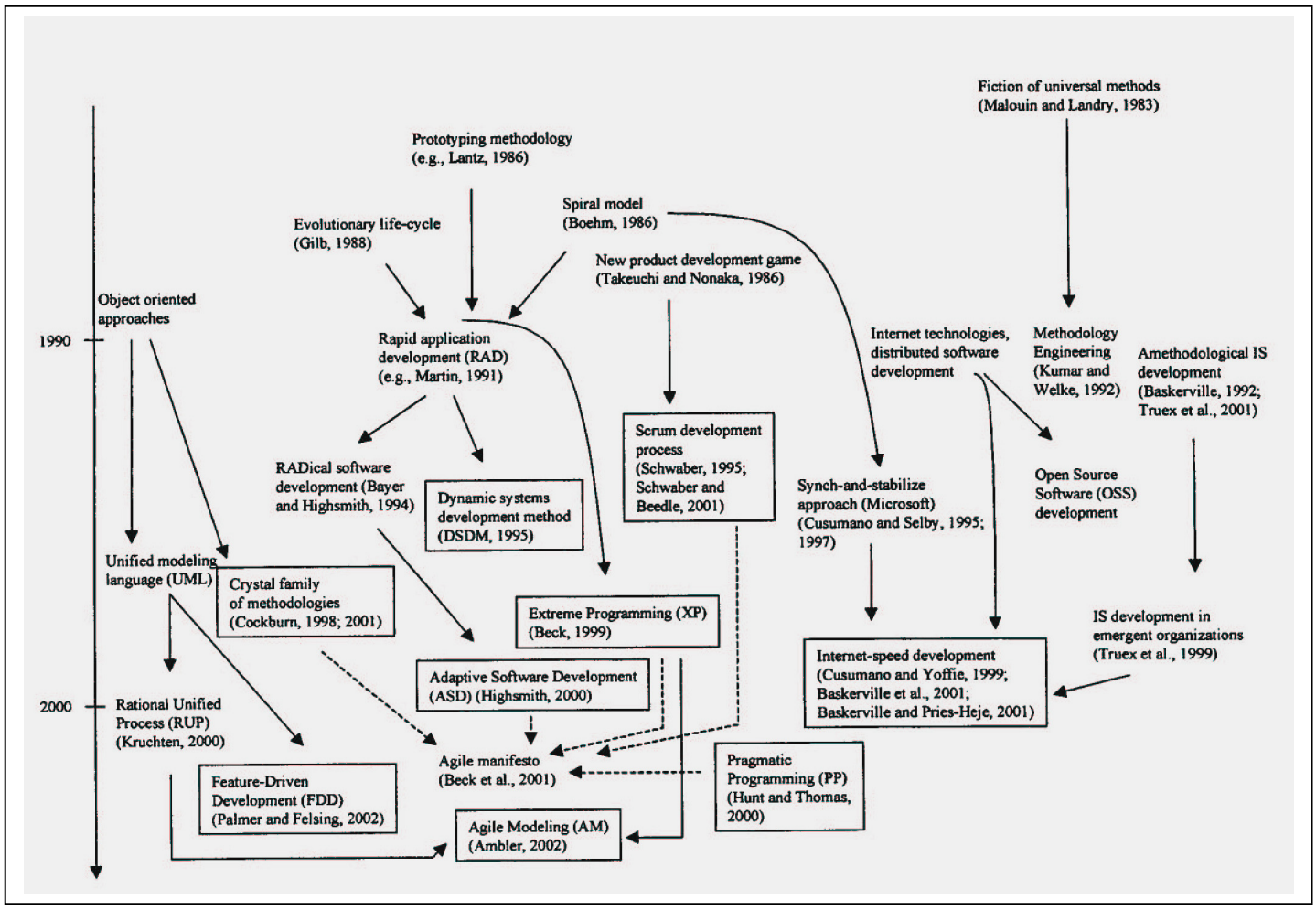

Figura 7. Árbol genealógico de los métodos ágiles. Extraído del artículo *New Directions on Agile Methods*: *A Comparative Analysis* (Abrahamsson et al.).

¿Cómo sabemos si un método es ágil o no? En febrero de 2001 se celebró un taller en Snowbird (Utah, Estados Unidos) en el que se reunieron diecisiete metodólogos ágiles. Su principal objetivo consistió en **definir lo que significa que una metodología sea ágil**. De ahí surgió el manifiesto para el desarrollo de software ágil (http://agilemanifesto.org/), una recopilación de los principios y valores comunes a todos los métodos ligeros. De aquel taller también surgió la Alianza Ágil (http://agilealliance.org/), una organización sin ánimo de lucro que tiene como fines el mejor entendimiento de los métodos ágiles y la creación de condiciones favorables para discutir e intercambiar opiniones sobre ellos.

El manifiesto, que también cuenta con una serie de principios subyacentes (http://www.agilemanifesto.org/principles.html), es un intento de aclarar qué significa que un proceso o metodología sea "ágil". Los principios tras el manifiesto se exponen aquí:

1) Nuestra prioridad más importante es satisfacer al cliente mediante la entrega temprana y continua de software que le aporte valor.
2) Dé la bienvenida a los requisitos cambiantes, aunque sea tarde en el desarrollo. Los procesos ágiles aprovechan el cambio en favor de la ventaja competitiva del cliente.
3) Entregue frecuentemente software que funcione, desde un par de semanas hasta un par de meses, con preferencia por las escalas de tiempo más cortas.
4) Las personas de negocios y los desarrolladores deben trabajar juntos diariamente durante el proceso.
5) Construya software alrededor de individuos motivados. Deles el ambiente y apoyo que necesiten, y confíe en ellos para tener el trabajo hecho.
6) El método más eficiente y efectivo de comunicar información a un equipo de desarrollo y dentro de él es la conversación cara a cara.
7) Las mejores arquitecturas, requisitos y diseños emergen de equipos que se organizan a sí mismos.
8) El software que funciona es la principal medida de progreso.
9) Los procesos ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían poder mantener un paso constante indefinidamente.
10) La atención continua a la excelencia técnica y al buen diseño aumenta la agilidad.
11) La simplicidad –el arte de hacer máximo el trabajo no realizado– es esencial.
12) A intervalos regulares, el equipo reflexiona sobre cómo ser más efectivo; entonces ajusta su comportamiento de acuerdo con ello.

Como considero que esta lista de características es demasiado general, prefiero considerar que las metodologías ágiles son un conjunto de metodologías o procesos de desarrollo de software que tienen en común, entre otras, estas características mínimas:

a) **Orientación a las personas.** Los clientes están, al menos durante cierto tiempo, en el mismo lugar que el equipo de desarrollo; así pueden contestar a las preguntas que surgen durante la marcha del

proyecto. Se fomenta el trabajo en equipo, en el mismo lugar de trabajo, y se prefiere la comunicación directa, cara a cara, entre los desarrolladores.

b) **Rechazo a la burocracia de los métodos *pesados* (como los del proceso en cascada).** Se rechaza el exceso de documentación y se asume que ésta no desempeña un papel primordial en la generación de software de buena calidad.
c) **Adaptación a las circunstancias cambiantes de las empresas.** Se asume que los requisitos del software evolucionan al mismo tiempo que se avanza en su desarrollo, de modo que se proponen prácticas dinámicas, adaptables a los cambios.
d) **Son métodos más adaptables que predictivos.**
e) **Rechazo de la especialización.** Se considera que los desarrolladores no deben realizar siempre las mismas tareas: deben rotar regularmente de puesto.
f) **Desarrollo incremental e iterativo.** Se considera que el software puede, y debe, desarrollarse en incrementos e iteraciones, cuanto más cortas mejor (días, semanas).
g) **Orientación al código.** Se considera que el código es lo único esencial y se afirma que el diseño está en el código, no en modelos independientes del código. La escritura de código y las pruebas se valoran más que el análisis y el diseño.
h) **Rechazo al diseño.** Dado que los requisitos cambian durante el proyecto, se estima innecesario realizar diseños pormenorizados.
i) **Uso de la refactorización.**
j) **Rechazo de la reusabilidad.** Se considera que la reusabilidad no debe guiar la construcción de software.
k) **Reducción de los costes del cambio.** Se intenta que los costes de los cambios no crezcan excesivamente con el tiempo.
l) **Se organizan a sí mismos.** Los equipos ágiles tienen la suficiente autonomía como para organizarse a sí mismos para alcanzar los objetivos de los proyectos.
m) **Son emergentes.** La tecnología y los requisitos emergen durante el ciclo de desarrollo del producto.

Individuos e interacciones	Procesos y herramientas
Software que funciona	Documentación comprensible
Colaboración con el cliente	Negociación de contratos
Responder al cambio	Seguir un plan

Figura 8. Los métodos ágiles valoran más las prácticas de la izquierda que las de la derecha.

Métodos ágiles	Métodos tradicionales
Se basan en heurísticas para la producción de código	Se basan en normas
Aceptan e incluso fomentan el cambio	Mayor o menor resistencia a los cambios
El cliente forma parte del equipo	El cliente no forma parte del equipo, sólo mantiene reuniones con éste
Equipos de trabajo pequeños o medianos	Equipos grandes (>15-20 miembros)
Procesos con pocas reglas	Procesos con muchas normas
Poca documentación	Mucha documentación
Poco análisis y diseño	Mucho análisis y diseño
Conceden poca importancia a la arquitectura de los sistemas	Conceden mucha importancia a la arquitectura de los sistemas

Figura 9. Algunas diferencias significativas entre los dos tipos de métodos.

De los trece puntos señalados en la página anterior, los tres primeros son los más importantes para comprender el espíritu de los métodos ágiles.

Por un lado, los métodos ágiles están orientados hacia las personas. Todas las disciplinas de ingeniería han tendido a construir procesos y métodos que funcionen bien independientemente de las características particulares de las personas que los apliquen. Los métodos ligeros opinan que las capacidades de las personas de un equipo no pueden ser normalizadas u obviadas, y que cualquier proceso realista debería apoyar al equipo de trabajo en el desarrollo de su trabajo. Propugnan que la programación es una actividad divertida, humana y comunitaria; no una actividad de “vaqueros solitarios” como Lucky Lucke. La importancia que dan a la comodidad y al bienestar de los programadores (respeto a sus condiciones de trabajo, jornada laboral de cuarenta horas, etc.) redunda –afirman– en una mayor productividad.

Por otro lado, estos métodos repudian la acumulación de documentos en la construcción de sistemas de software. Mientras que las metodologías más tradicionales (“pesadas”) –como las basadas en el modelo en cascada– necesitan una ingente cantidad de documentos para recoger el conjunto completo de los requisitos del sistema, las ágiles propugnan una visión no

documentocéntrica. Coinciden en su propósito de evitar o reducir al mínimo la documentación para cada tarea y en su firme convicción de que la parte más importante de la documentación es el propio código fuente.

Finalmente, una de las premisas comunes de las metodologías ágiles o ligeras es que resulta imposible conocer todo lo necesario sobre un proyecto de desarrollo de software antes de comenzar el proyecto. En consecuencia –según ellos–, lo mejor es proponer prácticas para adaptarse a los cambios, no para anticiparse a ellos. Su preferencia hacia prácticas que se adapten rápidamente a las circunstancias cambiantes de las empresas es también una crítica hacia los métodos tradicionales.

Para entender su crítica, conviene repasar un poco los métodos tradicionales. Los métodos de desarrollo tradicionales, como los basados en el modelo en cascada, incluyen una fase de identificación de los requisitos, una de diseño, una de construcción, y finalmente el lanzamiento de la primera versión del sistema, sobre la cual se aplicarán las pruebas.

En general, la etapa de diseño puede desarrollarse con la participación del cliente o sin ella. Cuando se concluye esta etapa, al cliente se le puede entregar una versión beta del sistema, una demo o un informe de la marcha del proyecto. Muchas veces, por problemas de plazos, el cliente no recibe más que buenas palabras antes de la fecha de entrega del sistema. En estos casos, cuando el cliente prueba la primera versión del sistema suele encontrarse insatisfecho con los resultados. Que el cliente no quede contento con el sistema no es necesariamente achacable a la incompetencia de los desarrolladores: puede deberse a que el cliente ha cambiado de opinión acerca de los requisitos del sistema; o a que, a pesar de que el sistema cumple los requisitos antes especificados, no lo hace como de verdad quería el cliente. Debido a la separación estricta entre las etapas del proyecto, métodos como los basados en el modelo en cascada se consideran rígidos: no se comportan bien frente a cambios.

Los métodos ágiles se oponen a los rígidos o pesados por varios motivos:

a) La recopilación completa de los requisitos es cara y requiere tiempo.
b) Los requisitos suelen cambiar a lo largo de la realización de los proyectos, con lo que el trabajo realizado puede ser inútil.
c) No son métodos que puedan adaptarse fácilmente a los cambios, y menos aún a los cambios rápidos.
d) Carecen de realimentación entre clientes y desarrolladores.
e) Aunque el sistema final cumpla los requisitos iniciales, el cliente puede no estar satisfecho con los resultados finales (“Esto es lo que pedí, pero no es realmente lo que necesitaba”).

Las metodologías basadas en el proceso de desarrollo orientado a objetos y el proceso de desarrollo iterativo no son tan rígidas como las basadas en el modelo en cascada; pero los métodos ágiles rechazan la importancia que dan al análisis y a la captura de requisitos.

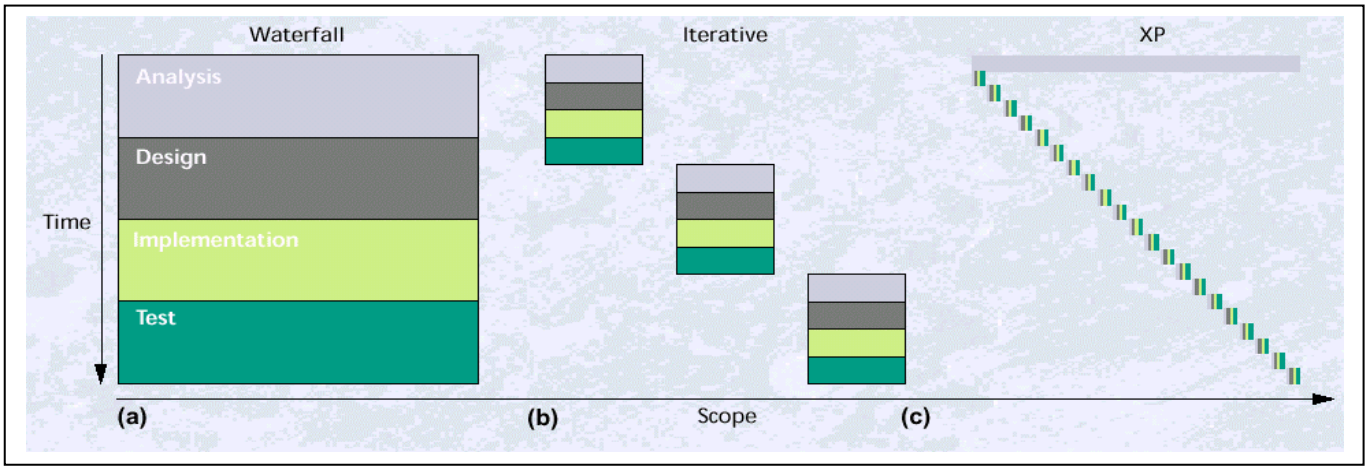

Figura 10. Las etapas de análisis, diseño, implementación y de pruebas en el modelo en cascada, en el iterativo y en la XP.

2.3. La programación extrema.

La **XP** (*eXtreme Programming*; programación extrema), la metodología ágil más popular, fue inventada en 1996, cuando Kent Beck comenzó a trabajar en el proyecto *Chrysler Comprehensive Compensation* (C3) para la empresa DaimlerChrysler. Parte de las ideas de la XP proceden de Ward Cunningham.

El proyecto C3 era un proyecto escrito en Smalltalk que tenía como fin sustituir al sistema de nóminas de la compañía. Había comenzado en 1995, y Beck se incorporó a él a comienzos de 1996. Él fue quien recomendó a la empresa desechar el código ya escrito y comenzar de nuevo. DaimlerChrysler accedió y Kent tuvo las manos libres para inventar y aplicar la nueva metodología. En ocho meses, Kent y su equipo consiguieron producir un sistema piloto que contó con la aprobación del cliente. Sin embargo, en febrero de 2000, el proyecto fue cancelado porque había sobrepasado con creces el presupuesto original y sólo funcionaba para las nóminas de un tercio, aproximadamente, de los empleados de la empresa. El proyecto C3 llegó a tener unas 2.000 clases y unos 30.000 métodos.

El término XP no guarda ninguna relación con el sistema operativo Windows XP; Kent Beck afirma que se llama “programación extrema” porque lleva al extremo ciertas prácticas bien conocidas para la producción de software de buena calidad. Otros piensan que el nombre se debe a que apareció en un momento en que los deportes extremos o de riesgo estaban de moda. Quizá no vayan desencaminados, pues hay bastantes libros y artículos en los que se muestra a programadores XP flotando en el aire, con patinetes y paracaídas, mientras teclean en el ordenador. Pese a ello, parece que tirarse desde un puente o un acantilado no es requisito imprescindible para practicarla. Menos mal.

La XP se popularizó con el lanzamiento del libro *Extreme Programming Explained: Embrace Change* (1999), donde se reflejaban las experiencias obtenidas en el proyecto C3. Según el libro, el objetivo de la XP es conseguir código ligero, conforme a las necesidades del cliente, y que pueda modificarse

con rapidez, respondiendo a las fluctuantes necesidades del cliente. Según Kent Beck:

> ¿Qué es la XP? La XP es una manera ligera, eficiente, de bajo riesgo, flexible, predecible, científica y divertida de desarrollar software. Se distingue de otras metodologías por
>
> - Su retroalimentación temprana, concreta y continua de los ciclos cortos.
> - Su aproximación de planificación incremental, que rápidamente surge con un plan de conjunto que se espera que se desarrolle a través del ciclo de vida del proyecto.
> - Su capacidad para planificar flexiblemente la implementación de las funciones, respondiendo a las necesidades cambiantes del negocio.
> - Su dependencia de pruebas automáticas escritas por los programadores y clientes para monitorizar el progreso del desarrollo, para permitir que el sistema se desarrolle, y para capturar los defectos al principio.
> - Su dependencia de la comunicación oral, las pruebas y el código fuente para comunicar la estructura y el propósito del sistema.
> - Su dependencia de un proceso de diseño evolutivo que dura tanto como dura el sistema.
> - Su dependencia de la estrecha colaboración de programadores con capacidades ordinarias.
> - Su dependencia de prácticas que funcionan con los instintos a corto plazo de los programadores y con los intereses a largo plazo del proyecto.

Y:

> La programación extrema es una disciplina de desarrollo de software que mide el desafío de entregar software excelente a tiempo por medio de llevar al extremo prácticas simples y comúnmente aceptadas.
> El desarrollo de software trata sobre construir un modelo de algo del mundo real (específicamente un modelo de la información asociada con algo, de manera que podamos automatizarlo y también mejorar cómo manejarlo). El mundo real es vasto y complejo, siempre cambiando en el nivel microscópico y el macroscópico, a velocidades igualmente rápidas y lentas. Nuestro conocimiento del mundo real también está siempre cambiando. En el momento que un proyecto de desarrollo de software se completa, normalmente es incorrecto o irrelevante. O ambas cosas.

El sitio web extremeprogramming.org proclama de manera optimista:

> En muchos entornos de software, los requisitos que cambian dinámicamente son la única constante. Por eso la XP triunfará, mientras que otras metodologías no.

La XP, al igual que el resto de los métodos ligeros, rechaza la idea de etapas cerradas en el desarrollo de software. Rompe el proceso de desarrollo en series de ciclos pequeños, cada uno de los cuales permite al cliente comprobar cómo marcha el proyecto. Estas iteraciones de tamaño reducido impiden que, al final del proyecto, el cliente se encuentre con un sistema que no atiende sus necesidades: el cliente puede –tras estudiar los resultados intermedios– modificar las prioridades iniciales; incluir, modificar o eliminar funciones del sistema; cambiar la dirección del proyecto o incluso abandonarlo. Si el cliente decide modificar o añadir características, el desarrollador lo tendrá en cuenta para la próxima iteración del sistema. La realización práctica de estas iteraciones implica el uso de contratos de alcance opcional (*optional scope contracts*), que se explicarán más adelante.

La XP presenta una característica que llama la atención de los programadores tradicionales (o no extremos): **las pruebas para comprobar el código deben escribirse antes de escribir el código**. Esta forma de proceder es exactamente contraria a la de la programación convencional: por lo general no se programa con la intención de pasar pruebas. En la XP, el propósito de una clase no es cumplir sus requisitos, sino pasar todas las pruebas que corresponden a esa clase. La XP afirma que obrar así reduce las posibilidades de que cambios en algunas porciones del código, aparentemente inofensivos, puedan provocar graves fallos en los programas completos. Existen dos tipos de pruebas: pruebas funcionales (*functional tests*) y pruebas unitarias (*unit tests*). Las primeras afectan a todas las clases del sistema; las segundas van destinadas a clases concretas.

Las pruebas unitarias suelen usarse de forma automática (usando herramientas como JUnit, desarrollado por Erich Gamma y Kent Beck), de modo que no se necesite la interacción del usuario cada vez que se modifica el sistema. Se llaman así porque implican comprobar por separado cada unidad de código para asegurarse de que funciona por sí misma, independientemente de las otras unidades del sistema. *Unidad* suele corresponder a clase (en los lenguajes OO) o a módulo (en los lenguajes estructurados). La intención que alienta estas pruebas es sencilla: si todas las clases del sistema pasan las pruebas unitarias y, aun así, el sistema presenta problemas, la causa reside en la manera en que se han juntado las clases. Aseguran, pues, que las clases de la aplicación funcionan tal y como fueron concebidas.

Según esta metodología, las pruebas unitarias deben cumplir estos requisitos:

- Estar organizadas en grupos o baterías.
- Cubrir todas las clases del sistema.
- Ejecutarse al cien por ciento.
- Ejecutarse cada vez que se hagan cambios en el código.

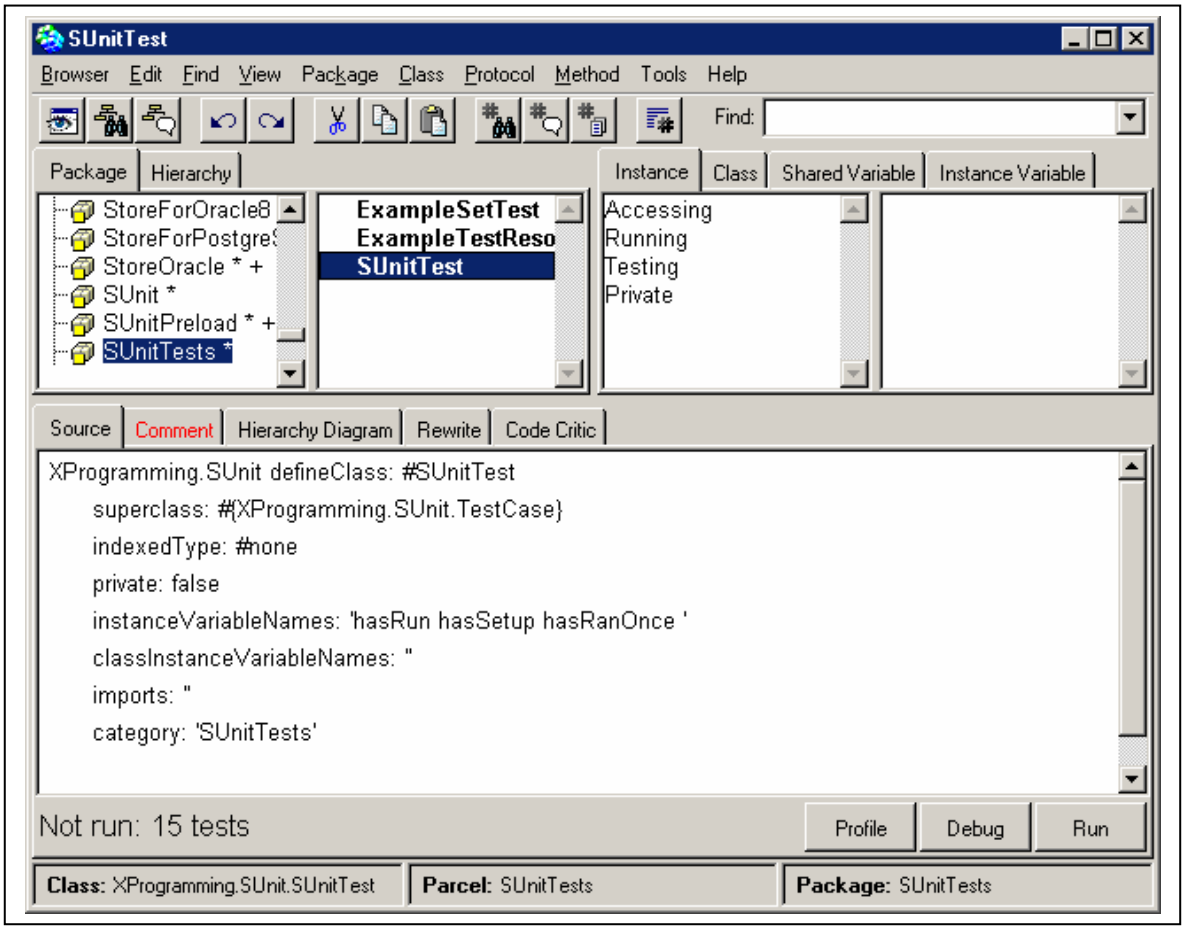

Figura 11. Captura de pantalla de la herramienta SUnit, que permite automatizar las pruebas en Smalltalk.

```
import junit.framework.TestCase;

public class TestColaEnteros extends TestCase {

	public void testColaEnteros() {
		ColaEnteros colaVacia = new ColaEnteros();
		assertEquals( "Una cola vacía debe tener cero elementos ",
			0, colaVacia.getTamanyo() );
	}

}
```

Figura 12. Ejemplo de código Java en JUnit para probar una clase que representa una cola de números enteros.

Otra característica llamativa de la XP es la **refactorización del código**, compartida por todos los métodos ágiles. Refactorizar es realizar modificaciones en el código fuente con el propósito de mejorar su estructura interna, pero sin alterar su comportamiento externo. De este modo, el programador siempre puede tener su código en un estado óptimo. Martin Fowler et al. proporcionan una definición más general en su libro *Refactoring: Improving the Design of Existing Code*:

> Refactorización (nombre): un cambio hecho a la estructura interna del software para hacerlo más fácil de entender y más barato de modificar, sin cambiar su comportamiento observable.
>
> Refactorizar (verbo): reestructurar software aplicando una serie de refactorizaciones, sin cambiar su comportamiento observable.

La preservación del comportamiento observable es imprescindible: en caso contrario, el código resultante de refactorizar no sería directamente comparable con el código previo. Para asegurar esto, suelen emplearse herramientas automáticas para la refactorización y herramientas para hacer pruebas automáticas, como JUnit (para Java), CPPUnit (para C++) o Nunit (para los lenguajes de la plataforma .Net).

El proceso general de una refactorización incluye los siguientes pasos:

- Se examina el código fuente.
- Se señalan los lugares donde conviene refactorizar.
- Se eligen los métodos de refactorización apropiados (extraer método, etc.).
- Se escriben pruebas unitarias para comprobar las funciones existentes. Si se sigue la XP, este paso ya se habría hecho antes.
- Se aplican las refactorizaciones de una en una, por medio de alguna herramienta
- En cada refactorización, se ejecutan las pruebas unitarias. Si fallan, hay que deshacer la última refactorización y ejecutarla en pasos de menor tamaño.

Por supuesto, para poder hacer estos pasos de una forma realista se precisa un entorno integrado de desarrollo (como Eclipse o NetBeans) y un sistema de control de versiones (como CVS).

El lector interesado en saber más sobre esta técnica puede consultar el tutorial *Refactoring*, de Martín Pérez, publicado en **javaHispano**. Es un tutorial práctico, claro y con muchos ejemplos. Vale la pena leerlo, se sea o no partidario de los métodos ágiles, pues la refactorización puede usarse con cualquier código orientado a objetos.

```
void printOwing() {

    printBanner();
    // Print Details
    System.out.println( “Name       " +  name);
    System.out.println( “Amount " + getOutstanding());

}
```

Figura 13. Ejemplo de código Java antes de ser refactorizado. Tomado del libro *Refactoring: Improving the Design of Existing Code*.

```
void printOwing() {
    printBanner();
    printDetails( getOutstanding() );
}

void printDetails (double outstanding) {
    System.out.println( “Name: " +  name);
    System.out.println( “Amount: " + outstanding);
}
```

Figura 14. Ejemplo de código Java después de ser refactorizado. Tomado del libro *Refactoring: Improving the Design of Existing Code*.

La XP defiende cuatro valores fundamentales:

- **Comunicación.** Para la XP es esencial mantener a todos los miembros del XP informados de lo que está sucediendo.
- **Simplicidad.** Cualquier arquitectura no debe ser más que una herramienta que cualquiera pueda entender y modificar sin necesidad de tener unos conocimientos especiales. Toda arquitectura debería comprenderse de un solo vistazo.
- **Retroalimentación (*feedback*).** La arquitectura no debería ser más estática que el sistema al que modela. Una arquitectura que permita modificaciones sencillas debería capturar estado o tiempo de alguna manera.
- **Valentía.** La forma de la arquitectura debería proclamar que “Esto es un proyecto XP”. Un programador XP debe ser lo bastante valiente como para desechar el código que no funciona y empezar de cero. También debe tener el coraje suficiente para involucrarse activamente en los proyectos.

Los doce mandamientos de la XP se detallan aquí:

1. **El juego de la planificación**. La XP exige que los clientes se involucren en los proyectos, aportando historias (*user stories*). Una historia es una breve descripción de una característica o función del sistema que se quiere construir, contada desde el punto de vista del usuario del sistema. Veamos un ejemplo: "Cuando el cliente encuentra un producto que quiere comprar, lo añade a su carrito de la compra y continúa la compra". A partir de estas historias y de la prioridad que los clientes les asignan, se construye toda la planificación del proyecto.
   La planificación se descompone en dos partes: planificación de las versiones (*release planning*) y planificación de las iteraciones (*iteration planning*).
   Durante la planificación de las versiones, el cliente expone sus necesidades contando historias. Luego, el equipo de desarrollo estima el trabajo que llevará implementar cada historia y el trabajo que el equipo puede producir por iteración (una iteración suele equivaler a tres semanas). Con esta información, el equipo informa al cliente de las estimaciones calculadas y se traza con él un plan, al cual se volverá regularmente. Es el cliente quien decide qué historias se implementarán primero y en qué orden. Tras el proceso de planificación de las versiones, el cliente y el equipo de desarrollo saben con exactitud qué historias van a ser implementadas en cada versión del sistema y las fechas de entrega de cada versión. La planificación de las versiones se usa para crear planificaciones de las iteraciones para cada iteración individual.
   En el proceso de planificación de las iteraciones se perfecciona la planificación de las versiones. Justo antes de cada iteración, los programadores se reúnen y deciden qué tareas de programación deben realizarse para implementar todas las historias correspondientes a la iteración. Cada pareja de programadores se responsabiliza de ciertas tareas.

2. **Procedimiento iterativo e incremental.** Un proyecto XP se divide en iteraciones o ciclos. La versión del sistema para cada ciclo proporciona el código para un pequeño conjunto de funciones (en cada ciclo o iteración únicamente se implementan unas cuantas historias de los usuarios). Por otro lado, un proyecto XP también es incremental: cada nuevo bloque de código se incorpora a la aplicación en cuanto está preparado.

3. **Metáfora del sistema.** Cada proyecto XP tiene una metáfora que actúa como el diseño del sistema y que proporciona un sistema común de nombres a desarrolladores y clientes. Una metáfora es una historia acerca del funcionamiento del sistema, compartida por el equipo XP. Equivale a una especie de imagen mental compacta del sistema.

4. **Tamaño del código.** Los programadores XP intentan que sus programas sean tan pequeños como sea posible. Para eliminar la duplicación de código utilizan la refactorización. A menos código, menos errores.

5. **Jornada laboral.** La XP propugna una jornada laboral máxima de cuarenta horas. Considera que sobrepasar esas horas conlleva un aumento de los errores que no compensa el trabajo que pueda realizarse en las horas extra. La XP busca ilusionar al programador, conseguir que su trabajo sea gratificante y que lleve una vida plena: metas incompatibles con jornadas excesivas.

6. **Comunicación con el cliente y entre los desarrolladores**. Un proyecto extremo necesita que haya al menos un representante de la empresa promotora con el que se pueda hablar fluidamente, de manera que pueda seguir en tiempo casi real el desarrollo del proyecto, así como proponer modificaciones. El cliente debe estar "en casa" (*on site*); es decir, debe permanecer junto al equipo XP durante todo el proyecto. En palabras de Kent Beck: "Un verdadero cliente debe sentarse con el equipo, y estar disponible para contestar preguntas, resolver disputas y establecer las prioridades a pequeña escala". La participación del cliente es imprescindible en la XP: se intenta que el cliente participe en las pruebas de software y que se atiendan lo antes posible todas las modificaciones que vaya formulando.
   Por otro lado, todos los miembros del equipo (programadores, probadores del código e instructores) trabajan juntos, en un mismo espacio. Para cumplir este mandamiento, los programadores y el cliente deben estar en un mismo sitio físico, o no muy alejados.

7. **Inmediatez.** Una aplicación XP sólo debe estar escrita para cumplir los requisitos del ahora: no debe pensarse en posibles ampliaciones futuras o en el qué pasará.

8. **Trabajo en parejas.** En la XP se trabaja en parejas. Cada pareja de programadores comparte un mismo ordenador. Uno se encarga de la escritura de código, y el otro revisa los errores, hace sugerencias, etc. Cada cierto tiempo deben intercambiarse los papeles.

9. **Integración continua.** El software se construye y se integra varias veces al día como parte del proceso de desarrollo (cada nuevo código que haya superado las pruebas se incorpora al código ya existente), de modo que todos los programadores de un equipo XP saben hacia dónde se dirige el proyecto.

10. **Pruebas.** La XP subraya la importancia de las pruebas: éstas se escriben antes que el propio código. Con la ayuda del cliente, los programadores definen las pruebas que debe pasar el software. Luego, los programadores se dedican a escribir código que pase esas pruebas. El proceso de ensayo del software es constante, y sólo acaba cuando acaba el proyecto.

11. **Modo de programar.** Dentro de un equipo extremo se exige que todos los programadores escriban código de manera similar (mismo tipo de comentarios, misma manera de nombrar las clases, las variables, los métodos, etc.), de manera que el código siempre ofrezca un aspecto similar. Obrando así, cualquier nuevo programador que entre en el

equipo podrá entender enseguida el trabajo ya hecho.

12. **Código sin propiedad.** En la XP, el código pertenece a todos. Se insiste en el espíritu de comunidad, incompatible con que los programadores piensen o crean que algunos fragmentos del código son de su propiedad. Un programador puede modificar cualquier clase, no sólo las escritas por él.

Estos doce mandamientos deben seguirse de forma completa, pues existe –según Beck– una red de vínculos y refuerzos entre ellos (véase la figura 15). El incumplimiento de alguno deriva en la mala aplicación de la XP.

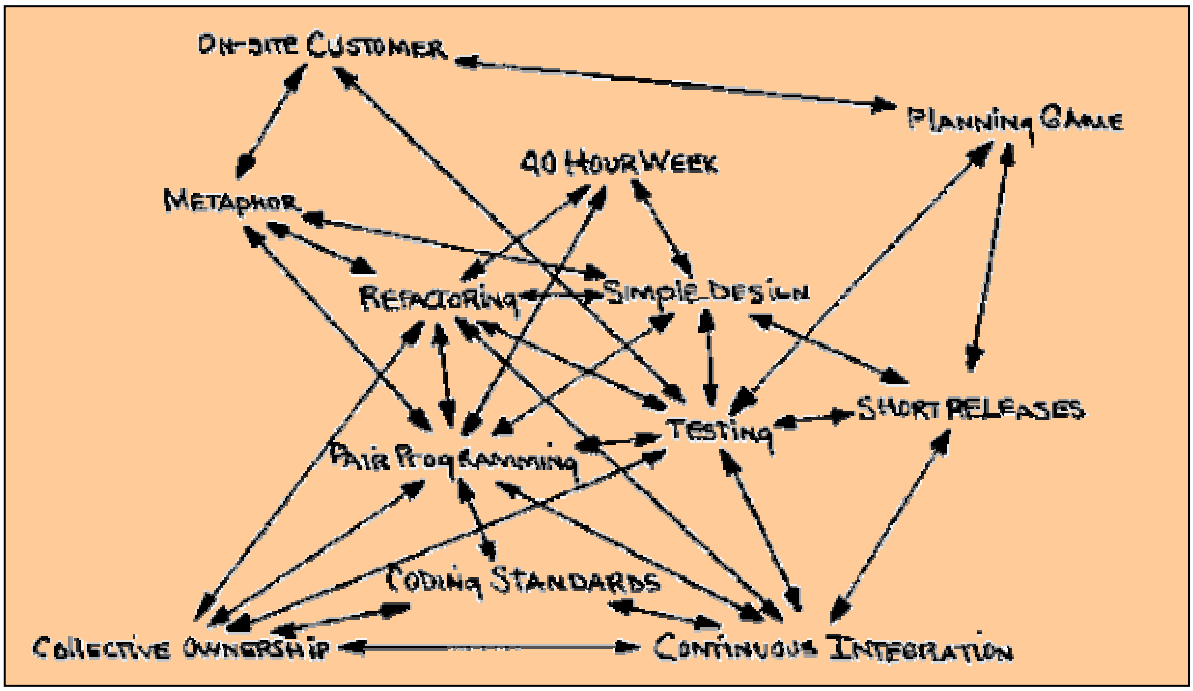

Figura 15. Interrelaciones ente los mandamientos de la programación extrema. Extraído del libro *Extreme Programming Explained: Embrace Change*.

Pese a no figurar en la lista anterior, podría hablarse de un mandamiento adicional (y optativo): **los contratos de alcance opcional** (*optional scope contracts*). La XP insiste en la importancia de que los proyectos XP se hagan con contratos de alcance opcional, por motivos que se verán a continuación. "Alcance" hace referencia a las características o funciones del sistema que es objeto del contrato.

Un ejemplo de *contrato extremo* nos lo proporciona el propio Beck (junto con algunas razones de por qué debería funcionar):

> El contrato debería parecerse a éste:
>
> > Le pagaremos al equipo de seis 75.000 dólares/mes durante los próximos dos meses. Cualquiera que sea el software que nos entreguen, reunirá los estándares de calidad de la letra pequeña de abajo. Hay algunas estimaciones iniciales en el apéndice A, pero sólo son para divertirse.

> Esto, posiblemente, no puede funcionar. El cliente no sabe qué va a tener por sus 150.000 dólares. Los proveedores podrían hacer el tonto durante dos meses y entregar entonces una pantalla de *login* de alta calidad.
> Primero, esto no va a suceder. El proveedor quiere repetir el negocio. Si el proyecto completo va a durar un año, el proveedor sólo tiene la sexta parte del dinero tras el primer contrato. En cualquier caso, hemos mantenido el contrato por poco tiempo, de manera que el cliente solamente está arriesgando el dinero de dos meses antes de que averigüe si el proceso parece o no razonablemente rápido. Las estimaciones iniciales sobre las características y un poco de matemáticas le darán al cliente una bastante buena idea de lo que podría obtener en los próximos contratos.
> Segundo, ¿qué sucede cuando la comprensión actual de los requisitos se revela errónea? Ambas partes sólo cambian la dirección. El proveedor no tiene ninguna motivación para no responder, porque desechar el código existente no afecta a su capacidad para cumplir el contrato.

En contraste, los contratos convencionales suelen ser de alcance fijo. Suelen ser fáciles de reconocer; casi siempre se piden así (más o menos): "Propónganos un sistema que satisfaga el siguiente pliego de especificaciones (ruido del mamotreto al caer sobre la mesa). Páseme un presupuesto especificando la fecha de entrega y el coste total. No se encante con el presupuesto, que lo necesito para ya, y modérese con el precio, o me buscaré a alguien más barato".

La idea de los contratos de alcance opcional viene de una experiencia personal de Kent Beck:

> En los emocionantes primeros días de la comercialización de Smalltalk, acepté un contrato para desarrollar un motor de hoja de cálculo para un banco de inversiones de Wall Street. No estaba exactamente seguro de lo que me iba a costar, pero estaba seguro de que podría hacerlo en seis semanas. Contraté a Ward Cunningham, que fue mi colaborador durante mucho tiempo, para esconderse conmigo durante dos días para tener un buen comienzo. Al final de los dos días, básicamente habíamos acabado. Empleé otros dos o tres días limpiando el código, y estuve desocupado durante cuatro semanas, entonces entregué el software una semana antes. El cliente estuvo lo bastante contento como para firmar un contrato para extensiones.
>
> Esta vez lo tenía fácil. Conseguiría otro programador para implementar los cambios, que no eran todos difíciles. Estuve en contacto con él mientras el tiempo pasaba. Parecía estar haciendo buenos progresos. Sin embargo, una semana antes de la fecha límite un ominoso silencio descendió. Entonces, dos días antes de la fecha límite, tuve una extraña llamada. "Ummm... Estoy en el hospital. Mi apéndice se rompe. Estoy tan lleno de medicinas que no puedo ni ver en línea recta. Espero que [el código] esté bien."
>
> ¡¡¡Aaaaarrrrgggghhhh!!! Pánico. Cogí el código que había recibido tan tarde, me reuní con Jolt (reservo a Jolt para los pánicos especiales), y en una sesión de programación de 48 horas de maratón conseguimos cumplir

> la fecha límite. No hace falta decir que la calidad de la segunda entrega no alcanzó la calidad de la primera ni que el cliente no quedó tan contento.

En esta autobiográfica historia de Beck se aprecian los peligros inherentes a los contratos convencionales (contratos de alcance global): si el programador acaba antes de tiempo el trabajo, ambas partes quedan satisfechas. El programador ha tenido tiempo libre o tiempo para dedicarse a otras tareas, y el cliente queda satisfecho, porque se han respetado los plazos de entrega y el presupuesto inicial. Si cuando llega la fecha límite no ha entregado su trabajo o la calidad de éste no es buena, el cliente puede perder dinero por el retraso o puede tener que incurrir en nuevos gastos para tener las funciones deseadas; el programador puede afrontar las penalizaciones o consecuencias legales derivadas de su incumplimiento, amén de que posiblemente habrá perdido para siempre un cliente.

Los contratos donde quedan fijados los plazos de entrega, el coste, los requisitos y la calidad implican riesgos para ambas partes. A veces, la implementación de una función resulta más complicada de lo esperada y se debe rebajar el tiempo dedicado a otras funciones o hacer horas extra no deseadas. Por otro lado, el cliente puede encontrarse con una aplicación que cumple los términos del contrato (requisitos, calidad y fecha de entrega), pero que no se adecua a sus verdaderas necesidades. Si el cliente se da cuenta de este hecho antes de que finalicen los planos de entrega, puede exigir modificaciones; lo cual conlleva el riesgo de que la calidad del sistema se resienta (el programador debe hacer más cosas en el mismo tiempo) o de gastarse más dinero (el programador puede decir que se ha limitado a cumplir los requisitos). Si se da cuenta después, la única solución para que el sistema se adapte a sus necesidades consiste en firmar un nuevo contrato (y gastar, por tanto, más dinero).

Con los contratos de alcance opcional, la XP intenta evitar esos problemas. Un contrato de este tipo establece el tiempo, el coste y la calidad del proyecto; a cambio, permite que el conjunto de funciones del sistema se ajuste cuando se necesite, de común acuerdo entre el cliente y el desarrollador. En apariencia, estos contratos parecen trasladar los riesgos del proyecto del desarrollador al cliente: un desarrollador tiene garantizado cobrar *X* unidades monetarias en la fecha *Z*, independientemente de las funciones que proporcione el sistema entregado. Sin embargo, la XP argumenta que el riesgo para el cliente es mínimo. Como el contrato de alcance opcional trabaja con el modelo de la XP, el cliente puede establecer un primer ciclo de desarrollo para reducir todo lo posible los riesgos que corre. Tras la primera versión, el cliente conocerá lo que va a obtener por su dinero, y si ha seleccionado o no a un desarrollador que puede satisfacer sus necesidades. Si está contento con los resultados, decidirá seguir adelante con el proyecto, lo cual conllevará más versiones (y más dinero) para el desarrollador. Imaginemos un proyecto en el que la primera versión ocupa la tercera parte del proyecto. Si el cliente no queda satisfecho con esta primera versión, puede abandonar el proyecto; lo cual conlleva que el programador dejará de percibir las dos terceras partes del dinero total del proyecto.

Este tipo de contrato posibilita que, si el cliente detecta que sus requisitos no eran completos o no eran lo que en realidad quería, tenga la oportunidad de cambiarlos en el próximo ciclo. Como el cambio de alcance no afecta a la

entrega del software o al pago del trabajo realizado por los otros ciclos, el programador no tiene ninguna objeción al cambio de requisitos. Al contrario: le supondrán nuevos ingresos.

En suma, **los contratos de alcance parcial funcionan como una suscripción**. El cliente suscribe iteración tras iteración. En cada una, sabe cuánto le costará la iteración y cuáles serán las funciones que deberá tener el sistema.

Un proyecto XP comienza con la creación de las historias de usuario (*user stories)*. Los desarrolladores se reúnen con los clientes, quienes identifican sus necesidades mediante tarjetas donde escriben historias acerca de cómo deberá funcionar el sistema (pueden usarse otros sistemas: documentos escritos, hojas de cálculo, etc.). Estos últimos son los que organizan las historias por prioridad y riesgo, y los que proponen pruebas funcionales (*functional tests*) para cada historia. Por medio de las tarjetas en las que se registran historias de usuario, los desarrolladores disponen de un medio de comunicación entre ellos y los clientes: como están escritas por los clientes, usan el lenguaje del negocio.

Una vez que los desarrolladores, con la ayuda de los clientes, han entendido las historias, proporcionan una estimación del tiempo y coste de la implementación de cada historia. Para ello, los desarrolladores utilizan su experiencia o descomponen cada historia en una secuencia de etapas más simples, y luego suman las estimaciones de las etapas. Resulta común usar algún tipo de ordenación por puntos; a cada historia se le asigna un número de puntos. Generalmente, un punto equivale a una semana teórica o ideal de trabajo.

El siguiente paso consiste en calcular la velocidad del equipo de trabajo (número de puntos por semana real de trabajo). Las semanas teóricas o ideales no tienen en cuenta el tiempo dedicado a las pruebas, reuniones, previsiones, etc. La velocidad del equipo es variable (un programador puede ponerse enfermo o marcharse del proyecto, por ejemplo), pero la XP trata de obtener una velocidad media. Además, las estimaciones de la velocidad media del proyecto mejorarán a medida que vayan pasando las semanas del proyecto. A partir de la velocidad del equipo y de la evaluación en puntos de las historias, se puede proporcionar una estimación de la duración del proyecto y un calendario de las iteraciones que serán necesarias. El período de tiempo de un ciclo o iteración de desarrollo multiplicado por la velocidad del equipo da como resultado el número total de puntos de historias que pueden completarse en la iteración (una iteración XP dura, por término medio, tres semanas).

Una vez comunicado a los clientes el número de puntos que pueden completarse en una iteración, éstos deciden qué historias incluir en el ciclo y con qué prioridad, en función de sus conocimientos del negocio. Las historias de más riesgo o dificultad son las primeras en ser abordadas en la iteración. Cuando la iteración inicial esté concluida, la primera versión, completa en cuanto a funciones y pruebas, se mostrará al cliente. Si el cliente está conforme, se comenzará otro ciclo siguiendo con las historias que no se han abordado o modificando las ya existentes. Cada iteración terminada incluirá de forma completa las historias previstas para esa iteración y todas las historias de las iteraciones anteriores (quizá modificadas).

Un ejemplo sencillo de organización de un proyecto XP:
Consideremos un proyecto de implementación de un sistema de matriculación electrónica en una universidad. Asimismo, consideremos que el equipo XP que va a abordarlo se compone de ocho programadores. Tras reunirse con los clientes, el equipo XP ha obtenido las historias de los usuarios y ha estimado el número de puntos de cada una (1 punto = 1 semana teórica):

- Entrada en el sistema (2 puntos)
- Matriculación en el sistema (6 puntos)
- Consulta de las notas del curso actual (4 puntos)
- Consulta del expediente académico (3 puntos)
- Introducción de las notas de un alumno (3 puntos)
- Consulta del estado económico de la matrícula (1 punto)
- Pago electrónico de la matrícula (5 puntos)

Supongamos que el equipo ha determinado, basándose en sus proyectos anteriores, que su velocidad es de 2 puntos/semana real. En consecuencia, los ocho programadores podrán acabar 8 x (1/2) puntos por semana real (4 puntos). En una iteración de tres semanas podrán acabar ocho puntos. O lo que es lo mismo, tres semanas de calendario equivaldrán a ocho semanas teóricas. Para la primera iteración, el cliente podrá elegir las historias que juzgue más necesarias o difíciles, pero de manera que la suma de sus puntos no supere los ocho puntos. Faltarán, pues, otros dieciséis puntos (24-8) para cubrir todas las funciones del sistema. Si el cliente necesita todas las funciones, habrá que considerar un plazo total para el proyecto de nueve semanas o tres iteraciones (24 puntos divididos por 8 puntos/iteración).
El equipo puede comprometerse, en cada iteración, a implementar tantos puntos como fueron completados en la anterior iteración. Por ejemplo, si el equipo sólo acabase 6 de los 8 puntos de la primera iteración, para la segunda iteración únicamente propondría implementar historias cuya suma de puntos sea 6 puntos (o menos).

Figura 16. Ejemplo sencillo de la planificación de un proyecto extremo.

Toda iteración, no sólo la primera, va precedida de la celebración de reuniones entre clientes y desarrolladores para planear cada iteración de tres semanas; estas reuniones suelen consumir el 10-20% del tiempo de cada etapa. En ellas, los clientes explican las historias que deberán hacerse en cada iteración. La coincidencia entre las historias de esta etapa y las recogidas en las iteraciones anteriores o en la primera reunión no tiene por qué ser exacta: el cliente puede haber modificado, añadido o suprimido historias. La principal tarea de los desarrolladores XP en esta etapa de planificación de las iteraciones consiste en asegurarse de que comprenden bien las historias.

Al principio de cada iteración, los programadores traducen las historias de los usuarios en tarjetas CRC *(Class-Responsibility-Collaboration*) y desarrollan metáforas. En la XP, una **metáfora** es una historia compartida por el equipo XP acerca del modo como funciona el sistema, una especie de imagen mental compacta del sistema. Normalmente, la historia comprende un conjunto de

clases y patrones, y da a los programadores un entendimiento común de como funcionan las cosas en cada etapa del proyecto y de como deberían funcionar. Para el proyecto C3, verbigracia, la metáfora se basaba en la manufacturación: como DaimlerChrysler fabrica automóviles, los empleados y sus salarios eran procesados como si fueran productos de una fábrica.

A partir de las tarjetas CRC y las metáforas, los programadores extraen las clases, los métodos, las variables y las responsabilidades fundamentales. La revisión de cada iteración no se hace cada tres semanas. Un par de veces a la semana intercambian opiniones sobre la marcha de su trabajo, que usan para estimar las futuras iteraciones. El cliente “en casa” siempre está disponible para aclarar detalles de los requisitos. Si éste propone alguna modificación de las funciones del sistema o alguna nueva función (lo cual implica nuevas clases o modificaciones en las relaciones entre las clases existentes), el equipo de programadores celebrará una sesión de tarjetas CRC para aclarar y fijar cómo deben implementarse las funciones nuevas o modificadas.

Antes de escribir código para implementar una función del sistema, los programadores XP definen pruebas unitarias (*unit tests*) para el código. Su objetivo es escribir el código más simple posible que pueda pasar esas pruebas. Ni una más ni una menos. Las pruebas unitarias comprueban el comportamiento completo de una clase, mientras que las pruebas funcionales comprueban el sistema completo; las primeras son responsabilidad del programador, las segundas son propuestas por el cliente.

Por regla general, una prueba unitaria es un programa que envía un mensaje a una clase y verifica que la respuesta es la predicha; suelen implementarse como aserciones. Cuando se va a incorporar nuevo código al sistema, se ejecutan todas las pruebas unitarias existentes, no sólo las asociadas al nuevo código. Si en el sistema se introduce código nuevo que pasa las pruebas definidas para él, pero hace que fallen pruebas unitarias de otras clases, se rechaza el nuevo código. Con esta forma de trabajar, el programador siempre sabe que los fallos se deben al nuevo código que acaba de incorporar, pues en la última versión funcionaban todas las pruebas.

Los programadores XP realizan frecuentes entregas o versiones (*releases*) de su código. Al final de cada iteración se realizan las pruebas funcionales delante del cliente.

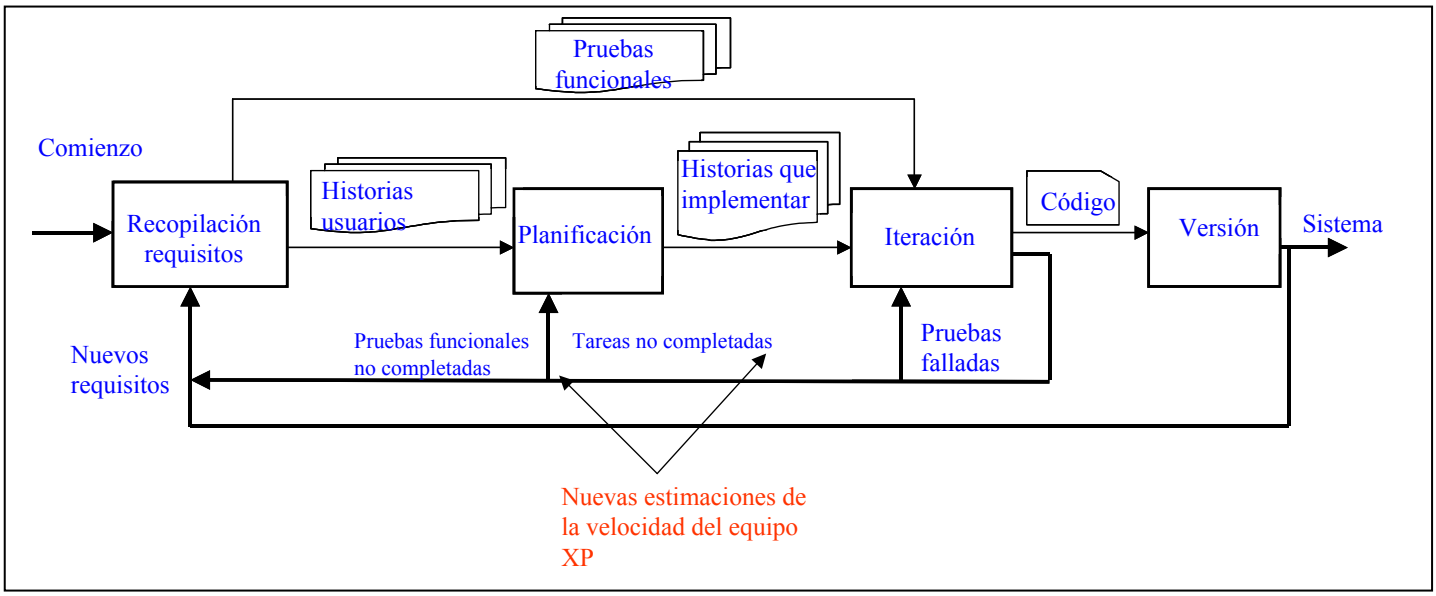

Figura 17. Esquema de un proyecto extremo.

Los papeles fundamentales en cualquier proyecto XP son los siguientes:

- **Cliente "en casa" (*on site*).** Se encarga de escribir las historias de usuario y de especificar las pruebas funcionales. Participa en las sesiones de tarjetas CRC y toma las decisiones del día a día concernientes a las funciones que debe proporcionar el sistema y a las que no. Por regla general, es un representante de la empresa con poder para decidir las prioridades del sistema.
  Al cotejar los libros originales de la XP con documentación más moderna, me he dado cuenta de que el papel del Cliente en la XP ha cambiado. Originalmente era una sola persona, correspondía al perfil de un cliente real (preferentemente, un usuario final del sistema) y permanecía durante todo el proyecto en la misma habitación o estancia que el equipo de programadores. Hace un par de años, el papel de Cliente se asignaba a un solo analista de negocios, en lugar de a un cliente real o a un usuario. En la actualidad, el papel de Cliente lo desempeñan un equipo de analistas.
- **Programador.** Se encarga de estimar el tiempo que llevará implementar las funciones del sistema asociadas a cada historia, define las tareas que conlleva cada historia e implementa las historias y las pruebas unitarias.
- **Instructor XP (*XP coach*).** Vigila todo el proceso y se asegura de que el proyecto continúa siendo extremo. Es decir, que se cumplen los doce mandamientos antes expuestos.
- **Encargado de las pruebas (*Tester*).** Se encarga de implementar y ejecutar las pruebas funcionales. También ayuda al cliente a escribirlas. De acuerdo con Kent Beck: "Un probador XP no es una persona separada, dedicada a romper el sistema y humillar a los programadores".

- **Supervisor (*Tracker*).** Se acerca al lugar donde están los programadores una o dos veces por semana, les pregunta por la marcha de sus trabajos y toma decisiones (como sugerir una sesión de tarjetas CRC o establecer una reunión con el cliente) en función de las respuestas. También se encarga de calcular la velocidad del equipo. Para ello, verifica el grado de acierto de las anteriores estimaciones. Así puede mejorar las próximas.
- **Gestor o Director (*Manager*).** Programa las reuniones y comprueba que se celebran. Acude a las reuniones e informa de los resultados a sus superiores. Por lo general, es el responsable ante la compañía solicitante del proyecto. Sus tareas no implican decir a la gente lo que debe hacer ni comprobar cómo lo están haciendo. Muchos programadores XP insisten en que la tarea más importante del director consiste en pagar las facturas de las pizzas.
- **Promotor (Gold owner).** Es la empresa o institución que solicita el proyecto, que puede coincidir o no con el Cliente.

Algunos de estos papeles pueden combinarse en una sola persona; por ejemplo, la misma persona puede tener los papeles de Gestor y de Supervisor. En un equipo XP no existen los papeles de Diseñador o de Arquitecto, ni tampoco existe el de Jefe Supremo que decide qué diseños son buenos o malos: las decisiones de diseño las toma el equipo de programadores.

Un aspecto bastante controvertido de la XP es el diseño. Como ya se ha adelantado, la XP rechaza la idea de planes completos o definitivos. La noción tradicional de usar un proceso detallado de captura de requisitos, seguida del proceso de construir un diseño detallado a partir de ésta, es rechazada por la XP, que critica a procesos tradicionales, como el de cascada, por el tiempo que transcurre hasta que las ideas de diseño se convierten en código. La opinión generalizada entre las metodologías ágiles es que el coste de la captura completa de requisitos resulta demasiado elevado. La XP rechaza la idea de un análisis completo y la de comunicar el diseño de un sistema mediante diagramas UML y especificaciones textuales. En lugar de eso, propone que el diseño se comunique mediante conversaciones, metáforas, anotaciones en las tarjetas, refactorización y código. Sobre todo, mediante el propio código.

Desde luego, la XP no propone evitar cualquier tipo de diseño. Tal y como leí una vez: “Es programación extrema, no programación estúpida”. Recomienda usar papel y UML tan poco como sea posible, de manera que el período de diseño mediante documentación escrita y UML se reduzca al mínimo. El diseño, tal y como se entiende en la XP, se realiza con las tarjetas CRC, las metáforas del sistema y el código (“El código fuente es el diseño”). Los programadores XP aprenden del código el diseño; al escribirlo y refactorizarlo, diseñan.

Una cuestión íntimamente relacionada con el diseño es la documentación. La programación extrema está diseñada para usar comunicación humana cara a cara en el lugar de documentación escrita, siempre que sea posible. La XP prefiere la comunicación a la documentación, por cuanto estima que la conversación es más rápida y efectiva que la documentación escrita. En esta

metodología se usan las historias de los usuarios y las pruebas funcionales para la documentación de requisitos. Los documentos exigidos por el cliente (manuales de usuario, etc.) se incorporan a los proyectos XP mediante historias de usuario en las cuales se piden los documentos. Para la documentación del diseño se emplean tarjetas CRC. Con respecto a la documentación del código, se recomienda clarificar el código en lugar de comentarlo. Sólo después de haber hecho todo lo posible por evitar que un bloque de código necesite ser comentado se admite comentarlo. En todo caso, si un comentario es imprescindible, se recomienda expresarlo desde el punto de vista de "cómo usarlo", no de "cómo funciona". Alistair Cockburn escribe:

> La programación extrema a la manera de un proyecto como C3 requiere cuatro cosas:
> - Que haga programación en parejas.
> - Que entregue un incremento cada tres semanas.
> - Que tenga un usuario en el equipo con dedicación exclusiva.
> - Que tenga pruebas unitarias de regresión que pasen el 100% del tiempo.
>
> Como recompensa por hacer esto:
> - No tiene que poner comentarios en el código.
> - No necesita escribir ningún requisito o documentación de diseño.

Si bien podría parecer que las historias de usuario y los casos de uso (secuencia de acciones que realiza un actor en el sistema para conseguir un objetivo) son técnicas similares, no es así. La XP rechaza el uso de casos de uso; los considera demasiado complicados y formales. Considera inútil el tiempo empleado en la redacción de casos de uso, en el uso de plantillas de casos de uso y en la notación UML de *includes*, *extends* y *uses*. Las historias de usuario son distintas de los típicos requisitos escritos, pues están enfocadas más hacia un nivel de descripción que permita estimar cuánto tiempo se necesitará para implementarlas.

La XP utiliza unos lemas que son compartidos por toda la comunidad de desarrolladores XP. Ahí van algunos de ellos, junto con su explicación:

- **Pregúntele al código (*Ask the code*).** Cuando un programador XP se encuentra con que el código no funciona como quería o no acaba de comprender su comportamiento, pone en práctica este lema. Significa que, en lugar de tratar de comprender mentalmente a qué se debe el comportamiento indeseado del código, debe colocar puntos de interrupción (*breakpoints*) en el código para averiguar qué valores tienen las variables o cuál es el flujo de control que sigue de verdad el programa (o escribir código adicional para averiguar qué esta sucediendo).
- **El código huele mal (Code smells).** Este lema es una metáfora olfativa. Decir que un código huele mal es una indicación de que algo va mal en el código. Supongo que el olfato XP puede desarrollarse con la experiencia. Hay muchos motivos para decir que un código huele mal: exceso de métodos estáticos en una clase, código duplicado,

tamaño excesivo de los métodos, clases con demasiado código o con poco código, uso de métodos con el mismo nombre pero con distinto significado, sentencias *catch* vacías, demasiada documentación, antipatrones, mal uso de la herencia...

- **Escuche al código (Listen to the code).** Si le preguntamos al código y nos contesta, lo lógico es que escuchemos sus respuestas. Al menos eso piensan los programadores XP. Lo extraño del asunto es que, cuando un programador XP experimentado escucha al código con problemas, encuentra que el código huele mal (nótese la doble metáfora, porque son metáforas, ¿verdad?). Curiosa mezcla de sentidos; recuerda a las sinestesias que provoca la mescalina: los colores se perciben como notas musicales, la música se ve, etc. A diferencia de los numerosos motivos por los cuales el código puede ser maloliente, los motivos para que el código sea "ruidoso" son pocos: exceso de comentarios monótonos, nombres demasiado largos y ruidos introducido por el lenguaje de programación empleado (sintaxis adicional como "\n", "\t", etc.)
- **No va a necesitarlo (You aren't gonna need it o YAGNI).** Este lema establece que las cosas deben implementarse cuando el programador verdaderamente las necesite, nunca cuando sólo prevea que puede llegar a necesitarlas. Un buen programador XP nunca se preocupa por las futuras funciones del código, sino por las actuales. Si añade un método a una clase y se da cuenta de que va a necesitar otros métodos, no añadirá los otros métodos si no los necesita en ese momento. Es decir, prefiere trabajar menos ahora, aun a riesgo de trabajar más en el futuro. El motivo que alegan los programadores XP para este lema es que resulta mala idea hacer más cuando uno ya tiene demasiadas cosas que hacer. Los programadores extremos viven al día.
- **No lo va a necesitar más (You don't need it anymore o YDNIA).** Esta regla se usa cuando el programador XP encuentra un trozo de código que nunca ha usado y que no parece necesario. La recomendación es borrarlo inmediatamente. Hay varios motivos para esto. A saber: evitar perder tiempo intentando comprender código que no se usa; evitar el trabajo de depuración del código inservible; evitar la refactorización del código inútil; no aumentar en vano el tiempo que tardarán en ejecutarse las pruebas unitarias y funcionales; e impedir que su refactorización impida una refactorización óptima del código útil.
- **Una vez y solamente una vez (Once and only once).** Este lema resume la idea de que un fragmento de código sólo debe existir en un lugar. La idea, por supuesto, no es nueva. En programación siempre se ha tendido a evitar el código redundante, por lo difícil que es mantenerlo. Para evitar el mal olor que produce el código duplicado, la XP cuenta con muchos desodorantes: aplicación del principio abierto/cerrado, refactorización, uso de patrones, rechazo visceral del "cortar y pegar", etc.
  Unas palabras de Beck nos aclararán (o nos confundirán más, según las inclinaciones hacia el misticismo que tenga el lector): "*El código*

quiere ser simple. Si eres sensible al olor del código, y el código duplicado es uno de los peores, y reaccionas de acuerdo con ello, tus sistemas serán más simples. Cuando comencé a trabajar de esta forma, tuve que abandonar la idea de que tenía una visión perfecta del sistema a la cual el sistema tenía que acomodarse. En lugar de eso, tuve que aceptar que yo era solamente un vehículo para que el sistema expresara su propio deseo de simplicidad. Mi visión podía cambiar de forma la dirección inicial, y mi atención a los diseñadores de código podía afectar a cuán rápidamente y cuán bien el sistema encontraba su tamaño deseado, pero el sistema me guía a mí mucho más de lo que yo estoy guiando al sistema".

- **Refactorice sin piedad (Refactor mercilessly).** Dado que el código quiere ser simple, hay que ayudarlo. Este lema nos recomienda evitar cualquier duplicación del método mediante el uso inmisericorde de la refactorización. Por ejemplo, si un programador extremo encuentra dos métodos que parecen iguales, refactorizará el código para combinarlos en uno. Si encuentra dos objetos con funciones comunes, los refactorizará hasta hacerlos uno. La amenaza de la refactorización se cernirá sobre cualquier código que no cuide su olor corporal.
- **Haga la cosa más simple que posiblemente podría funcionar (Do the simplest thing that could possibly work).** Esta regla viene a decir que, cuando se implementa una nueva función del sistema, debe hacerse de la manera más simple que uno piensa que puede funcionar (lo cual incluye sentencias *if*, *switch*, etc). Las pruebas unitarias ya nos dirán si funciona o no. Si se ejecutan correctamente, el siguiente paso consistirá en aplicar el lema "Refactorizar sin piedad" hasta que se cumpla el lema "Una vez y solamente una vez".
- **El código fuente es el diseño (The source code is the design).** Este lema ya ha sido citado. Aunque a mí me recuerda la famosa frase de Marshall McLuhan ("El medio es [el] mensaje"), algunos dicen que procede de *La guerra de las galaxias* ("¡Usa el código, Luke!"). Los desarrolladores extremos explican, por analogía con el ciclo del producto de un producto industrial, que la actividad de la programación es análoga a la fase de diseño; y que la compilación es equivalente a la fase de manufacturación. Por cierto, algunos programadores XP se quejan de que esa metáfora les compara con los obreros de las fábricas de manufacturas. Mala suerte, programadores XP elitistas, tal es el poder de las metáforas. Generalmente, se acepta que la frase expresa la idea de que la actividad de programar es diseñar.
- **La regla de oro (Golden Rule).** Los seguidores de la XP utilizan esta expresión para referirse a la conocida frase "No haga a los demás lo que no desea que le hagan a usted". Otras versiones de esta regla son: "Quien tiene el oro hace las reglas" (versión cínica) y "Hágaselo a los demás antes de que se lo hagan a usted" (versión anticipatoria). Usan "Goal donor" (donante de la meta) y "Gold owner" (dueño del oro) para referirse a las personas sobre las que habla la regla de oro. "Goal donor" designaba en el proyecto C3 a quienes tenían la responsabilidad de fijar las metas y objetivos del proyecto. Asimismo, "Gold owner" designaba al promotor del proyecto. No traduzco estos

términos en el texto porque hay un juego de palabras que se pierde al traducirlo al castellano (“gold” y “goal” se pronuncian en inglés de forma muy parecida).

- **¡Manténgalo simple, estúpido! (Keep it simple, stupid! o KISS).** Sin comentarios.

2.4. Otros métodos ágiles.

Presento a continuación un breve resumen de las características más importantes de algunos otros métodos ágiles, no tan populares como la programación extrema.

2.4.1. Desarrollo adaptable de software (Adaptative Software Development)

Se basa en las ideas de complejidad y emergencia, procedentes de la teoría de la complejidad. La teoría de sistemas complejos, pese a reconocer la existencia de situaciones impredecibles, no considera que la capacidad de no predecir implique la incapacidad de hacer progresos.

Al igual que la teoría de los sistemas complejos, esta metodología se adapta al cambio en lugar de luchar contra él. Asimismo, se basa en la adaptación continua a circunstancias cambiantes. En ella no hay un ciclo planificación-diseño-construcción del software, sino un ciclo de vida especular-colaborar-aprender.

El concepto de especulación es similar al de planificación. Existe, sin embargo, una importante diferencia: la especulación reconoce que los problemas complejos son de naturaleza incierta y que no podemos saberlo todo sobre ellos.

El concepto de colaboración en esta metodología se basa en que los sistemas complejos no se construyen directamente tal y como son, sino que son el resultado de un proceso de evolución.

El concepto de aprendizaje hace referencia a que hay que probar el conocimiento del equipo de manera continua, sin dar nada por supuesto. Esto se puede conseguir de distintas maneras: volviendo hacia atrás en el proyecto, teniendo reuniones con los clientes, etc. Lo importante es aprender en cada iteración del proyecto, no al final del mismo; de este modo, el aprendizaje sirve para mejorar el proyecto.

Un proyecto ASD se centra en los resultados (funciones del sistema para el usuario, que deben desarrollarse en cada iteración), no en las tareas.

2.4.2. Familia de métodos Crystal

Esta familia de métodos se basa en la idea de que todos los procesos son situacionales. El término “familia” obedece a que su creador (Alistair Cockburn) cree que cada tipo de proyecto requiere un tipo de metodología.

Los métodos Crystal están enfocados hacia las personas y se agrupan según el número de participantes en el proyecto y el riesgo que el cliente puede aceptar. Conociendo el nivel de riesgo y el número de participantes, se determina, mediante una tabla, el método adecuado para el proyecto en cuestión.

Cada método Crystal intenta ser, dado un nivel aceptable de riesgo y un número de personas, la metodología menos disciplinada o rígida que puede funcionar. Dicho de otro modo: fijadas esas dos variables, se selecciona el nivel de rigor mínimo (y las prácticas asociadas) que pueden asegurar el éxito del proyecto. No más, pero tampoco menos. Dependiendo de las circunstancias

del proyecto, se precisa seguir un proceso más o menos formal.

Un rasgo importante de la familia Crystal es su énfasis en la revisión del código tras cada iteración.

2.4.3. SCRUM

Mi diccionario de ingles define *scrum* como "una situación desordenada o confusa que involucra a un grupo de personas" y como una jugada de rubgy.

En esta metodología, un proyecto se divide en iteraciones de treinta días (llamadas *sprints* o carreras cortas). Antes de cada *sprint* se establecen las funciones que deberá satisfacer el sistema al final del *sprint*.

Cada día, el equipo de desarrollo se reúne con los directores del proyecto durante unos quince minutos (*scrum*, por similitud con el rugby) e informa a éstos del progreso del proyecto y de los problemas que se van encontrando.

Este método carece de toda fundamentación teórica y afirma basarse en experiencias que dan buenos resultados: es un proceso empírico de gestión del desarrollo de software.

2.4.4. Desarrollo controlado por funciones o características (*Feature Driven Development*)

Está enfocado hacia iteraciones cortas (unas dos semanas), de modo que se entreguen a menudo resultados tangibles que funcionen.

Se centra en las funciones del sistema. Para identificarlas utiliza plantillas de este tipo:

<Acción> el <resultado> <por|de|para|a> un <objeto>

Veamos algunos ejemplos de funciones: calcular el valor de una venta; determinar el número medio de artículos vendidos diariamente por un vendedor; averiguar el producto más vendido por una tienda; devolver el cambio de una venta al cliente; calcular la cantidad total vendida por un dependiente para un artículo.

Curiosamente, es la única metodología ágil que hace un fuerte hincapié en el uso de herramientas CASE. Propugna cinco procesos básicos:

- Desarrollo de un modelo de clases del sistema en conjunto.
- Formulación de una lista de las características o funciones del sistema ordenada por prioridades.
- Planificación del desarrollo
- Diseño por función.
- Construcción por función.

Los tres primeros se hacen al principio del proyecto; los dos últimos durante cada iteración del proyecto. Supongo que el desarrollo de un modelo inicial completo –casi un anatema en los métodos ágiles– se deriva de que uno sus creadores es un experto en OO (Peter Coad).

En un proyecto con el FDD hay dos clases de desarrolladores: los programadores jefe y los programadores de las clases (o dueños de éstas: el propietario de una clase es quien la ha programado). Los primeros se encargan de decidir las funciones o características que debe proporcionar el sistema. Analizan qué clases se necesitan para implementarlo y hacen que los programadores dueños de esas clases trabajen en grupo. El papel de programador jefe es de gestión: coordina, supervisa y lidera al grupo encargado de una función del sistema. Los propietarios de las clases de un grupo escriben el código necesario para implementar la función, tarea en la cual puede ayudarles su programador jefe.

3. Contra los métodos ágiles

Mi gran reproche a los métodos ágiles se resume así: lo que es original no funciona, y lo que funciona no es original. Aun así, tienen algunos puntos positivos. Si considerase que ni siquiera proponen una buena idea, no escribiría sobre ellos, al igual que no escribiría sobre La Asociación para la Tierra Plana (una entrañable y curiosa asociación estadounidense con varios cientos de miembros). En estos métodos encuentro interesantes varias ideas:

- La comunicación entre programadores, y entre clientes y desarrolladores.
- El hincapié en la refactorización.
- El hincapié en las pruebas que debe pasar el software.
- La importancia que dan a mantener estándares de escritura del código.

Ahora bien, estas ideas son usadas desde hace tiempo en la ingeniería del software. Algunas son evidentes incluso para los no programadores: ¿quién piensa que un proyecto, sea de informática o de cualquier ingeniería, terminará bien si no hay comunicación entre los clientes y los encargados del proyecto, o entre los miembros del equipo de trabajo? Son reglas que derivan del mero sentido común.

Con respecto a la idea de mantener a todos los programadores en una habitación, existen maneras más sensatas de conseguir que funcione el trabajo en equipo. Cualquier manual de Recursos Humanos enseña mejores técnicas. El lector interesado en saber cómo conseguir que un equipo de programadores trabaje satisfactoriamente en equipo puede consultar *Software Project Survival Guide* (**[McConnell, S.C., 1997]**). Libro excelente, práctico y realista. Por añadidura, jamás ha llegado a vender ni la décima parte que *Extreme Programming Explained*.

La refactorización es importante para mejorar el diseño de cualquier programa y para conseguir código más simple, de acuerdo; pero refactorizar a todas horas es innecesario cuando se dispone de diseños previos.

Ideas como la tercera y la cuarta se recomiendan y se usan desde hace tiempo: figuran en todos los textos de ingeniería del software; pero no veo mal que se haga hincapié en ellas. Sí considero que constituye grave error mezclar estas prácticas con otras sumamente inestables (mínima documentación, programación en pareja, metáfora, etc.) y propugnar que el conjunto funciona mejor que las partes.

No menciona la jornada semanal de cuarenta horas, sobre la cual no tengo una opinión definida, porque sé que quienes la predican no la siguen.

A continuación presento mis razones para estar en contra de los métodos ágiles. No están todas las que son, pero sí son todas las que están. Algunas críticas son específicas de la programación extrema. A fin de cuentas, es de lejos el método más popular.

3.1. Sobre hechos y otras menudencias.

En 1939, recién instalado en la ciudad de Nueva York, Salvador Dalí aceptó un encargo para decorar unos escaparates comerciales, pertenecientes a los grandes almacenes Bonwit-Teller. Dalí escogió como tema el Día y la Noche; el primero lo evocó mediante un maniquí metido en una bañera peluda; el segundo, mediante brasas y paños negros. La dirección de los grandes almacenes cambió el decorado sin avisar al pintor, y eso provocó la ira de Dalí, quien arrojó la bañera (llena de agua) contra los cristales del escaparate. El escándalo fue monumental, y sirvió para que Dalí fuera conocido en todos los Estados Unidos.

¿Era Dalí un destrozador de escaparates, un vándalo? ¿Pensaba pasarse toda la vida rompiendo cristales? No, claro que no. En realidad, era un hombre de negocios que sabía exactamente lo que hacía. Tenía muy claro lo que quería (fama, popularidad, dinero), y el grotesco espectáculo de la rotura del escaparate fue una exitosa manera de llamar la atención sobre su obra, de una calidad innegable.

Pensemos un poco en las metodologías ágiles: han hecho mucho ruido, han hecho muchos pronunciamientos, incluso un manifiesto. A su manera, han roto muchos cristales. Pero ¿qué hay detrás? ¿Dónde esta sus obras (los resultados)? Bien, de acuerdo, hemos oído mucho ruido, muchas promesas: ¿dónde están los maravillosos y ágiles proyectos de software que nos prometían? A lo largo de este apartado iremos contestando a estas preguntas.

El símil entre los metodólogos ágiles y Dalí no está traído por los pelos. Cada metodología ágil suele apoyarse en las ideas y la personalidad de algún líder (o de varios). Al igual que Dalí se rodeó de un núcleo de fieles acólitos (como Popper, McLuhan o Freud), Kent Beck se ha rodeado de un fiel núcleo de seguidores de sus técnicas. Entre los metodólogos ágiles y pesados es bien conocida su habilidad para atraer la atención de la gente hacia sus ideas, manteniendo su papel de líder y protagonista. Veamos la semblanza que de él hace la revista *Wired* (http://www.wired.com/wired/archive/11.09/xmen_pr.html)

> Kent Beck, quien escribió el primer libro sobre programación extrema, es un estrella de rock de la computación y un provocador. Descrito por varios colegas como "el programador más brillante que conozco" y admirado por programadores de todo el mundo, es tan abierto y apasionado como un niño. La bibliografía de *Extreme Programming Explained* [...] incluye *La Estructura de las Revoluciones Científicas*, *El Esquiador Atlético* y *Trucos Sexuales para Chicas*.
>
> Pese a todo su entusiasmo, Beck no es ningún bobo. Confiesa que quiere vender la XP y quiere venderla a base de bien. Quizás no es el dinero tras lo que va; su rancho de 20 acres en el Obregon rural donde sus cinco hijos pueden vagar –explica– es bastante riqueza. Pero Beck está librando definitivamente una batalla para los corazones y las mentes [...].
>
> "El marketing de la XP es muy deliberado y consciente", reconoce. "Parte de él está en cooperar con el poder de los medios; me aseguro de que soy de interés periodístico de vez en cuando. Parte está en cooperar algo con el presupuesto de publicidad de mi editor". Y parte, admite, es el arte de

> vender: "Si yo no hubiera tenido carisma (*charisma-ectomy*, en el original) al principio, la XP no habría ido a ninguna parte".

El que uno se rodee de seguidores fieles no es en sí mismo un problema, pero yo preferiría rodearme de críticos: **de ellos sí te puedes fiar hasta la muerte**. El problema surge cuando los discípulos tienen que ganarse la vida, y publicar, y forjarse una reputación. A menudo, los seguidores tienen que formular unas ideas más revolucionarias e innovadoras que los creadores del método. En caso contrario, ¿quién los necesitaría o los escucharía? Así es como las ideas básicas comienzan a deformarse y como los métodos se usan para situaciones para las cuales no estaban pensadas. La XP, al igual que muchas otras metodologías ágiles, se ha llenado de autores que cuentan lo que creen que cuentan personas como Beck. Y a los creadores originales les suele costar mucho descalificar a los propagadores de sus ideas deformadas. Al fin y al cabo, les están haciendo propaganda gratis. Todo esto fomenta que se haga la vista gorda con respecto a los fracasos. Después de todo, los creadores originales de cada método siempre se puedan excusar ante los fracasos con frases como "Fulanito no comprendió bien mi método" o "Menganito lo aplicó mal".

Poca gente se ha molestado en demostrar empíricamente que la programación extrema y sus compañeros de viaje no funcionan. Tampoco nadie se ha preocupado en demostrar afirmaciones como "La ingestión de hierba cura la gripe", aunque son fácilmente demostrables o falsables. ¿Por qué? Pues porque no hay a priori razones lógicas que inviten a pensar que comer hierba cura la gripe o que la programación extrema funciona.

Poniéndome en el papel de abogado del diablo, podría aceptar que los métodos ágiles funcionaran en la práctica sin que se supiera muy bien por qué, o, incluso, aunque sus principios estuvieran equivocados. Pero si no hay una base lógica para que funcionen ni tampoco tienen unos resultados superiores a los de métodos anteriores, ¿qué hacer con ellos?

La situación de falta de resultados empíricos y contrastables es tal que Pekka Abrahamsson et al., en un interesante artículo llamado *New Directions on Agile Methods: A Comparative Analysis* (**[25ª International Conference on Sofware Engineering, mayo de 2003]**), tratan de poner un poco de orden en los métodos ágiles. Las conclusiones son bastante tajantes: no hay evidencia empírica sólida e inequívoca de que los métodos ágiles sean útiles o beneficiosos; **en el mejor de los casos no se puede hablar más que de indicios de su eficacia frente a los métodos tradicionales**. La mayoría de las reglas ágiles se basa en experiencias subjetivas y en reglas prácticas, difíciles de extender a situaciones generales. Casi toda la bibliografía se basa en experiencias concretas, descritas por practicantes de las metodologías o por sus creadores.

Este artículo es un duro pescozón a los metodólogos ágiles, así como un intento de poner vallas al espacioso campo de estas metodologías. Veamos cómo acaba el artículo:

> La tendencia corriente sobre métodos ágiles se ha centrado en fabricar una pila de métodos conceptuales. En vez de apresurarse a introducir todavía más métodos ágiles, los desarrolladores de métodos deberían prestar

> particular atención a dirigirse a los problemas descritos. El campo está chillando por métodos sensatos; es decir, calidad metodológica, no cantidad metodológica.

Aun cuando acepto los resultados de Abrahamsson y de otros autores independientes, creo que han olvidado un detalle fundamental: ***el efecto Hawthorne.*** Éste fenómeno, bien conocido en psicología industrial, dice así: "**Cualquier cambio del entorno de trabajo de los trabajadores aumenta su productividad, independientemente del cambio concreto**". Me gustaría ver algún estudio en el cual se siguieran estos pasos:

1) Selección de dos grupos de programadores y de un proyecto de software.
2) Desarrollo del proyecto por el grupo 1, según una metodología ágil.
3) Desarrollo del proyecto por el grupo 2, grupo al cual se le habría asegurado antes que iba a desarrollar software siguiendo una nueva metodología, innovadora y de excelentes resultados. A estos programadores se les cambiarían las condiciones de trabajo, pero de forma aleatoria y distinta a la propugnada por la metodología ágil empleada en el punto 2. La "nueva metodología" sería, por supuesto, un montón de normas imprecisas y de recomendaciones conocidas por todo el mundo.
4) Comparación de los resultados de ambos grupos.

A mi entender, un estudio similar al anterior sí sería una prueba científica incontrovertible, o un "experimento decisivo", para dar cuenta de la eficacia de los métodos ágiles. Desde luego, habría que repetir el estudio en distintos grupos de programadores para asegurar una mínima validez estadística. Por lo que sé, nadie ha puesto en marcha un estudio similar al descrito. Aprovecho estas páginas para proponerlo.

El caso de SCRUM es un claro caso de caradura intelectual disfrazada de empirismo. Como ya se mencionó, el método SCRUM se basa en reuniones diarias de 15 minutos y en la producción de software en etapas de 30 días. Algunos lo han comparado con el rugby, pero sin las armaduras ni las animadoras.

En el artículo antes mencionado se cita a Schwaber (creador del método) y a Beedle: "SCRUM se apoya en que '*las prácticas [de SCRUM] han sido establecidas mediante miles de proyectos scrum'* (y no citan ni un solo ejemplo)". Primero: a Schwaber y Beedle les ha bailado algún cero. Segundo: es realmente grave que alguien se atreva a presentar como prueba una evidencia circular (SRUM funciona porque funcionaba). Con esta afirmación tan rotunda, estos autores vienen a decir que las prácticas del propio método se usan porque se extraen de los proyectos que siguen el método. ¿Y qué prácticas usaba el método SCRUM antes de que hubiera "miles de proyectos scrum"? Dicho de otra manera: ¿qué fue antes: el huevo o la gallina?

En la página oficial de SCRUM (www.controlchaos.com) se proclama que las características de SCRUM son, entre otras, las siguientes:

- Es un proceso ágil para gestionar y controlar el trabajo de desarrollo.
- Constituye una aproximación iterativa basada en equipos.
- Es un proceso que controla el caos de los intereses y necesidades conflictivos.
- Sirve como manera de mejorar la comunicación y aumentar la cooperación.
- Es una vía para hacer máxima la productividad.
- Es escalable desde proyectos individuales a organizaciones completas. SCRUM ha controlado y organizado desarrollos e implementaciones para muchos productos interrelacionados y proyectos con un millar de desarrolladores e implementadores.

¿Y qué pruebas se dan de todo ello? Ninguna. Ni una sola. ¿Dónde están las empresas en las que se han implantado proyectos con el método SCRUM? ¿Y los resultados? ¿Adónde han ido las estimaciones iniciales de gastos, los gastos finales y los plazos de entrega? ¿Dónde yace escondida la satisfacción del cliente?

Veamos lo que escriben John Karm y Francesco Bernardini en una revisión del artículo *The Scrum software Development Process for Small Teams* (**[IEEE Software, julio/agosto de 2000]**):

> Hay poca evidencia en el artículo que sugiera que Scrum tiene cualquier superioridad sobre los métodos tradiconales. Si el propósito de los autores era dar una visión de Scrum a las partes interesadas, han tenido éxito; pero si el propósito era convencer a la gente de la necesidad de adoptar Scrum en lugar de otros procesos existentes, entonces han fallado en ese aspecto.

Hirsch, en *Making RUP Agile* (**[Conference on Object Oriented Programming Systems Languages and Applications, 2002]**), describe dos proyectos que fueron abordados siguiendo una versión ágil del RUP (iteraciones cortas, construcción diaria del software, uso restringido de la documentación y de los artefactos). Ambos proyectos fueron conducidos por la empresa suiza Zuhlke Engineering AG. Uno de ellos, un proyecto para construir herramientas de software que diseñen láminas de turbinas de vapor, fue un rotundo éxito. El otro, un sistema de planificación de los programas de televisión, fue un completo desastre: no llego a acabarse y nunca llego a satisfacer los requisitos del cliente. Hirsch hace notar que uno de los motivos del triunfo del primer proyecto fue contar con un equipo de desarrolladores con experiencia y muy motivados. Aunque no puedo demostrarlo, estoy seguro de que ese mismo equipo hubiera construido un buen sistema usando cualquier metodología, ágil o pesada. Lo cual no dice mucho a favor de los métodos ágiles.

3.2. La importancia de los requisitos.

Frente al desprecio de los métodos ágiles hacia la recolección completa de requisitos, me permito recordar, a partir de un texto del artículo *Requirements Engineering as a Success Factor in Software Projects*, de H.F. Hofmann y Franz Lehner (**[IEEE Software, volumen 18, enero/febrero de**

2001]), la existencia e importancia de la ingeniería de requisitos. Ese mismo artículo se inicia con "***Los requisitos deficientes son la única y mayor causa de los fallos en los proyectos de software. A partir del estudio de cientos de organizaciones, Capers Jones descubrió que los RI (requisitos de ingeniería) son deficientes en más de 75 por ciento de todas las empresas. En otras palabras, tener los requisitos apropiados puede ser la única, más importante y difícil parte de un proyecto de software***".

> La ingeniería de requisitos (RI) indica tanto el proceso de especificación de requisitos estudiando las necesidades de los contratantes como el proceso de análisis y purificación sistemática de aquellas especificaciones. Una especificación, el resultado principal de RI, es un enunciado conciso de los requisitos que el software debe satisfacer; esto es, de las condiciones o capacidades que un usuario debe tener para lograr un objetivo o que un sistema posee para satisfacer un contrato o estándar. Idealmente, una especificación permite a los contratantes aprender rápidamente sobre el software y desarrollos para entender exactamente qué es lo que los contratantes quieren. A pesar de la terminología heterogénea en toda la extensión del texto, RI debe incluir cuatro actividades relacionadas pero independientes: extracción, modelado, validación y verificación. En la práctica, la mayor parte de ellos variarán en tiempo e intensidad para diferentes proyectos. Por lo general, primero deducimos los requisitos a partir de cualquier fuente que esté disponible (expertos, depositarios, o el software actual) y luego los modelamos para especificar una solución. La extracción y modelado de requisitos están vinculados. El modelado describe una solución observada en el contexto del dominio de una aplicación utilizando apuntes informales, semiformales o formales. La normalización gradual de estos modelos desde el punto de vista de los requisitos lleva a una especificación de candidatos satisfactoria, que luego deben ser validados y verificados. Esto entrega a los contratantes la información de la interpretación de sus requisitos, de manera que puedan corregir los errores lo antes posible.

3.3. Problemas comunes a los métodos ágiles.

Entre los métodos ágiles son comunes estos problemas:

- **Falta de documentación de diseño (en el sentido convencional).** Las tarjetas CRC, la metáfora, la refactorización y el código no proporcionan una documentación perdurable sin ambigüedad. El código no puede tomarse como una documentación. Primero, porque ningún lenguaje actual proporciona código legible directamente (Eiffel quizá sea el más cercano a esa meta). Y segundo, porque en sistemas de tamaño medio o grande se necesitaría leer los cientos o miles de páginas del listado de código fuente.
  Desde el punto de vista del mantenimiento y uso del sistema, proponer poca o ninguna documentación constituye una mala práctica. Sería como si un médico rechazara tomarse su tiempo para redactar el historial de un paciente.
  Las tarjetas CRC no son recomendables para utilizar en sistemas grandes (con más de 30-40 clases). Incluso en sistemas pequeños, estas tarjetas no capturan las interacciones (envío de mensajes) entre objetos. Las metáforas se tratan en el subapartado 3.12.
- **Problemas derivados de la comunicación oral.** Este tipo de comunicación resulta difícil de preservar cuando pasa el tiempo y está sujeta a muchas ambigüedades.
- **Falta de calidad.** Probar el código de forma constante no genera productos de calidad, sólo revela falta de análisis y diseño. Por otro lado, ningún método ágil cuenta con algún sistema de aseguramiento de la calidad de los sistemas. Métodos como el TSP sí miden y gestionan la calidad del producto final.
- **Fuerte dependencia de las personas.** Como se evita en lo posible la documentación y el diseño convencionales, los proyectos ágiles dependen críticamente de las personas. La substitución de los clientes que actúan como interlocutores ante los desarrolladores puede provocar cambios importantes en el desarrollo de los proyectos, dependiendo de los intereses de los nuevos interlocutores (esto no ocurriría si se hubiera consensuado antes los requisitos mínimos exigibles). El proyecto C3 fracasó, entre otros motivos, porque se marchó la persona clave que representaba los intereses de DaimlerChrysler. Por otro lado, la desaparición e incorporación de personas en el equipo de trabajo puede dirigir el proyecto a rumbos inciertos.
  La fuerte dependencia de las personas puede ocasionar problemas cuando éstas no son tan diligentes como cabría espera: un programador que deje de refactorizar o que no haga que su código pase las pruebas al cien por ciento puede poner en dificultades el proyecto.
- **Discrepancias entre las prácticas ágiles y el modelo actual de control de la calidad.** Compaginar estas prácticas con la documentación necesaria para certificar un sistema con la norma ISO 9001 resulta difícil.

- **Falta de procesos de revisión del código.** Con métodos como el PSP o el TSP se han conseguido reducciones de errores que oscilan entre el sesenta y el ochenta por ciento. La programación en parejas tiene resultados del 20-40%, que no es mucho frente al 10-25% de un programador convencional.
- **Falta de reusabilidad.** La regla YANG y la falta de documentación convencional hacen difícil que pueda reutilizarse el código ágil.
- **Sobrecostes y retrasos derivados de la refactorización continua.** Para un sistema de ciertas proporciones, los costes y retrasos derivados de la refactorización no pueden despreciarse.
- **Restricciones en cuanto a tamaño de los proyectos abordables.**
- **Olvido de ciertas características de los sistemas.** A saber: eficacia, escalabilidad, seguridad y facilidad de mantenimiento. Ninguna de estas propiedades puede conseguirse sin un diseño explícito e inicial.
- **Rigidez.** Algunos métodos ágiles son muy rígidos: deben cumplirse muchas reglas de una forma estricta para garantizar el éxito del proyecto. La XP es un buen ejemplo de método ágil que fomenta "la diversión de los programadores", pero que exige en realidad mucho esfuerzo, concentración y orden.
- **Abrazar el cambio.** Los métodos ágiles abrazan el cambio, incluso lo fomentan. Esta actitud puede ser peligrosa: abrazar el cambio a todas horas implica cambiar constantemente el código, y no siempre se puede actuar así.

  Los modelos de datos son "pesados" y no pueden cambiarse así como así sólo porque el cliente quiera incorporar más funciones al sistema. Las interfaces de los sistemas o de sus componentes tampoco pueden cambiarse alegremente, aun cuando sea para mejorarlas, en aplicaciones que exceden el alcance de una empresa u organismo.

  Responder exageradamente al cambio ha sido la fuente de muchos desastres en el desarrollo de software. El más famoso es el sobrecoste de 3 millones de dólares del *US Federal Aviation Administration's Advanced Automation System*, desarrollado para el control del tráfico aéreo en los Estados Unidos.
- **Problemas derivados del fracaso de los proyectos ágiles.** Estos problemas jamás se describen en los métodos ágiles, si bien deberían tomarse en cuenta. Si un proyecto ágil fracasa, no hay documentación o hay muy poca; lo mismo ocurre con el diseño. La comprensión del sistema se queda en las mentes de los desarrolladores. Quizá ahí esté bien cuidada y preservada para las generaciones futuras, pero dudo que eso satisfaga a los promotores del proyecto.

3.4. Limitaciones inherentes a los métodos ágiles.

Las propias prácticas de los métodos ágiles (véase el apartado 2) limitan o descartan su uso en algunos proyectos. A continuación se detallan algunos casos donde no convendría usar métodos ágiles (al menos en su formulación actual).

- **Aplicaciones distribuidas.** Las pruebas unitarias son complicadas de aplicar entre componentes. Sería necesario construir una arquitectura de pruebas para probar directamente los componentes, que podría ser tan complicada como el sistema que se desea construir. Por otro lado, el entorno de ejecución de las pruebas debería ejecutarse en el entorno (arquitecturas cliente-servidor, por ejemplo) para poder comprobar por separado los componentes. En Java, eso implicaría que el entorno de pruebas se construyera con servlets o páginas JSP.
- **Aplicaciones basadas fundamentalmente en interfaces gráficas de usuario.** No es fácil aplicar pruebas unitarias a las interfaces gráficas. Aun cuando se consiga, eso no asegura que sean cómodas para los usuarios.
- **Aplicaciones que requieren seguir un diseño estricto.** Por ejemplo: sistemas operativos, software de telecomunicaciones, etc.
- **Bibliotecas de clases.** El *carpe diem* que propugna la XP no es adecuado para construir diseñar bibliotecas de clases. Es casi imposible conseguir bibliotecas reutilizables escribiendo a trozos el código de las clases.
- **Aplicaciones que requieren una documentación exhaustiva.** Por ejemplo: sistemas militares, médicos o industriales.
- **Aplicaciones con código heredado.** Habría que reescribir todo el código heredado siguiendo los principios ágiles.
- **Proyectos muy grandes.** En estos casos, la comunicación cara a cara entre los miembros del equipo es difícil de conseguir o es imposible.
- **Proyectos escritos en lenguajes no orientados a objetos.** Lenguajes como C, Pascal, Cobol o Fortran hacen imposible técnicas como la refactorización. No hay que olvidar que las prácticas de la XP se desarrollaron con Smalltalk (véase el subapartado 3.10). Un lenguaje híbrido como C++ también plantea problemas para las metodologías ágiles: el código más simple que ejecuta una tarea suele ser de difícil lectura directa (por el uso de punteros).
- **Aplicaciones donde la escalabilidad o la eficacia sean importantes.** La escalabilidad o la eficacia no son características que se puedan añadir durante el proceso de desarrollo de software o que puedan obtenerse refactorizando: deben considerarse desde un principio.
- **Aplicaciones destinadas al mercado.** Nadie en su sano juicio sacará un producto de software comercial sin haber decidido detenidamente qué funciones espera del producto y a qué segmento del mercado se dirige.

3.5. Falta de rumbo: ¿cómo se puede saber si se ha llegado a la meta si se desconoce dónde está?

La falta de requisitos escritos detallados implica un claro desconocimiento de adónde se quiere ir. La XP afirma que las historias de los usuarios y las tarjetas CRC ya proporcionan todos los requisitos necesarios. Puede que sea así, pero la XP también afirma que se diseñó para gestionar "proyectos de alto riesgo con requisitos dinámicos". Me resulta inverosímil que un proyecto de alto riesgo se aborde con unas historias, unas tarjetas y una metáfora. Sería mucho más sensato examinar con lupa el proyecto para determinar cómo disminuir los riesgos. Así se evitaría que los desarrolladores se toparan con los riesgos cuando ya se ha escrito código.

Reconozco que la idea de abordar un proyecto de alto riesgo sin prever los peligros antes de escribir una sola línea de código me parece pintoresca, casi esperpéntica. Si yo fuera el promotor de un proyecto delicado, experimentaría una cariñosa taquicardia al oír que "el proyecto acabará cuando acabe y hará lo que tenga que hacer". No me cabe duda de que mi dinero también "se quedará donde se tenga que quedar".

La existencia de requisitos detallados me parece un buen modo de saber en qué estado está el proyecto: nos permite conocer en todo momento qué diferencias hay entre el sistema que se desea y el que se tiene por el momento. Los requisitos pueden cambiar, desde luego, durante el transcurso del proyecto. ¿Hay que abandonar por ello todo intento de registrarlos? No: claro que no. Lo que hay que intentar es documentarlos de una forma eficaz y clara (evitando tener dos mil casos de usos, usando prototipos, etc.), no renunciar a ellos. Tan extravagante me parece renunciar a la especificación detallada de los requisitos alegando que éstos cambian como renunciar al uso de los mapas alegando que las calles y plazas cambian de nombre.

Por otro lado, en un equipo XP no hay un jefe con poder decisivo para señalar el camino que conviene seguir. Los programadores deciden constantemente en qué orden y cómo programan las funciones exigidas al sistema. Después de todo, los requisitos varían día a día, momento a momento. A la falta de rumbo contribuye la movilidad de los programadores. Como un programador XP puede asumir distintas tareas, puede darse el caso de que un experto en protocolos de red se encuentre programando interfaces gráficas o conexiones a bases de datos. La XP cree que con la programación en pareja pueden limarse las diferencias de conocimientos entre individuos. Sea o no cierta esa creencia, plantea un problema interesante: un programador sin conocimientos específicos para una tarea puede ser ayudado por un compañero que sí los tenga. Resulta lógico pensar que su aprendizaje retrasará el proyecto. Y este problema se acentúa si ese programador cambia constantemente de lugar.

Veamos la crítica de Doug Rosenberg y Kendall Scott (http://www.sdmagazine.com/documents/s=730/sdm0106c/0106c.htm; se necesita registro previo) a la falta de rumbo de los proyectos XP:

> **No revisar los requisitos en absoluto.** En cambio, invite a la "funcionalitis" al permitir a los programadores construir lo que quieran. Una de las proclamas fundamentales de la programación extrema (XP) es que,

> como los requisitos cambian cada día, no tiene mucho sentido tratar de manejarlos explícitamente. Los proponentes de esta aproximación no solo pierden toda la trazabilidad de los requisitos, sino también la confianza entre clientes y desarrolladores, que puede venir únicamente de la negociación intensiva, cara a cara. Los amigos de la XP incluso tienen eslóganes llamativos para describir este fenómeno. Kent Beck usaba uno en su sitio Wiki Web para diagnosticar el fracaso del proyecto C3: "… el problema fundamental fue [que] el Gold Owner y el Goal Donor no eran el mismo. El cliente que suministraba historias no se preocupaba de las mismas cosas que los gestores que evaluaban el rendimiento del equipo… Los nuevos clientes querían pellizcos al sistema existente más de lo que querían parar el próximo sistema de nóminas. Los directores de informática querían parar el próximo sistema de nóminas." Traducción: en la jerga XP, el Goal Donor es el representante de los clientes que se sienta en la habitación con los programadores, que explica que todo está bien para cambiar los requisitos en medio del proceso, mientras que el Gold Owner es el promotor del proyecto. En el caso de C3 ([…]), el Gold Owner arrojó "inexplicablemente" la toalla en febrero de 2000, cuando, después de cuatro años de trabajo, fue evidente que el programa (sin duda lleno de bonitas características y funciones) sólo estaba pagando a un tercio de los empleados.
>
> ¿Habría salvado la revisión de los requisitos a C3? No podemos decirlo seguro, pero sabemos que la "funcionalitis", cuando se hace a expensas del calendario, es un resultado predecible de dejar al equipo que decida (y continuamente cambie) las prioridades de los requisitos sin informar a los promotores del proyecto.

El analista tradicional se encarga de reunirse con los clientes y de ayudarles a definir exactamente lo que quieren. La idea subyacente es recoger los requerimientos antes de comenzar a escribir una sola línea de código. Los programadores extremos no se encargan de tratar de orientar al cliente. Fomentan el cambio y aducen que "el cambio es barato". Estoy seguro de que el cambio no será lo bastante barato como para que ellos renuncien a cobrar sus honorarios.

Por mucho que los métodos ágiles hablan de "tiempo de Internet", la realidad demuestra que los proyectos que cambian de requisitos con demasiada rapidez suelen estar mal planteados o definidos. Fomentar el cambio cuando un proyecto está mal definido es irresponsable (aunque provechoso para las arcas de los desarrolladores): existen métodos bien establecidos para tratar proyectos con frecuentes cambios, que incluso cuentan con cálculos de los costes de los cambios y con trazabilidad de los requisitos. Fomentarlo cuando un proyecto está bien definido es absurdo: ¿acaso no está bien definido?

3.6. El manifiesto ágil

El manifiesto de las metodologías ágiles (www.agilemanifesto.org) es un desastre de manifiesto (o un manifiesto del desastre). En él sólo se explican obviedades (para cualquiera que no venga de otro planeta) y se hacen declaraciones de buenos propósitos. En realidad, parece que los autores quieren construir software de buena calidad y que los programadores lleven una existencia confortable, pero da la sensación de que no tienen muy claro cómo conseguirlo. Además, parecen desconocer que el camino al infierno está empedrado de buenas intenciones.

En los manifiestos de las vanguardias artísticas se gritaba, se farfullaba, se buscaban nuevos caminos; había una cierta alma, un cierto espíritu. Así, Tomasso Marinetti escribía (manifiesto futurista): "*El Tiempo y el Espacio murieron Ayer. Nosotros ya vivimos en lo absoluto, pues hemos creado ya la eterna velocidad omnipresente*". O "*Nosotros queremos destruir los museos, las bibliotecas, las academias de todo tipo*". Maldita sea, Marinetti pese a sus lamentables filias políticas, tenía agallas: gritaba y se retorcía, convulsionado por el espíritu de la época. Veamos, en contraposición, un principio del manifiesto ágil: "*Construya proyectos alrededor de individuos motivados. Deles el ambiente y apoyo necesario, y confíe en ellos para tener hecho el trabajo*". Ah, cuántas verdades nuevas, cuántas revelaciones sobre lo que anida en los corazones humanos, cuánta originalidad, cuánta sapiencia, cuántos años de investigación. Si éste es el espíritu de la época, qué época esta. Falta añadir que a los programadores extremos se les debería subir el sueldo cada semana y que la empresa debería despertarlos llevándoles el desayuno, calientito, a la cama. Pobres mineros y temporeros: ellos no tienen quien les escriba manifiestos.

Veamos otro principio del manifiesto ágil: "*La simplicidad [...] es esencial*". Sí, claro, y Madrid es la capital de España... La frase a secas, sin explicaciones adicionales de ningún tipo (que es tal y como está escrita en los principios tras el manifiesto), aporta cero bits de información, y para el receptor es tan redundante como el enunciado sobre Madrid.

Textos como "*Todo lo que realmente necesitaba saber sobre la programación en parejas lo aprendí en el jardín de infancia*" no ayudan a mejorar la situación. Si Laurie A. Williams y Robert R. Kessler creen de verdad que lo que escriben en su artículo va a mejorar el software, no deberían haber salido del jardín de infancia. ¿Por qué estos cantos a la cordialidad, a la buena fe y al "querámonos todos" proceden siempre de un país que fomenta y premia el egoísmo, la competencia y el individualismo, y que se construyó sobre el resquebrajado mito del vaquero solitario y rudo de "El Salvaje Oeste"?

3.7. Hacer una metodología ágil es sencillo: coja prácticas de otras metodologías o derivadas del sentido común y llévelas al extremo.

Aquí detallo un pequeño recetario para construir un método extremo (seguro que algunos puntos les suenan):

- Si la comunicación entre los desarrolladores es importante, póngalos a todos en una habitación. Nada dicta que se comunicarán más eficazmente, pero al menos sentirán un poco de calor humano (y físico).
- Si la comunicación entre el equipo de desarrollo y el cliente es importante, ponga al cliente, durante todo el proyecto, en la habitación donde están los desarrolladores. Es posible que el cliente tenga otras cosas que hacer (cosas triviales, por supuesto: ganar dinero para la empresa que le paga el salario... ese tipo de cosas), pero eso no es problema suyo.
- Si probar el código es bueno, pásese todo el tiempo probando el código. A veces no necesitaría ser probado… pero por si acaso.
- Si la simplicidad del software es una buena cualidad, use siempre para sus problemas el código más simple.
- Si el diseño del software es importante, diseñe todo el tiempo (y extienda el significado de “diseño” de manera que poner tabuladores en el código fuente sea también “diseñar”).
- Si las iteraciones cortas son beneficiosas, emplea las iteraciones más cortas que pueda.
- Si la integración del código es importante, integre a todas horas.
- Si la arquitectura de las aplicaciones es importante, dedíquese constantemente a mejorarla.
- Si la documentación convencional es tediosa, bórrela del mapa. Total, nadie la lee.
- Si el análisis previo es largo y costoso, olvídelo: hágalo sobre la marcha. Esas tareas tan complejas deshumanizan al programador, que podría estar tecleando sin tregua.
- Si refactorizar es una buena práctica, refactorice cada quince minutos y sin piedad.
- Si el código es importante, haga del código su religión: código, código y nada más. Si algún día se dedica a la arquitectura, ya sabe: ladrillo, ladrillo y nada más.
- Si cuatro ojos ven más que dos, use siempre cuatro ojos. Es posible que haya situaciones en que no se necesiten dos programadores en lugar de uno, pero qué más da. El que paga es el cliente.
- Si los contratos actuales no le satisfacen, ¡al cubo de la basura con ellos! Proponga contratos que favorezcan descaradamente a los programadores. Si el cliente se queda con un palmo de narices, culpa suya. No era lo bastante ágil.

Este recetario, claro, sería ilógico en la vida cotidiana (“Si beber una copa de vino cada día es bueno para la salud, beba veinte”, “Si pagar impuestos es malo (para su bolsillo), evítelo”, etc.). Pero en la ingeniería del software casi todo vale.

Un planteamiento más razonable, si bien menos extremo, hubiera pasado por hacerse unas cuantas preguntas: ¿por qué la documentación es tan tediosa?, ¿por qué casi nadie la lee?, ¿cómo debe redactarse la documentación para que los usuarios finales la comprendan fácilmente?, ¿qué falla en el análisis convencional?, ¿por qué no nos empeñamos en averiguar qué necesita realmente el cliente?, ¿se emplean bien los casos de uso?, ¿por qué no empleamos sólo las partes del UML que necesitamos?, ¿por qué se pierde tanto tiempo con las herramientas de diseño?, ¿por qué no hacemos reflexionar al cliente que cambia a todas horas sus requisitos?

Lo más sencillo es decir: vamos al extremo. Pero ¿es eso eficaz? Hasta ahora no lo ha sido. La XP se ha revelado como un estupendo argumento para que algunos programadores rehuyan prácticas de análisis y diseño bien establecidas. Como señalan Jerzy Nawrocki, Bartosz Walter y Adam Wojciechowski en *Toward Maturity Model for extreme Programming* (http://www.cs.put.poznan.pl/jnawrocki/publica/euromicro2001.doc): "[…] nuestra experiencia es que muchas personas encuentran la XP como una buena excusa para no usar prácticas de programación aprobadas. Ellos no siguen ni las prácticas clásicas ni las de la XP".

3.8. *Extreme Programming Explained: Embrace Change*. Si esto es explicar... Si esto cambia la manera en que se desarrolla el software…

Extreme Programming Explained: Embrace Change fue el libro con el que se hizo popular la XP. Es un libro breve, que puede leerse de un tirón y que no presenta ninguna dificultad especial.

Compararlo con el libro de patrones del *Gang of Four*, tal y como hace una de las favorables críticas de la contraportada, es simple y llanamente absurdo; es como comparar *Adelgace en siete días con la dieta de la sopa* con un tratado médico. Cuando uno acaba de leer *Design Patterns: Elements of Reusable Object-Oriented software*, piensa: "Cuántas cosas he hecho mal hasta la fecha. Menos mal que ahora mejoraré mi código con la ayuda de este libro". Cuando uno acaba de leer *Extreme Programming Explained*, se pregunta: "¿Y?". El libro adolece de imprecisión y de inconsistencia interna por cada uno de sus costados. Veamos un ejemplo: al principio del libro, Kent Beck escribe que usar el diseño orientado a objetos y el código de una manera flexible –ágil– permitirá que los requisitos cambien sin que se precise un aumento exponencial del tiempo necesario para modificar el código. Varios capítulos después, escribe que el programador debería escribir código sencillo, que cumpla lo antes posible los requisitos presentados, sin preocuparse por futuros cambios o por la flexibilidad del sistema. ¿Con qué tiene que quedarse el lector?

La somera y endeble "estructura" del libro se basa en historias breves sobre la infancia de Kent Beck y sobre compañerismo. Todos los ejemplos se refieren a un sólo proyecto: el C3; pero nunca aparecen datos concretos sobre él. Proyecto, por cierto, que fue cancelado por el cliente, DaimlerChrysler, alegando que el tiempo de desarrollo había sido excesivo (véase el subapartado 3.11). En muchos capítulos, Beck me da la sensación de que trata de decir algo importante sobre la vida, o sobre el ser o no ser, o sobre la construcción de software. Algo que se quedará grabado perennemente en la

mente del lector y que merecerá ser esculpido en la lápida del autor como recuerdo para las generaciones futuras. Nunca llega a decirlo.

El libro cuenta unas prácticas para la construcción de software, sin resultados concretos ni ninguna motivación organizadora. Hay doce prácticas, de acuerdo; pero ¿por qué son doce y no ocho o nueve? ¿Por qué tienen que aplicarse en conjunto para que funcionen? Estas prácticas se podrían haber resumido, si se quitan las historias y anécdotas de Beck y su ampulosa prosa, en un artículo de veinte hojas; no se necesitaba un libro de ciento dieciséis páginas. Algunas ideas revisten interés: **las pruebas del software**, **la comunicación**, **la simplicidad**, **el uso de estándares para el código**. Son buenas ideas, desde luego, pero eso lo sabe cualquier ingeniero del software o programador con una mínima experiencia. No es necesario revestirlas con prosa de predicador *new age* y combinarlas con historias sobre la infancia.

Como libro de psicología, *Extreme Programming Explained* podría tener un cierto valor: nos explica cuáles son las creencias de Kent Beck. Como libro de ingeniería del software, carece del más mínimo valor. **Faltan hechos, faltan datos, faltan maneras concretas sobre cómo aplicar los principios de la XP**. Lo único que sobra es el término “XP”. Aparece por todas partes, como si Beck pretendiera dignificar con él todas las líneas del libro. El efecto que se consigue es el contrario: la repetición hace que se torne un término vacío y frívolo.

Los únicos motivos que encuentro para explicar el éxito del libro son sus promesas: diversión, jornadas de cuarenta horas semanales, revolución en el mundo del software, modernidad, nuevas maneras de hacer las cosas, falta de preocupación por el futuro. Promesas que dan para hacer manifiestos y libros estupendos (y vendibles como rosquillas), pero no para hacer buenos libros de ingeniería.

Figura 18. ¿Ingeniero de software con talento para la publicidad o publicista con talento para la ingeniería del software? Kent Beck en la OOPSLA 02. Fuente: archivos fotográficos de la OOPSLA 02.

Otros textos de la XP y similares discurren por este cauce: un grupo de programadores se plantea cómo solucionar una necesidad del cliente, debaten entre sí, debaten con el cliente, no se aclaran (posiblemente porque no tienen ningún plan previo: *carpe diem*), vuelven a reunirse... Parece como si dijeran "Escribamos código y alguien nos sacará del atolladero de la falta de planificación". Y el lector, el pobre y sufrido lector, malgasta tiempo y dinero en compartir sus errores y sufrimientos. Las metodologías ágiles son –creo yo– como un grupo de niños dentro de una bañera de agua caliente: todos se ríen, intercambian bromas, se ilusionan, se lo pasan bien... pero generar código que funcione... Bueno, depende de lo que se entienda por "código que funcione". Si aceptamos una definición ágil...

Personalmente, la XP me recuerda a la inteligencia emocional. Tanto los textos fundamentales de la XP (y similares) como los de la inteligencia emocional –ahora ya van por la alquimia emocional– se han caracterizado por expresar obviedades y simplezas revestidas con una patina de novedad y actualidad (como "el saber no da la felicidad" y "no aumente sus conocimientos, busque nuevos amigos [a ser posible influyentes]" –inteligencia emocional–, o "el factor humano es importante" –XP). Ambas han generado bonitos libros, de cuidada edición y elevado precio, pero ideas nuevas, lo que se dice nuevas...

Reciclar ideas de otros no constituye delito, e incluso es preferible a escribir naderías, si bien no suele aportar mucha luz a los problemas del ahora. Uno de los motivos del escaso eco que ha encontrado la XP en el mundo académico procede de que muchas de sus técnicas ya se probaron hace tres décadas.

3.9. La curva de Boehm y la programación extrema.

El matemático Barry Boehm estableció en 1987, tras estudiar los datos de sesenta y siete proyectos de software, que encontrar y solucionar un problema de software después de que haya sido entregado al cliente cuesta cien veces más que encontrar y arreglar el problema en las etapas de diseño iniciales. En 2001, Boehm postuló que, para sistemas pequeños, el factor de proporcionalidad se halla más próximo a cinco que a cien.

En el libro de Kent Beck *Extreme Programming Explained*, aparece una gráfica que se conoce ahora como la curva de Boehm (Figura 19). En ella podemos ver cómo aumenta el coste de los cambios en una u otra etapa del desarrollo de software.

Hago notar que Beck habla del "costo del cambio", mientras que Boehm habla del "coste de solucionar errores". Si se acepta la afirmación de Beck, se pueden saltar las etapas de análisis y de diseño: lo mismo cuesta cambiar algo o arreglar algún error en esas etapas que en las de implementación, pruebas y distribución.

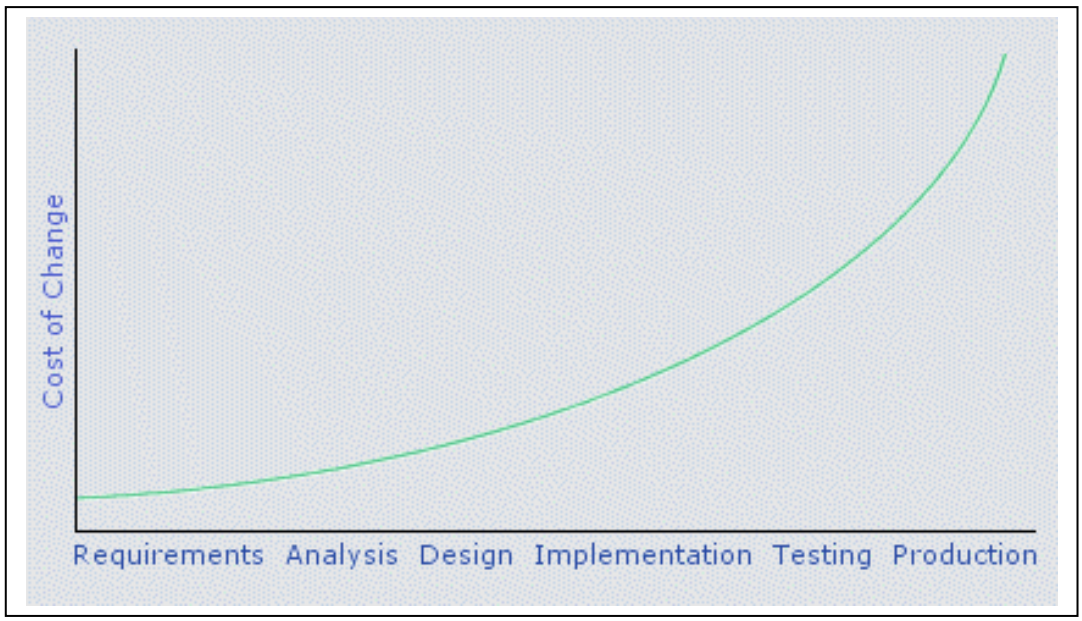

Figura 19. Evolución de los gastos derivados del cambio. Extraído del libro *Extreme Programming Explained: Embrace Change*.

Según Beck, hoy día no es válida la curva exponencial de Boehm, pues el uso de lenguajes orientados a objetos, de las modernas herramientas de desarrollo y de ciertas prácticas (usadas en la programación extrema) transforman la curva de Boehm en una curva plana (véase la figura 20). En consecuencia, con la XP puede conseguirse un coste fijo para todas las etapas del proceso de desarrollo del software. Tal y como escribe: "El cambio es barato. El coste del cambio no se eleva exponencialmente cuando el sistema crece. Contrariamente a la creencia popular, el aumento en el coste del cambio disminuye gradualmente".

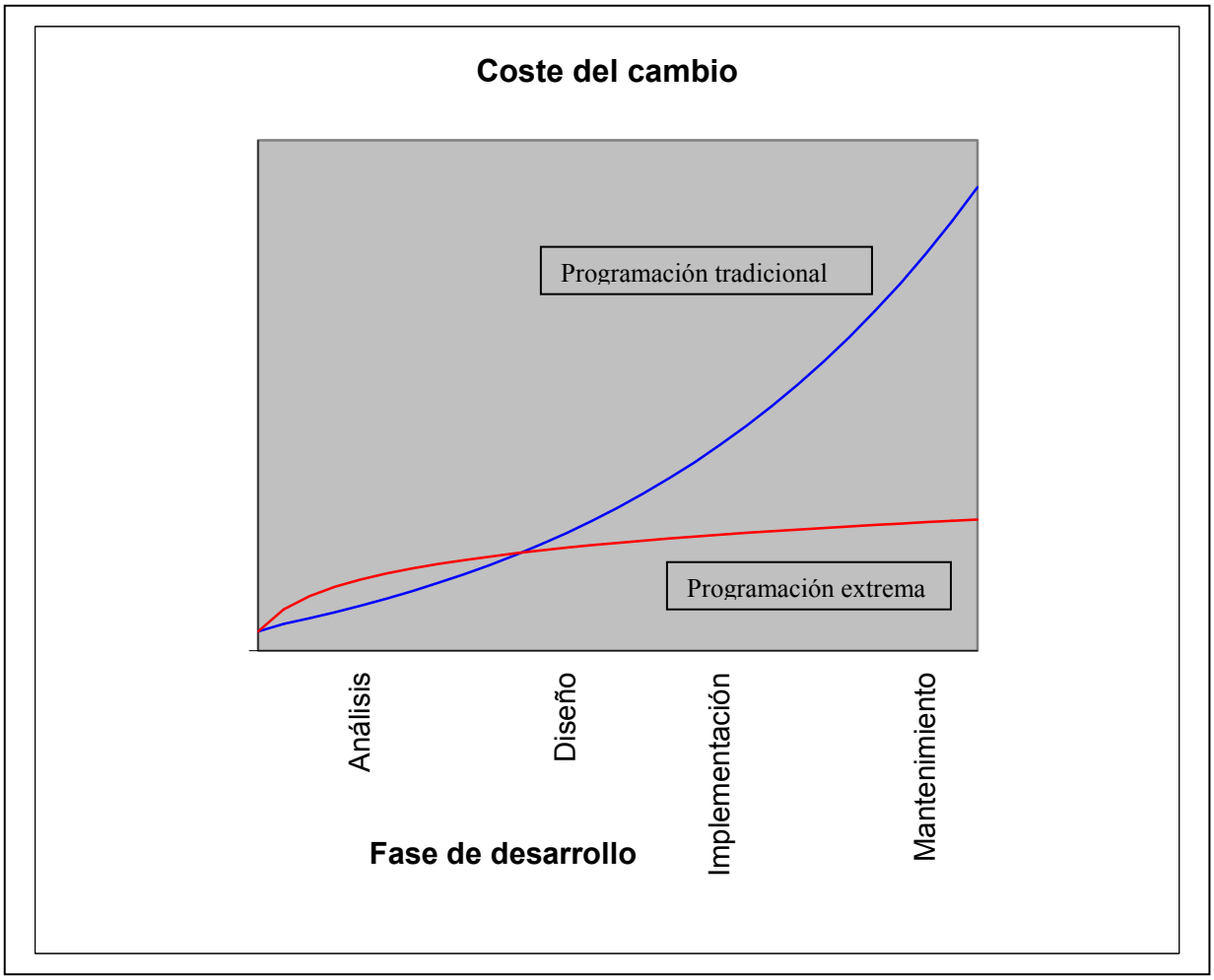

Figura 20. Evolución, según Kent Beck, de los gastos derivados del cambio en la XP y los métodos tradicionales.

Existe, sin embargo, un problema fundamental: la curva plana de Beck no está justificada por los datos empíricos actuales. Pese a todos los avances en software, la curva sigue siendo exponencial (y nada invita a pensar que acabará siendo plana). Los defensores más realistas de la XP afirman que la curva de Boehm sigue siendo exponencial, y que la XP es sólo un método para atacar los costes que aparecen en la curva de Boehm (véase http://www.agilealliance.com/articles/articles/Chapter12-VanCauwenberghe.pdf).

Reconocer eso resulta muy restrictivo XP, pues la reduce a emplearla cuando el coste de implementar las funciones del sistema no crece demasiado velozmente (o sea, para proyectos pequeños o muy pequeños). Para otro tipo de proyectos, todos los errores e inexactitudes procedentes de las historias de usuarios, etc., se arreglarán en la etapa de implementación y en la de pruebas (lo que costará mucho más tiempo y dinero que haberlos solucionado desde el principio en las etapas de análisis y diseño).

La XP argumenta que, aun cuando no pueda conseguirse una curva plana, esta metodología sí permite controlar el coste del proyecto, mediante iteraciones de corta duración y lanzamiento de versiones. Tiene razón, desde

luego, pero eso ya lo hacían las metodologías basadas en procesos iterativos e incrementales (veáse el Apdo. 2.1). Por lo que se ve, ellas no tenían tan buenos publicistas como la XP.

¿Cómo atraer el interés de la industria? Se postula que la curva de Boehm se vuelve plana con la XP. Como eso no es cierto, se dice luego que, pese a todo, permite controlar los gastos del proyecto y se recuerda que la XP tiene algunas buenas prácticas (conocidas y usadas desde hace treinta años, claro). ¿Controlará los gastos mejor que cualquier otro proceso iterativo e incremental? No hay un motivo lógico para que así sea.

Los métodos ágiles hablan mucho de "adaptarse al cambio". Olvidan, intencionadamente o por la ignorancia propia de los iluminados, que David Parnas ya enseñaba el principio del "diseño para el cambio", contradictorio con las ideas de la XP, en 1978. Sí, en 1978; para una exposición general puede consultarse *Designing software for ease of extensión and contraction* (**[D. Parnas, Proceedings of the Third International Conference on Software Engineering, 10-12 mayo 1978]**). Para los buenos artículos no pasa el tiempo: parece como si hubiera sido escrito ayer.

3.10. La programación extrema y Smalltalk.

Los creadores de la programación extrema (Cunningham y Beck) proceden de la programación en Smalltalk (lenguaje por el que reconozco sentir un cierto cariño). Consciente o inconscientemente, han llevado a la XP algunas prácticas específicas de ese lenguaje, que no son generalizables con facilidad a otros (C++, Java, lenguajes .Net).

Smalltalk es un lenguaje puro orientado a objetos. Es de tipos dinámicos, con ligadura dinámica pura. En Smalltalk, por consiguiente, no se declara que una variable es de cierto tipo. Este plantemiento presenta ciertas desventajas:

- Sólo en tiempo de ejecución se conoce si el mensaje enviado a un objeto forma parte de su protocolo. Como puede esperarse, esto conlleva en tiempo de compilación una gran cantidad de errores del tipo "no coinciden los tipos" o la necesidad de escribir código que compruebe los tipos en tiempo de ejecución o que procese las excepciones generadas.
- Todas las operaciones, incluso las más simples (sumas de números, etc.), se ligan dinámicamente. Esto redunda en un tiempo de procesamiento adicional en tiempo de ejecución. En jerarquías muy profundas de clases y con muchos métodos, el tiempo necesario para localizar el método adecuado puede ser significativo.

Sin embargo, los tipos dinámicos y el enlace dinámico puro permiten que Smalltalk sea un lenguaje muy sencillo: hay cinco palabras reservadas y la sintaxis es muy simple. Lenguajes como C++, Java y C# tienen más de cincuenta palabras reservadas, amén de contar con una sintaxis más compleja que la de Smalltalk. Se cuenta, quizás apócrifamente, que Alan Kay –creador de Smalltalk– dijo una vez: "Cuando inventé el termino *orientado a objetos* no estaba pensando en C++".

El entorno de desarrollo de Smalltalk forma una unidad con lo que es el lenguaje en sí, y permite modificar o añadir código durante la depuración, sin que sea necesario parar el programa y volver a compilar. Con los modernos entornos de desarrollo integrado, quizá esto parezca lo habitual; pero en la decada de los ochenta, e incluso cuando se desarrolló la programación extrema, el entorno de Smalltalk era muy superior al resto.

La idea de escribir pruebas unitarias antes de escribir el código que se desea probar tiene sentido en Smalltalk, pues no existe impedimento a referirse a código que aún no existe. En cambio, en lenguajes como C++, Java o C# no pueden usarse referencias a objetos que aún no existen. Para poder seguir este principio de la XP, se debe recurrir a *frameworks* específicos o a objetos *mock*. Los objetos *mock* son objetos que se usan para comprobar el comportamiento de otros objetos. Los podemos imaginar como implementaciones de una clase (o interfaz) que simulan el comportamiento real que tendría una verdadera implementación de ésta. No deben confundirse los objetos *mock* con los objetos *stub*; estos últimos sólo proporcionan la implementación de una clase mediante la implementación de su interfaz (es decir, del conjunto de métodos públicos de la clase). Un objeto *mock*, en cambio, incluye además la capacidad de comprobar cómo interaccionan sus métodos con el resto de los objetos del sistema, y permite avisar al programador de cualquier discrepancia con respecto al comportamiento esperado. Para ello permite, antes de ejecutar una prueba, cargar objetos con datos, los cuales pueden llamarse durante el transcurso de una prueba o de varias. Un uso habitual de los objetos *mock* es simular conexiones JDBC, lo cual permite realizar pruebas continuas de conexión y desconexión.

La escritura de objetos *mock*, innecesarios en Smalltalk, conlleva un coste que debe ser tenido en cuenta al abordar proyectos extremos escritos en otros lenguajes. Por otro lado, la escritura de código sencillo de leer es mucho más difícil en lenguajes del estilo de C que en Smalltalk.

3.11. El proyecto C3: un Titanic en version extrema

Cuando uno comienza a leer la bibliografía de la XP, parece que el proyecto C3 (véase el Apdo. 2.3) es como París en mayo del 68: todo el mundo estuvo allí. Ningún otro proyecto de software ha sido citado tantas veces en la bibliografía.

Apartando las mitificaciones y los olvidos voluntarios de algunas personas involucradas, podemos dar por ciertos estos hechos:

- El proyecto C3 comenzó en enero de 1995 como un proyecto de precio fijado realizado en Smalltalk (el lenguaje ideal para la XP, debido a su tipado dinámico).
- En mayo de 1996, la empresa encargada del proyecto había fallado: no consiguió entregar un sistema que funcionara. En ese momento, Kent Beck entró en el proyecto y lo volvió extremo.
- Durante las treinta semanas posteriores a la llegada de Beck, la productividad del equipo aumentó de manera significativa.
- En agosto de 1998, el C3 pagaba las nóminas a un grupo piloto de unas 10.000 personas (en pruebas probó ser capaz de pagar a otras

20.000 más).
- En febrero de 2000, el proyecto fue cancelado. DaimlerChrysler abandonó la XP como método de desarrollo.

Como vemos, el proyecto estuvo cuatro años siguiendo las prácticas extremas y bajo la batuta del creador de la XP. Resultado: un sonoro fracaso. Un escéptico se sentiría tentado a decir que no fue un comienzo muy esperanzador. Pero se trata de ingeniería del software: hasta los fracasos se pueden vender y rentabilizar. Veamos, por ejemplo, lo que apareció publicado en el artículo *Extreme measures* de la revista semanal *The Economist* (7 de diciembre de 2000, ocho meses después de que el proyecto C3 fuera desechado):

> La XP fue inventada en 1996, cuando Kent Beck, desarrollador de software, fue llamado por un fabricante americano de coches, Chrysler, para rescatar un proyecto que había demostrado ser tan frustrante que había sido desechado. Cuando el señor Beck trabajó en esta empresa sumida en la ignorancia, conocida como Chrysler Comprehensive Compensation (C3), formuló una serie de directrices para mantener el código "elegantemente escrito". El sistema C3 proporciona ahora la información correcta de las nóminas mensuales para más de 86.000 empleados.

Resulta difícil creer que *The Economist* dejara que transcurrieran ocho meses entre la redacción del artículo y su publicación. Resulta mucho más sencillo creer que Kent Beck no reflejó "exactamente" en la entrevista cuál había sido el final del proyecto (el sistema jamás pagó a más de 10.000 empleados; ni siquiera en fase de pruebas demostró ser capaz de pagar a más de 30.000 personas). Tras examinar los números de *The Economist* hasta marzo de 2001 inclusive, no he visto ninguna rectificación de la revista o de Beck al respecto (quizá la haya, pero si es así debe de estar escrita en letra muy pequeña).

Esta falsa información, publicada en una revista muy influyente en los ambientes económicos, animó a muchos empresarios a probar suerte con la XP. No hay que olvidar que el artículo también decía cosas como

> ENSEÑE a un programador alguna disciplina y programará lógica y limpiamente mientras dure el día; dele libertad absoluta y programará toda su vida a su modo idiosincrásico. Pero en el mundo rápidamente cambiante del diseño de software, nadie quiere pasar meses estudiando minuciosamente los rollos de código indescifrable cuando se necesitan reparaciones urgentes. Por eso, los informáticos tras el movimiento llamado "programación extrema" (XP) argumentan que más método igual a menos locura, sobre todo cuando los equipos de programadores afrontan *presupuestos apretados* y *fechas límites estrictas*. [La cursiva es mía]

Kent Beck y sus colaboradores no entendieron por qué se canceló el proyecto C3 (según ellos, "era un éxito" y "formaban el mejor equipo de desarrollo sobre la faz de la tierra"). Bueno, sin ser un lince, me atrevería a decir que el proyecto fue divertido para los desarrolladores, pero no para los clientes, esos seres incautos que pagaban las divertidas y caras extravagancias de los primeros.

El lector que sea poco sensible a los sueños rotos y las esperanzas frustradas puede consultar el artículo *Chrysler Goes to "Extremes"* (http://www.xprogramming.com/publications/dc9810cs.pdf), publicado dos años antes de la cancelación del proyecto. Qué pronto se torció todo.

Figura 21. Un antecesor directo del proyecto C3, también extremo. No gozó de tan buena prensa.

Las desventuras iniciales de la XP no acaban en el proyecto C3. El segundo proyecto (*Vehicle Cost and Profit System*: VCAPS) para la empresa Ford tampoco acabó bien: fue cancelado antes de finalizar. El VCAPS fue pensado como un sistema para jubilar a otro más antiguo, desarrollado con el proceso en cascada. **Resulta irónico que un proyecto hecho con una metodología "nueva y revolucionaria", repleta de supuestas virtudes, no pudiera construir un sistema que reemplazara a otro construido con un proceso tan denigrado y atacado como el de cascada**. Quizás los dioses ridiculizan a los orgullosos antes de enviarlos hacia el olvido.

Algunos creen que el proyecto "murió con las botas puestas", otros piensan que fue cancelado por motivos políticos, y algunos otros creen que "la operación fue un éxito, pero el paciente murió". Siempre se cree algo, siempre se piensa algo, siempre se dice algo. Las opiniones son libres, pero los hechos no. Se crea lo que se crea, parece que la XP no es capaz de establecer lazos duraderos con los automóviles.

3.12. La metamorfosis de la metáfora.

La idea de que una arquitectura de software venga definida principalmente por una metáfora me resulta extravagante. Pensaba atacar ese concepto; pero, al actualizar mi documentación, me he encontrado con que el propio Kent Beck tiene que dar charlas para averiguar "si las metáforas y su importancia en la programación están empezando a cobrar sentido o si debería callarme sobre ello" (cito el párrafo original completo para evitar tergiversaciones: "At the end of the talk, we will take a vote on whether metaphors and their importance to programming are beginning to make sense, or if I should just shut up about it"). Él mismo escribe: "Hay dos posibilidades: he hecho un mal trabajo explicando las metáforas y su importancia, o estoy equivocado y las metáforas no son tan importantes".

Como estoy de acuerdo con ambas posibilidades (no son excluyentes, creo yo), no tengo nada que añadir sobre las metáforas.

3.13. La programación en parejas.

La programación en parejas (o a dúo: *Pair programming*) postula la conveniencia de que los programadores trabajen en parejas, cada una en un mismo ordenador. Supongo que han aprovechado el viejo refrán de "cuatro ojos ven más que dos". Según este principio, un programador del dúo debe dedicarse a escribir código, y el otro debe revisar el código escrito, señalar los errores y dar consejos al primero. ¿Una buena idea? Depende, a la mayoría de la gente no le gusta que les observen y controlen mientras trabajan. Yo me sentiría muy incómodo si tuviera que trabajar con un auditor, pegado a mi asiento, que valorara cada frase que escribo. Que luego cambiaran los papeles no me ayudaría mucho. Se puede hablar de espíritu comunitario, de interacción social, etc., etc.; pero muchas personas son susceptibles a las críticas o los "consejos", y prefieren encontrar ellas mismas sus propios errores.

En cierto modo, la programación en parejas es una reacción a la figura del programador solitario, semejante a la del vaquero solitario. Que esta última figura sigue vigente en la sociedad norteamericana resulta evidente. Ronald Reagan es el ejemplo más inmediato que me viene a la cabeza. Ningún presidente estadounidense ha dado mejor el tipo de vaquero que Ronald Reagan, quizás porque él había interpretado antes ese tipo de papeles. Bromeó con los médicos que le atendían en urgencias tras ser tiroteado; bromeó una vez con los periodistas diciendo que acababa de ordenar bombardear la Unión Soviética... Las típicas bravatas del vaquero duro. Defendió como un vaquero arrogante a los sindicalistas y huelguistas de Polonia, pero cargó con saña contra los pilotos en huelga de su propio país. Supongo que le gustaban los sindicalistas y los sindicatos... a distancia.

Desde luego, la figura del programador solitario, dueño absoluto de su código y que almacena toda la información en su cabeza, lleva camino de la extinción. Existen, sin embargo, maneras más sencillas de fomentar el trabajo en equipo que la programación en parejas y el agrupamiento de todos los programadores en una estancia (véase la introducción al Apdo. 3).

¿Funciona la programación en parejas? Desde luego, se necesita pagar a

dos programadores en lugar de a uno. Algunos seguidores de la programación en pareja afirman que una pareja de programadores escribe código tres veces más rápidamente que un solo programador. Sin embargo, los hechos son éstos:

- Existe un estudio realizado en la Universidad de Utah (http://collaboration.csc.ncsu.edu/laurie/Papers/WilliamsUpchurch.pdf) que apunta a que los programadores en pareja escriben menos código que por separado (un 15% más de tiempo de trabajo), y que éste es ligeramente más claro que en la programación tradicional (un 15%).
- J. Nawrocki y A. Wojciechowski afirman en *Experimental Evalution of Pair Programming* (**[ESCOM 2001]**) que "[…] la programación es parejas es una tecnología bastante cara" y que parece ser menos eficiente de lo anunciado en otros estudios anteriores (entre los que se incluye el mencionado en el punto anterior).
- Matthias M. Müller y Frank Padberg alertaron en su *On the Economic Evalutation of XP Projects* (**[European Software Engineering Conference, septiembre de 2003]**) de los riesgos económicos que conlleva la programación en parejas en algunas circunstancias.

La programación en pareja puede tener su sentido en actividades de enseñanza de la programación a alumnos (véase, por ejemplo, http://collaboration.csc.ncsu.edu/laurie/Papers/XPAUPairLearning.pdf) o cuando se tratan problemas muy complicados; pero la programación en parejas aplicada a cualquier situación está tan cerca de la realidad como la estación MIR del suelo. Los programadores no suelen ser gente excesivamente sociable (nótese el *excesivamente*); si lo fueran, es probable que se hubieran dedicado a otro tipo de actividades. Hay muchas excepciones, por supuesto, pero programar en serio requiere su tiempo, tiempo que suele quitarse a actividades sociales.

En muchas organizaciones, trabajar tan agrupadamente como propugna la XP resulta inaceptable. Basta con recordar lo que sucedió cuando un gurú de la gestión empresarial llegó a una conocida multinacional y mandó arrancar las puertas de los despachos y retirar las cortinas y persianas separadoras entre despachos, con el original y campechano propósito de que "todos supieran lo que estaban haciendo todos". Los empleados más valiosos comenzaron a huir, asustados por la pérdida absoluta de intimidad, y se refugiaron en los acogedores brazos de la competencia. Resultado: el gurú provocó la mayor fuga de talentos que se recuerda en la gestión empresarial contemporánea. Su innovadora gestión fue recompensada con una deshonrosa expulsión de la empresa, harta de comprobar como perdían cuota de mercado. Auguro que este gurú empresarial tendrá un excelente futuro si orienta su trabajo hacia la industria del software.

Figura 22. Situación en la que puede degenerar la programación en parejas cuando la experiencia o el talento están mal repartidos.

Por otro lado, los proyectos de informática suelen contratarse a otras empresas, ajenas a la cultura empresarial de la empresa solicitante. A menudo, es difícil lograr que exista una interacción social positiva entre los programadores de la empresa solicitante y los de la empresa encargada del proyecto. Es más: a veces, la interacción social resulta casi imposible. Basta con imaginar el caso de un programador subcontratado, con un contrato en prácticas y un futuro precario, trabajando como programador externo en una gran empresa. ¿Será capaz de integrarse con gente que cobra el doble o triple que él y que lo mira como un "externo", como alguien que está de paso? Si usted cree que sí, es que jamás se ha encontrado en esa situación.

3.14. El código fuente no es el diseño.

El eslogan "El código es el diseño" suena un poco añejo; ya en los años setenta Marshall McLuhan escribía que "El medio es [el] mensaje". Aparte de su escasa originalidad, se trata de una aseveración falsa y engañosa.

Por una parte, diseñar y escribir código son actividades completamente distintas. El código fuente manifiesta el diseño a partir del cual fue creado, pero no es el diseño. Consideremos ahora que implementamos un determinado diseño en varios lenguajes de programación, ¿tenemos un diseño por cada lenguaje usado? No: en realidad tenemos un único diseño materializado de distintas formas. A nadie se le pasa por alto que el diseño de unos pantalones

puede aprovecharse para fabricar pantalones de distintos materiales, ¿implica eso que los pantalones de distintos materiales también son de diferente diseño? En ingeniería del software, algunos piensan que sí.

Por otro lado, una manera de facilitar la resolución de problemas consiste en subir un peldaño en la escalera de la abstracción. Si en informática no se aceptara dicha estrategia, todavía programaríamos en código máquina o en lenguaje ensamblador. El uso de diagramas UML, por caso, nos facilita la comprensión del diseño de un sistema mucho más que el código fuente. Por su naturaleza icónica, el UML está mucho más cerca de nosotros ("Una imagen vale más que mil palabras") que el código fuente. Aparte, el código de cualquier lenguaje actual de programación no tienen ni un ápice de la complejidad y riqueza de cualquier lenguaje natural (ya sea hablado o escrito). En un buen diseño se incluye una descomposición del sistema en subsistemas, así como una descripción de los servicios que cada subsistema proporciona a los restantes. ¿Qué es más fácil: ojear diagramas UML o leer páginas y páginas de código fuente? El UML tiene sus problemas, y hay muchas situaciones que no pueden reflejarse exactamente con su notación, pero el código fuente aún está mucho más limitado. Si algún lingüista ha estudiado el código fuente, habrá llegado a la conclusión de que es de baja legibilidad (como algunos textos ágiles): se necesita leer mucho para entender poco.

Algunos seguidores de los métodos ágiles dicen que "***El mapa no es el territorio***". Esta frase está tomada del conde Korzybski, creador de la pseudociencia conocida como semántica general. Para entender las circunstancias en las que germinó este lema, debemos retroceder a la I Guerra Mundial. Durante un ataque contra el ejército del Káiser, el conde Korzybski contempló la masacre en que se convirtió el ataque del escuadrón de caballería polaca que dirigía. Masacre originada porque había un foso que no aparecía en los mapas de los que disponía, y que fue aprovechado por los soldados prusianos para plantar un nido de ametralladoras con el que sorprendieron y abatieron a los polacos. Podemos entender esta verídica historia como una fatal muestra de que ninguna representación tiene la complejidad del elemento del mundo real que modela. Ahora bien, ¿debemos renunciar por ello a los mapas? Un mapa puede ser inútil porque es inexacto o porque es malinterpretado, de acuerdo. Pero ¿no es mejor un mapa que no tener nada? **Los metodólogos ágiles han renunciado a cualquier mapa**. Espero que el conde Korzybski y los curiosos No-A no se enfaden si prefiero quedarme con un lema como "El territorio es más que el mapa, pero el mapa es mejor que nada."

Como se ha visto, los métodos ágiles dan una importancia capital al código ("Código, código y código", "¡Usa el código, Luke!"), pero no muestran mucho respeto hacia él. En la XP, por ejemplo, si un fragmento de código no funciona bien se deshecha ("Hay que tener *valentía* para tirarlo a la basura").

El planteamiento de escribir código-comprobar-refactorizar lleva a mezclar errores del código y errores de diseño (en el sentido convencional del término). Una vez superados los errores del código, hay que solucionar los problemas de diseño (las clases no se relacionan entre sí como deberían; los métodos están bien escritos, pero no dan los resultados esperados; etc.).

Todo eso implica cambiar código ya escrito o tirarlo a la basura,

desperdiciándose así el trabajo invertido en escribirlo y probarlo. Hasta en un artículo favorable a la XP como *Recognizing and Responding to "Bad Smells" in Extreme Programming* (disponible en http://www.thoughtworks.com/library/Recognizing%20and%20Responding%20to%20Bad%20Smells%20in%20Extreme%20Programming.pdf) se reconoce que la regla "Haga la cosa más simple que posiblemente podría funcionar**"** debe seguirse con cierto sentido común y ciertas matizaciones para no acabar tirando a la papelera mucho código ya escrito y probado.

Pese a toda la palabrería en contra del diseño y la recopilación de requisitos de toda la vida, ha surgido un método de modelado ágil (*Agile Modeling*) para tratar de solucionar los flancos débiles, en cuanto a diseño, de los métodos ágiles. Aún es pronto para juzgar su eficacia (apareció en 2002), pero podría dar un poco de fundamento a unas metodologías que lo necesitan desesperadamente.

3.15. El código fuente no es la documentación.

El argumento más importante contra este eslogan aparece en el subapartado anterior. Por cierto, empiezo a pensar que el código fuente es como la piedra filosofal: no sólo es el diseño y la documentación; también tiene olor y unas grandes dotes de comunicación ("Pregúntele al código"). Incluso es capaz de conducir al programador por el camino hacia la simplicidad, aun cuando sea necesario tirar a rastras de él ("El sistema me guía a mí más de lo yo le guió a él", "El código quiere ser simple").

Los métodos ágiles piensan que "refactorizando sin piedad" una y otra vez se obtiene código claro para todo el mundo ("todo el mundo" también incluye neófitos y recién llegados a los proyectos). Incluso algún seguidor de la XP afirma sin complejos que "la claridad extrema [es] la propiedad de un sistema tal que el código fuente exhibe todo lo que uno necesita saber sobre el sistema, inmediatamente a mano, claro a primera vista y de forma precisa". Supongo que la claridad extrema no aparecerá en el código escrito en C++, lenguaje que tiene muchas virtudes, pero la claridad –no extrema, sino la de toda la vida– no es una de ellas. Cuanto más eficiente es el código en C++, más difícil se hace comprenderlo. Por otra parte, me resulta difícil compaginar la claridad extrema con la idea de largos listados, de cientos y miles de páginas, por muy refactorizados que estén. Imagínese el lector que tuviera que comprender la plataforma J2EE a partir de su código fuente.

Soy de la opinión de que un sistema de software es más que el código fuente. Del mismo modo que un montón de ladrillos no es un edificio, un montón de líneas escritas en un lenguaje de programación no es información. Incluso si las líneas fueran perfectamente inteligibles, entender un sistema real implicaría partir de bajo nivel: primero se entenderían líneas de código aisladas, luego métodos, luego clases completas, luego relaciones entre clases. Suponiendo que el programador tuviera paciencia (cabe recordar que el ser humano no es muy eficiente manteniendo muchas ideas en su memoria inmediata) y que sus genes le predispusieran hacia la longevidad, tardaría bastante en entender cómo se interrelacionan las partes del sistema. Afortunadamente, hay una técnica, muy antigua, para entender lo que hace un

sistema sin analizar todos y cada uno de sus constituyentes atómicos. Se llama documentación.

3.16. Refactorizar sin ton ni son es diseñar de mala manera.

No pretendo atacar la práctica de refactorizar, que me parece una estupenda idea, pero sí la de “refactorizar sin piedad”. ¿No es más sencillo diseñar desde el principio, en lugar de refactorizar a todas horas? Recordemos que tras cada refactorización hay que volver a pasar las pruebas. Se consume, pues, tiempo y dinero.

La programación extrema insiste en que hay que tener coraje para refactorizar sin piedad y para desechar el código que no funcione tras una refactorización. Pero tirar código ya escrito es un desperdicio. Precisamente, **el diseño busca descartar código antes de que sea escrito**. Puedo entender que se usen las técnicas de refactorización para código que haya sido escrito sin ningún plan previo en la cabeza o por programadores con poca experiencia; pero no que se usen como substituto completo del diseño. La evolución natural ha generado órganos tan perfectos como los ojos sin un diseño previo; pero ha contado con millones de años de pruebas y errores. No creo que la programación extrema disponga de tanto tiempo para hacer sistemas perfectos.

Como a los programadores extremos les gusta verse como artesanos o artistas, pensemos en el David de Miguel Ángel. ¿Alguien se puede imaginar al artista trabajando al azar, arrancando trozos de mármol de aquí y allá, para modelar la figura humana? Algún programador extremo podría alegar, recurriendo a las metáforas, que escribir código es como tallar una figura: se va quitando código o mármol hasta que sólo queda lo que uno quiere (el programa o la estatua). La idea es atractiva, pero ingenua: muy pocas personas son capaces de lograr trabajos tan admirables como el David de Miguel Ángel sin una cuidadosa planificación (y a veces ni aun así). El resto nos vemos forzados a planificar de antemano lo que vamos a hacer.

En mi opinión, diseñar sólo con la refactorización es un proceso ineficiente. El uso correcto del polimorfismo y de la herencia implica un diseño previo. Aparte, la refactorización como diseño deja de lado características de los sistemas no expresables en lenguajes de programación (escalabilidad, velocidad, eficiencia, etc.).

Una de las múltiples ventajas que otorga partir de un diseño previo es que se sabe cuando un módulo, paquete o clase está acabado, y no se necesita perder más tiempo en refactorizar y ejecutar pruebas.

3.17. Las pruebas.

Desde luego, no pretendo atacar las pruebas. Para mí, **el aspecto más positivo de las metodologías ágiles reside en su insistencia en las pruebas del software**. Es su característica más realista. Por otro lado, ¿alguien se cree que escribiendo código a destajo y sin pruebas, sobre la marcha, puede salir un producto final que funcione sin respiración asistida? Sería como construir edificios a base de apilar ladrillos, sin orden ni concierto, hasta que la estructura se mantenga en pie. Un montón de líneas de código NO es un sistema de software. En el fondo, las metodologías ágiles usan las pruebas y la refactorización para poner un poco de orden y de razón en el código, escrito a golpes de impulsos, de intuiciones y –a veces– de errores.

Sin embargo, opino que centrarse sólo en las pruebas, ya sean unitarias o funcionales, no resulta buena idea. Una clase puede pasar todas las pruebas unitarias que se quiera; y, sin embargo, puede estar mal diseñada. Nada nos dicen las pruebas unitarias sobre los fallos de diseño, los cuales se pueden traducir en un mal rendimiento del sistema o en que sea difícil de mantener.

Consideremos, por ejemplo, una jerarquía de clases. Las clases pueden funcionar correctamente y pasar todas las pruebas que se han pensando para ellas. No obstante, la jerarquía puede estar mal diseñada. Puede ser que haya métodos que están a una altura incorrecta de la jerarquía o que hagan demasiadas llamadas innecesarias a métodos de otras clases de las jerarquías. En estos casos, refactorizar puede no ser la solución para los problemas.

Suponer que un sistema funciona bien porque pasa todas las pruebas es ingenuo: las pruebas pueden tener errores. Si se dispone de una especificación previa –y escrita– de los requisitos, se puede saber qué resultados reales se esperan del sistema y juzgar su comportamiento. Sin una especificación exacta y consensuada del sistema, el programador siempre puede alegar que el sistema funciona bien según sus pruebas; y el programador siempre puede decir, resignadamente, que el sistema no cumple sus expectativas. No escribir los requisitos se traduce, con el tiempo, en que cada uno los recuerda a su manera.

3.18. El papel del cliente “en casa”.

La idea del cliente “en casa”, en el sentido definido originalmente por la XP, plantea bastantes problemas prácticos. Por un lado, la empresa promotora debe prescindir de un trabajador durante el tiempo del proyecto (que pueden ser meses o años). Eso exige que no se elija para ese puesto a una persona con mucha experiencia o buena conocedora del negocio (directivos, socios, etc.), porque las personas con esas cualidades son necesarias en la empresa. Me resulta impensable, por ejemplo, que un responsable de calidad o un ejecutivo de ventas desaparezca durante varios meses en una habitación llena de programadores.

La lógica invita a pensar que la empresa promotora elegirá como cliente “en casa” a una persona que no lleve mucho tiempo en la empresa o cuyos conocimientos y experiencia no sean cruciales para el día a día de la empresa.

Una persona así no conocerá bien los entresijos del negocio ni las verdaderas necesidades de la empresa. Las empresas no se construyen mediante modelos lógicos o razonables; su objetivo es ganar dinero, no obedecer a ninguna lógica interna. Un proyecto XP necesita, por tanto, a alguien que sepa bien cómo se gana el dinero en la empresa, cómo se funciona por encima de las buenas palabras, cuáles son las verdaderas necesidades del sistema informático. Y es improbable que una empresa deje marchar a personas así para que estén en una habitación llena de programadores. Los programadores extremos pueden considerarse muy importantes, pero para una empresa no informática siempre es más importante la gente que genera dinero que los programadores. Este hecho puede gustar o no, pero así son las empresas.

Por otro lado, durante un proyecto XP, los programadores y el instructor XP guían al cliente entre las complejidades técnicas del sistema e incluso le invitan a programar sus propias pruebas. Este paternalista enfoque puede ocasionar que el cliente "en casa" se deje guiar por el equipo XP y que olvide las auténticas prioridades de la compañía que le paga el sueldo. Si sucede esto último, quizá la XP gane un converso; pero el promotor del proyecto maldecirá el día en que oyó por primera vez la palabra "eXtremo".

3.19. Los contratos de alcance parcial.

Me resulta muy difícil creer que una persona adulta haya propuesto los contratos de alcance parcial. Pongámonos en la situación de un responsable de informática de una empresa que firma un contrato de alcance parcial cuya primera iteración durará tres semanas y costará 150.000 €uros. Pasadas las tres semanas, puede encontrarse con una de estas opciones:

a) El sistema resultante de la primera iteración corresponde bastante bien a sus necesidades.

b) El sistema corresponde en parte a sus necesidades.

c) El sistema no guarda relación con lo solicitado. Es el caso de la "pantalla de *login* de alta calidad " propuesta por Beck (páginas 28 y 29)

En el caso c), la empresa habrá perdido 150.000 €uros, más los gastos que les ocasione no tener el sistema aún en marcha. ¿Qué ocurrirá cuando el responsable de informática deba rendir cuentas a sus superiores? ¿Les mostrará una pérdida de miles de euros y un programa inútil, para luego musitar "parecía una buena idea: me aseguraron que eran eXtremos de verdad"? La empresa ni siquiera podrá reclamar legalmente por incumplimiento de contrato, pues han firmado un contrato aceptando lo que salga al final de la primera iteración.

¡Y la única defensa del cliente –según Beck– es que la situación c) no se dará porque el desarrollador "quiere repetir el negocio"! Bendita ingenuidad de alma cándida. Como sabe todo el mundo, hay empresas –e individuos– que pagan con cheques sin fondos, y no parece que les detenga el argumento de "repetir el negocio" (¡pero si ni siquiera les detiene la amenaza de la ley!). Cualquiera que no se haya pasado toda su vida recluido en un monasterio o en una caja de Skinner sabe que ha habido, hay y habrá empresas e individuos que siguen al pie de la letra el título de una conocida película de Woody Allen:

Coge el dinero y corre.

Que los contratos de alcance parcial puedan lugar a abusos por partes de los programadores no impide que el mercado los acepte. Siempre queda sitio para prácticas que prometan grandes beneficios y rentabilidades, por peligrosas y arriesgadas que parezcan. Basta con recordar al hombre desconocido que fundó, en la época de la burbuja especulativa de la Compañía de los Mares del Sur, una compañía "para asumir y llevar a cabo empresas muy ventajosas, pero que nadie debía conocer". Miles de inversores depositaron a ciegas sus ahorros a cambio de acciones de la empresa. El fundador –al parecer, no muy avaricioso– cogió el dinero y se desvaneció para siempre, entrando así en los anales del timo de guante blanco. Supongo que no le interesó "repetir el negocio".

3.20. El poder de la palabra.

Posiblemente, el mayor éxito de los métodos ágiles sea la elección del nombre. La palabra *ágil* suena a dinámico, sencillito, flexible, rápido. Resulta atractiva para los desarrolladores. Es como un bombón: a los programadores les gusta el sabor que deja el chocolate mientras se derrite en sus bocas, hasta que notan la cucaracha en el núcleo del dulce. Y no digamos ***eXtreme Programming***: eso sí que es un nombre comercial.

Con la XP, usted no se sienta delante de un terminal y escribe código. No, usted realiza prácticas extremas, arriesgadas, y navega en la cresta de la tecnología (aun cuando realmente se limite a compilar "Hola, mundo" en C++).

3.21. No al *carpe diem*.

El cuarto mandamiento de la XP (inmediatez) es absurdo si se toma al pie de la letra. Un ejemplo muy sencillo nos lo mostrará. Supongamos que estamos escribiendo una aplicación en C++ donde se necesita transformar libras en kilogramos y poder sumar libras y kilogramos. Siguiendo al pie de la letra el principio de inmediatez, un programador XP (o dos, mejor dicho) crearía una clase *Masa* y sobrecargaría el operador +, de modo que funcionara este código:

```
Masa m1 m2 m3;
m1= new Masa(50.0, "Kg");
m2=new Masa(32.6, "lb");
m3=m1+m2;
```

Nuestro programador XP, ayudado por su pareja, comprobaría que su código pasa las pruebas que había diseñado antes y seguiría programando otras partes de la aplicación.

El código anterior, pese a su trivialidad, implica consecuencias graves para el programa. Por ejemplo, este código no funcionaría:

```
m3= m3*7.0;
m3+=m2;
```

A fin de cuentas, si el programador XP ha seguido el mandamiento de inmediatez no tiene por qué haberse preocupado de redefinir el operador += o de definir el producto de un número de coma flotante por un objeto de la clase *Masa*. Sin embargo, cualquier programador que viera que las últimas líneas no funcionan (aunque sea en una iteración inicial) comenzaría a dudar de la competencia de nuestro imaginario programador XP. Moraleja: muchas veces las funciones de una clase o de varias están vinculadas entre sí, y no deben implementarse a trozos, o siguiendo los impulsos del cliente.

Si en un ejemplo tan elemental como el anterior cualquier programador tradicional (es decir, sensato) notaría que algo va mal, el lector puede imaginarse lo que puede suceder cuando se aborde la programación de bibliotecas de clases.

Los problemas no dependen del lenguaje elegido. Si queremos ver cómo

van aumentando en número y tamaño, podemos considerar Java. Un programador XP escribiría dos clases así para cumplir con los requisitos de la página anterior.

```
public class ErrorConversion extends Exception{

    public ErrorConversion(String mensaje){
        super(mensaje);
    }

}

public class MasaXP {

    //Una clase similar a ésta
    //sería escrita por un programador XP, por una pareja de programadores
    //XP, mejor dicho, que siguiera al pie de la letra los mandamientos de
    //la XP

  private double masaKg;

  public MasaXP (double masa, String unidad) throws ErrorConversion {
    if (unidad.equals("Kg")){
      masaKg=masa;
    }
    else if (unidad.equals("lb")) {
      masaKg=0.4536 * masa;
    }
    else throw new ErrorConversion("Unidad introducida no válida");
    }

    public void mostrarMasaKg() {
        System.out.println("Masa en Kg " + masaKg + " Kilogramos");
    }

    public double getMasa(){
        return masaKg;
    }

    public static MasaXP sumar(MasaXP m1, MasaXP m2) throws ErrorConversion {
        return new MasaXP(m1.getMasa()+m2.getMasa(), "Kg");
    }

}
```

Supongamos que el programador descubre ahora que necesita transformar TeV/c^2 a kilogramos y sumar TeV/c^2 a kilogramos. Como uno de sus principios es la inmediatez, cambiará el código anterior de manera que incluya la existencia de TeV/c^2 y el factor de conversión correspondiente. Si se necesita que el programa maneje más unidades de masa, será necesario

continuar introduciendo más modificaciones.

Veamos cómo afrontaría el problema un programador no XP un poco experimentado:

```
public class Masa {

  //Una clase similar a ésta
  //sería escrita por cualquier programador con un poco de experiencia

  private double masa;
  private String nombreUnidad="Kilogramos";
  private String abrevUnidad="Kg";

  //Constructor 1
  public Masa(double masa) {
    this.masa=masa;
  }

  //Constructor 2
  public Masa(double masa, String nombreUnidad, String abrevUnidad) {
    this.masa=masa;
    this.nombreUnidad=nombreUnidad;
    this.abrevUnidad=abrevUnidad;
  }

  public void mostrarMasa() {
    System.out.println("Masa en " + abrevUnidad + ": " + masa + " " + nombreUnidad);
  }

  public double getMasa(){
    return masa;
  }

  public String getNombreUnidad(){
    return nombreUnidad;
  }

  public String getAbrevUnidad(){
    return abrevUnidad;
  }
    // Devuelve un objeto Masa que tiene las unidades del primer
    // argumento
  public static Masa sumar(Masa m1, Masa m2, double factorConv) {
    return new Masa(m1.getMasa()+m2.getMasa()*factorConv,
                m1.getNombreUnidad(), m1.getAbrevUnidad());
  }

      }
```

Con un mínimo esfuerzo, en comparación con el primer programador, el programador no extremo habría conseguido una solución mucho más satisfactoria, susceptible de manejar cualquier tipo de unidad. La solución del programador XP cumple la función que se le demanda, pero es sumamente inestable. La solución del programador no extremo es genérica, y cumple la misma función que la del otro, sólo que de manera extensible.

La diferencia entre ambas soluciones no reside en que cada una aporte distintas funcionalidades, sino en que cuentan con distintas representaciones internas de los datos. La solución extrema es miope: *piensa* en términos de kilogramos y libras; la otra solución *piensa* en términos más abstractos. ¿Y cómo piensa uno en términos más abstractos? Pues por medio del análisis y el diseño. Dicho de otro modo, pensando un poco antes de escribir el código. Para trabajar poco, hay que pensar mucho.

La fascinación de los programadores XP por el código actúa como el canto de las sirenas sobre los marineros de *La Odisea*. Al final, siempre hay un acantilado donde estrellarse. Una inclinación natural del ser humano es preocuparse de las necesidades inmediatas. ¿Cuántos políticos han ganado unas elecciones prometiendo estabilidad y prosperidad dentro de diez años? Supongo que el efecto 2000 fue una consecuencia irreversible de enfocar las cosas de una manera ágil ("haga que funcione ahora y olvídese del resto").

3.22. Elitismo y misticismo: una mala mezcla.

Cualquiera que indague un poco en las metodologías ágiles encontrara rasgos inconfundibles de misticismo. No en vano algunas de estas metodologías están fuertemente influenciadas por la *new age*. Si bien el misticismo no tiene que ser negativo para la ciencia o la tecnología (es más: puede ser beneficioso), su mezcla con un cierto elitismo desmerece bastante sus propuestas. En un magnífico artículo de Stephen J. Mallor (*Metaphors, Magic, War Numbers, and The Elite*, *eXtreme Programming Pros and Cons: What Questions Remain?* (**[IEEE Computer Society Dynabook, septiembre de 2001)**]), se ponen los puntos sobre las íes:

> ¿Por qué no escribirlo? ¿Hay algo más en marcha? Creo que la respuesta es Sí. Hay algo más, tenga paciencia conmigo aquí, el misticismo de la creación. Si lo escribe, la magia del hechicero se irá y no funcionará nunca más. Si realmente lo examina, después de escribirlo, no será "especial" y no funcionará.
> […]
> El poder de la XP para (es decir, actuando sobre) sus defensores es la glorificación del mago; el_programador_como_artesano, creador de abstracciones, el tejedor de sueños poderosos, el mago. La XP habla a esta gente. Y esto les da permiso para desdeñar los aspectos de ingeniería de nuestra disciplina. Creo que hay una guerra no demasiado bien escondida entre los artesanos/artistas/magos y los ingenieros/científicos/disciplinarios. El rechazo a anotarlo todo (excepto en forma de poción mágica, código) es, bajo esta teoría mía, una barrera tanto para impostores como para competidores potenciales. Dejando de lado los vuelos más salvajes de la imaginación, sin embargo, seguramente hay algún motivo para describir el

> sistema para la posteridad. Después de todo, este esquema completo asume que la tradición verbal puede mantenerse viva de alguna manera. ¿Qué pasa cuando la última persona que mantiene el sistema comienza a babear? Tengo mucha compasión por la idea de que deberíamos trabajar solamente en productos que necesitamos, y no crear nada "extra". Después de todo, si es extra, es, por definición, superfluo. Pero seguramente deberíamos preguntarnos por qué el código es el nivel de abstracción más alto que podemos manejar. (Mi interés reside en modelos ejecutables, y en construir "metáforas" ejecutables, separadas del problema de la aplicación. El modelo se convierte entonces en el nivel de abstracción más alto que usamos, no el código.) Se habla ahora, a propósito, de aplicar la XP para construir modelos. Aunque esto eliminaría algo de mi descontento, no contesta a la pregunta de por qué el modelo es de la manera que es. ¡Escríbalo, maldita sea!

Las declaraciones de algunos gurús y seguidores de las metodologías ágiles son sorprendentes. Veamos, como simple ejemplo, de qué manera se anuncia la conferencia "**You say you want a revolution..."** de Dave West en *Agile Universe* (reproduzco el texto completo, sin traducir, para evitar la sospecha de posibles tergiversaciones): "XP/Agile are 'revolutionary' approaches to software development. The latest in a distinguished lineage of such innovations. A revolutionary innovation does not, however, a revolution make. What forces are at work that will prevent XP/Agile from becoming truly revolutionary? How might the revolution be won? Within the context of these questions we will explore a bit of history, cultural anthropology, mysticism, and Maoism and outline future strategy".

¿Qué significa que las metodologías ágiles sean "verdaderamente revolucionarias"? ¿Qué revolución hay que ganar? Qué fácil resulta mezclar el maoísmo, el misticismo y la programación cuando se vive en el primer mundo y se come caliente día tras día (si todo va mal, siempre queda la *welfare*). Me gustaría saber qué opina el señor West sobre las familias chinas que intercambiaban a sus hijos porque necesitaban alimento y eran incapaces de comerse a sus propios hijos. Canibalismo, sí: consecuencia de las hambrunas derivadas del "Gran salto hacia Adelante" del "Gran Timonel" Mao, el mismo que mostró su desacuerdo con la condena de Stalin pronunciada por Nikita Kruschov. Seguro que eso no lo contó en su conferencia: no sería *cool*. No bastaba con que bolcheviques de salón, enteradillos del parchís y politicastros se llenaran la boca con La Revolución y La Verdad... ahora también los programadores.

No es raro encontrarse, en cualquier foro de XP, mensajes con frases como "Seguir las prácticas extremas no convierte automáticamente a uno en un programador extremo. Lleva años... hay que abrir la mente" o "Hay que olvidar lo aprendido. A mí me costó cuatro años entender la XP" o "Hay que vencer la resistencia del programador. Debes aprender a conocerte a ti mismo: tus debilidades y tus defectos". Cuando leo frases así, siempre pienso que me he equivocado de foro y que me he metido en uno de jardinería Zen para desocupados o de religiones orientales para principiantes. Seguro que algún programador extremo podría encontrar una metáfora del programador como jardinero de árboles enanos, y comparar el lento crecimiento controlado de los bonsáis con las iteraciones de la XP; y podría asimismo explicarnos que se

necesitan muchos años de esfuerzo y de reflexión para llegar a ser un buen jardinero japonés de bonsáis, al igual que para llegar a ser un buen programador o instructor extremo. No digo que no tuviera razón, no, ¿pero no suena más a descubrimiento espiritual que a ingeniería del software?

Ignorante de mí: yo pensaba que la ingeniería del software era una disciplina cuya meta es construir sistemas de software que funcionen, que cumplan los requisitos establecidos y que no sobrepasen los costes y calendarios previstos. Pues no: andaba entre tinieblas; la ingeniería del software busca –según la XP- que "los programadores se diviertan" y que se descubran a sí mismos espiritualmente. Que eso lleva unos años... pues habrá que esperar. **Larga es la senda del samurai extremo.**

El elitismo de algunos gurús y programadores XP resulta bastante evidente. ¿Qué significan esas recomendaciones de dar a los programadores el ambiente y apoyo necesario? ¿Acaso no merece cualquier trabajador o trabajadora desarrollar su actividad laboral en unas condiciones favorables y saludables, y contar con el respeto y apoyo de sus compañeros y superiores? Que yo sepa, los desarrolladores XP en los Estados Unidos o en el Reino Unido no cobran el sueldo mínimo (al contrario: están muy bien pagados); no son galeotes mal pagados ni tienen que bajar a la mina a extraer carbón. Algunos hasta se molestan con la metáfora de que la construcción de software es como un proceso de manufacturación. Deben pensar que ellos son "creadores", no simples obreros. **Programar no es pertenecer a la realeza ni ser un mago del código**: es una actividad profesional que admite la creatividad, al igual que tantas otras.

A veces, este elitismo toma carices peligrosos. Veamos lo que escribió Martha Baer en el número de septiembre de 2003 de la revista *Wired* (http://www.wired.com/wired/archive/11.09/xmen_pr.html):

> En una profesión conocida por sus lobos solitarios y cubos silenciosos, en una cultura ridiculizada habitualmente por sus ineptitudes sociales, poner a los programadores cerca unos de otros parece contraintuitivo, incluso arriesgado. La investigación reciente sobre el autismo sugiere que algunos ingenieros de software podrían sufrir realmente una enfermedad genética que dificulta su capacidad para interaccionar.

Curioso párrafo: es interesante tanto por lo que dice como por lo que no dice. **Lo que dice**: algunos ingenieros de software podrían ser ligeramente anormales. **Lo que no dice**: el programador extremo, que trabaja en grupo, es una persona socialmente integrada, comunicativa, no un solitario defectuoso genéticamente; los programadores extremos están sanos; el resto puede estar enfermo desde la cuna. ¿Cómo curarles? Dando generosas raciones de programación extrema a esos pobres diablos maldecidos por la genética desde su nacimiento. Si eso no es propaganda ponzoñosa, malintencionada y rebosante de "doble lenguaje", ¿qué es?

Johann von Neumann, Kurt Gödel y Alan Turing fueron personas solitarias, poco dadas a la colaboración en equipo. Gödel fue mentalmente inestable durante casi toda su vida, y murió de desnutrición; Turing acabó suicidándose, acosado por una sociedad que quería "normalizarlo" y hacerlo como la mayoría de la gente bienpensante. Ninguno de ellos hubiera

desacreditado jamás las ideas de un oponente insinuando que “podía tener enfermedades genéticas”. Y cualquiera de ellos tenía muchísimas cosas más interesantes que decir que todos los metodólogos ágiles juntos. En paz descansen.

Figura 23. ¿Necesitaban estas personas “curarse”? ¿Les habría ayudado la “teorización en parejas”? Si hubieran sido sanados, quizás habrían sido muy felices, habrían sonreído a todo el mundo y habrían alegrado todas las fiestas. Y quizás se habrían dedicado a actividades más sociables y lucrativas, como crear metodologías para construir software.

La opinión de la señora Baer no es una rara excepción: con frecuencia se leen frases como “hay que vencer la resistencia a la programación extrema” o “los programadores tradicionales oponen resistencia”. No: no es que opongan resistencia; es que conservan el sentido común. Y éste les lleva a no aceptar afirmaciones revolucionarias sin pruebas revolucionarias. Alegar *resistencia* suena a cháchara psicoanalista. Suena a “usted opone resistencia al psicoanálisis porque no acepta que está enamorado de su madre, porque de pequeño se vestía de niña, porque se le murió su perrito y nunca lo superó… En suma, porque no es usted normal”. Este truco es excelente para evitar toda discusión racional o basada en hechos. No sería necesario alegar resistencia si hubiera hechos sobre la mesa, pero…

Por último, cotejemos la XP, escasa de méritos, con alguien sobrado de ellos: Alexandre-Gustave Eiffel, quien construyó la torre que lleva su apellido para la Exposición Universal de 1889. La torre Eiffel se ajustó al presupuesto establecido, se acabó en el plazo previsto y ha sobrepasado con creces el tiempo de vida para el que fue pensada. Cuando un periodista preguntó al señor Eiffel qué pensaba de su obra, contestó: “Es el resultado de aplicar unos cuantos principios de ingeniería. Me he limitado a seguirlos con paciencia y rigor”. Nada de ínfulas o de elitismo. Sólo la humildad de quien sabe que ha obrado siguiendo los principios de su profesión. ¿Queda algo que añadir?

Nota biográfica del Autor: Miguel Ángel Abián nació en Soria (1972). Se licenció en Ciencias Físicas en 1995 por la U. de Valencia y consiguió la suficiencia investigadora en 1997 dentro del Dpto. Física Aplicada de la U.V con una tesina acerca de relatividad general y electromagnetismo. Además ha realizado diversos cursos de Postgrado sobre bases de datos, lenguajes de programación Web, sistemas Unix, comercio electrónico, firma electrónica, UML y Java. Ha participado en diversos programas de investigación TIC relacionados con el estudio de fibras ópticas y cristales fotónicos, y ha publicado diversos artículos en el *IEEE Transactions on Microwave Theory and Techniques* relacionados con el análisis de guías de onda inhomogéneas y guías de onda elípticas.

En el ámbito laboral ha trabajado como gestor de carteras y asesor fiscal para una agencia de bolsa y actualmente trabaja en el Laboratorio del Mueble Acabado de AIDIMA (Instituto Tecnológico del Mueble y Afines), ubicado en Paterna (Valencia), en tareas de normalización y certificación. En dicho centro se están desarrollando proyectos europeos de comercio electrónico B2B para la industria del mueble basados en Java y XML (más información en www.aidima.es). Ha impartido formación en calidad, normalización y programación para ELKEDE (Grecia), CETEBA (Brasil) y CETIBA (Túnez), entre otros.

Últimamente, aparte de asesorar a diversas empresas, trabaja en la implementación Java de un emulador del microprocesador MIPS multiciclo y es investigador en el proyecto INTEROP (Interoperabilidad de software) del Sexto Programa Marco de la Comisión Europea.

Sus intereses actuales son el diseño asistido por ordenador de guías de ondas y cristales fotónicos, la evolución de la programación orientada a objetos, Java, el intercambio electrónico de datos, el surrealismo y París, siempre París.