Métodos Heterodoxos en Desarrollo de Software

Contenidos:

- Introducción ... 2
- La ortodoxia metodológica ... 4
- Orígenes de la heterodoxia ... 6
- Los Métodos Ágiles ... 11
  - eXtreme Programming (XP) ... 12
  - Scrum ... 18
  - Evolutionary Project Management (Evo) ... 23
  - Crystal Methods ... 29
  - Feature Driven Development (FDD) ... 36
  - Rational Unified Process (RUP) ... 40
  - Dynamic Systems Development Method (DSDM) ... 43
  - Adaptive Software Development ... 47
  - Agile Modeling ... 50
  - Lean Development (LD) y Lean Software Development (LSD) ... 53
- Microsoft Solutions Framework y los Métodos Agiles ... 55
- Métodos y Patrones ... 60
- Agilidad, Caos y Complejidad ... 63
- Anti-agilidad: La crítica de los Métodos Ágiles ... 65
- Conclusiones ... 68
- Vínculos ágiles ... 71
- Referencias bibliográficas ... 73

Métodos Heterodoxos en Desarrollo de Software

Versión 1.0 – Abril de 2004

Carlos Reynoso – UNIVERSIDAD DE BUENOS AIRES

Revisión técnica de Nicolás Kicillof – Universidad de Buenos Aires

Introducción

> Los métodos Ágiles, tales como Lean Development, eXtreme Programming y Adaptive Software Development, son estrategias de desarrollo de software que promueven prácticas que son adaptativas en vez de predictivas, centradas en la gente o en los equipos, iterativas, orientadas hacia prestaciones y hacia la entrega, de comunicación intensiva, y que requieren que el negocio se involucre en forma directa. Comparando esos atributos con los principios fundacionales de MSF, se encuentra que MSF y las metodologías ágiles están muy alineadas tanto en los principios como en las prácticas para el desarrollo de software en ambientes que requieren un alto grado de adaptabilidad.
>
> - Documentación de Microsoft Solutions Framework 3.0 [MS03]

Ni duda cabe que a finales de la década de 1990 dos grandes temas irrumpieron en las prácticas de la ingeniería de software y en los métodos de desarrollo: el diseño basado en patrones y los métodos ágiles. De estos últimos, el más resonante ha sido la Programación Extrema (XP), que algunos consideran una innovación extraordinaria y otros creen cínica [Rak01], extremista [McC02], falaz [Ber03] o perniciosa para la salud de la profesión [Kee03]. Patrones y XP se convirtieron de inmediato en *hypes* de discusión masiva en la industria y de fuerte presencia en la Red. Al primero de esos temas el mundo académico lo está tratando como un asunto respetable desde hace un tiempo; el otro recién ahora se está legitimando como tópico serio de investigación. La mayor parte de los documentos proviene todavía de los practicantes, los críticos y los consultores que impulsan o rechazan sus postulados. Pero el crecimiento de los métodos ágiles y su penetración ocurre a un ritmo pocas veces visto en la industria: en tres o cuatro años, según el Cutter Consortium, el 50% de las empresas define como "ágiles" más de la mitad de los métodos empleados en sus proyectos [Cha04].

Los métodos ágiles (en adelante MAs) constituyen un movimiento heterodoxo que confronta con las metodologías consagradas, acordadas en organismos y apreciadas por consultores, analistas de industria y corporaciones. Contra semejante adversario, los MAs se expresaron a través de manifiestos y libros en tono de proclama, rehuyendo (hasta hace poco) toda especificación formal. El efecto mediático de esos manifiestos ha sido explosivo y ocasionó que la contienda entre ambas formas haya sido y siga siendo enconada. Lejos de la frialdad burocrática que caracteriza a las crónicas de los argumentos ortodoxos, se ha tornado común referirse a sus polémicas invocando metáforas bélicas y apocalípticas que hablan de "la batalla de los gurúes" [Tra02], "el

gran debate de las metodologías" [Hig01], "las guerras religiosas", "el fin del mundo" [McC02], un "choque cultural" [Cha04], "la venganza de los programadores" [McB02] o "la muerte del diseño" [Fow01].

Lo que los MAs tienen en común (y lo que de aquí en más obrará como una definición de los mismos) es su modelo de desarrollo incremental (pequeñas entregas con ciclos rápidos), cooperativo (desarrolladores y usuarios trabajan juntos en estrecha comunicación), directo (el método es simple y fácil de aprender) y adaptativo (capaz de incorporar los cambios). Las claves de los MAs son la velocidad y la simplicidad. De acuerdo con ello, los equipos de trabajo se concentran en obtener lo antes posible una pieza útil que implemente sólo lo que sea más urgente; de inmediato requieren *feedback* de lo que han hecho y lo tienen muy en cuenta. Luego prosiguen con ciclos igualmente breves, desarrollando de manera incremental. Estructuralmente, los MAs se asemejan a los RADs (desarrollo rápido de aplicaciones) más clásicos y a otros modelos iterativos, pero sus énfasis son distintivos y su combinación de ideas es única.

Si hubo una sublevación no fue inmotivada. Diversos estudios habían revelado que la práctica metodológica fuerte, con sus exigencias de planeamiento y sus técnicas de control, en muchos casos no brindaba resultados que estuvieran a la altura de sus costos en tiempo, complejidad y dinero. Investigaciones como la de Joe Nandhakumar y David Avison [NA99], en un trabajo de campo sobre "la ficción del desarrollo metodológico", denunciaban que las metodologías clásicas de sistemas de información "se tratan primariamente como una ficción necesaria para presentar una imagen de control o para proporcionar estatus simbólico" y que dichas metodologías son demasiado ideales, rígidas y mecanicistas para ser utilizadas al pie de la letra. Duane Truex, Richard Baskerville y Julie Travis [TBT00] toman una posición aún más extrema y aseguran que es posible que los métodos tradicionales sean "meramente ideales inalcanzables y 'hombres de paja' hipotéticos que proporcionan guía normativa en situaciones de desarrollo utópicas". En su reclamo de un desarrollo a-metódico, consideran que las metodologías estándares se basan en una fijación de objetivos pasada de moda e incorrecta y que la obsesión de los ingenieros con los métodos puede ser inhibidora de una adecuada implementación, tanto a nivel de sistemas como en el plano de negocios.

Pero también hay cientos de casos documentados de éxitos logrados con metodologías rigurosas. En ocasiones, el entusiasmo de los promotores de las revueltas parece poco profesional, como si su programa de crítica, muchas veces bien fundado, fuera de mayor importancia que su contribución positiva. También ellas, sin embargo, tienen su buen catálogo de triunfos. Algunos nombres respetados (Martin Fowler con el respaldo de Cutter Consortium, Dee Hock con su visión caórdica, Philippe Kruchten con su RUP adaptado a los tiempos que corren) e incluso Ivar Jacobson [Jac02] consideran que los MAs constituyen un aporte que no sería sensato dejar de lado. No habría que tratarlo entonces como si fuera el capricho pasajero de unos pocos *hackers* que han leído más arengas posmodernas de lo aconsejable.

En este documento se presentará una breve reseña de los MAs junto con una descripción sucinta de la situación y las tendencias actuales. No se pretende que el texto brinde una orientación operativa ni tampoco una evaluación de los métodos; se trata sólo de una visión de conjunto que tal vez ayude a comprender una de las alternativas existentes en la organización del proceso de desarrollo. Se ha procurado mantener una distancia crítica;

no se encontrarán aquí las expresiones que suelen proliferar en los textos y en la Red en las que se "define" a XP afirmando que es "un conjunto de valores, principios y prácticas para el desarrollo rápido de software *de alta calidad* que proporciona *el valor más alto* para el cliente *en el menor tiempo posible*". No es que la postura de este estudio sea equidistante, como procuran serlo, por ejemplo, los exámenes de Robert Glass [Gla01] o Barry Boehm [Boe02a]; sencillamente, la presente es una introducción analítica y no una tabla de evaluación.

Se podrá advertir en el cuerpo de este texto que las referencias comparativas y contextuales a los MAs apuntan muchas más veces a la ingeniería de software que a la arquitectura; la razón es que la elaboración metodológica de la arquitectura es reciente y se está formulando en el SEI y en otros organismos contemporáneamente al desarrollo de los MAs. Recién ahora, en otras palabras, la arquitectura de software está elaborando sus metodologías para el ciclo de vida; si éstas se asemejan más a las ágiles que a las ortodoxas es una pregunta importante que se habrá de responderse en documentos separados.

Una vez más, se examinará con detenimiento la relación de concordancia y complementariedad entre los métodos ágiles y los principios que articulan Microsoft Solutions Framework (MSF), un tema sustancial al que se ha dedicado una sección específica [pág. 55]. También se ha incluido un capítulo sobre el uso de patrones en MAs y otro sobre la respuesta crítica de la comunidad metodológica frente al avance arrollador de los nuevos métodos [pág. 65]. Otros documentos de esta serie detallarán algunos aspectos que aquí se desarrollan concisamente, tales como la relación entre los métodos ágiles con el paradigma de las ciencias de la complejidad y el caos.

La ortodoxia metodológica

Los MAs no surgieron porque sí, sino que estuvieron motivados (a) por una conciencia particularmente aguda de la crisis del software, (b) por la responsabilidad que se imputa a las grandes metodologías en la gestación de esa crisis y (c) por el propósito de articular soluciones.

Los organismos y corporaciones han desarrollado una plétora de estándares comprensivos que han ido jalonando la historia y poblando los textos de metodologías e ingeniería de software: CMM, Spice, BootStrap, TickIt, derivaciones de ISO9000, SDCE, Trillium [Wie98] [Wie99] [She97]. Algunos son verdaderamente métodos; otros, metodologías de evaluación o estimación de conformidad; otros más, estándares para metodologías o meta-modelos. Al lado de ellos se encuentra lo que se ha llamado una ciénaga de estándares generales o específicos de industria a los que se someten organizaciones que desean (o deben) articular sus métodos conforme a diversos criterios de evaluación, vehículos de selección de contratistas o marcos de referencia. Hay muchos *frameworks* y disciplinas, por cierto; pero lo primordial no es la proliferación de variedades, sino la sujeción de todos los ejemplares a unos pocos tipos de configuraciones o flujos posibles. Llamaremos aquí *modelos* a esas configuraciones.

Los modelos de los métodos clásicos difieren bastante en su conformación y en su naturaleza, pero exaltan casi siempre las virtudes del planeamiento y poseen un espíritu normativo. Comienzan con la elicitación y el análisis completo de los requerimientos del

usuario. Después de un largo período de intensa interacción con usuarios y clientes, los ingenieros establecen un conjunto definitivo y exhaustivo de rasgos, requerimientos funcionales y no funcionales. Esta información se documenta en forma de especificaciones para la segunda etapa, el diseño, en el que los arquitectos, trabajando junto a otros expertos en temas puntuales (como ser estructuras y bases de datos), generan la arquitectura del sistema. Luego los programadores implementan ese diseño bien documentado y finalmente el sistema completo se prueba y se despacha.

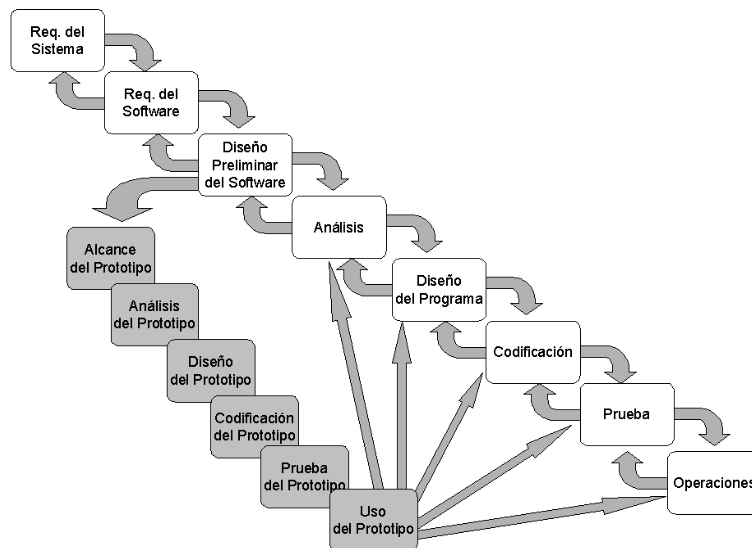
Modelo	Versión de origen	Características
Modelo en cascada	Secuencial: Bennington 1956 – Iterativo: Royce 1970 – Estándar DoD 2167-A	Secuencia de requerimiento, diseño del sistema, diseño de programa, codificación, pruebas, operación y mantenimiento
Modelo en cascada c/ fases superpuestas	Cf. McConnell 1996:143-144	Cascada con eventuales desarrollos en paralelo (Modelo Sashimi)
Modelo iterado con prototipado	Brooks 1975	Iterativo – Desarrollo incremental
Desarrollo rápido (RAD)	J. Martin 1991 – Kerr/Hunter 1994 – McConnell 1996	Modelo lineal secuencial con ciclos de desarrollo breves
Modelo V	Ministerio de Defensa de Alemania 1992	Coordinación de cascada con iteraciones
Modelo en espiral	Barry Boehm 1988	Iterativo – Desarrollo incremental. Cada fase no es lineal, pero el conjunto de fases sí lo es.
Modelo en espiral win-win	Barry Boehm 1998	Iterativo – Desarrollo incremental – Aplica teoría-W a cada etapa
Modelo de desarrollo concurrente	Davis y Sitaram 1994	Modelo cíclico con análisis de estado
Modelo de entrega incremental (*Staged delivery*)	McConnell 1996: 148	Fases tempranas en cascada – Fases ulteriores descompuestas en etapas

A fines del siglo XX había un abanico de tipos de procesos o modelos disponibles: el más convencional era el modelo en cascada o lineal-secuencial, pero al lado de él había otros como el modelo "V", el modelo de construcción de prototipos, el de desarrollo rápido o RAD, el modelo incremental, el modelo en espiral básico, el espiral *win-win*, el modelo de desarrollo concurrente y un conjunto de modelos iterativos o evolutivos (basados en componentes, por ejemplo) que en un primer momento convivían apaciblemente con los restantes, aunque cada uno de ellos se originaba en la crítica o en la percepción de las limitaciones de algún otro. Algunos eran prescriptivos, otros descriptivos. Algunos de los modelos incluían iteraciones, incrementos, recursiones o bucles de retroalimentación; y algunos se preciaban también de rápidos, adaptables y expeditivos.

La tabla que se encuentra en estas páginas ilustra el repertorio de los métodos que podríamos llamar "de peso completo" o "de fuerza industrial", que en sus variantes más rigurosas imponen una prolija especificación de técnicas correspondientes a los diferentes momentos del ciclo de vida. Es importante señalar que en esta tabla sólo se consignan las grandes formas de proceso metodológico, y no técnicas específicas que pueden aplicarse cualquiera sea esa forma, como las técnicas de ingeniería de software asistidas por computadoras (CASE), las herramientas de cuarta generación, las técnicas de modelado estático (Lai) o dinámico (Forrester), el modelo de software de sala limpia [MDL87] o los modelos evaluativos como el Capability Maturity Model (CMM) del SEI, que

popularmente se considera ligado a los modelos en cascada. De hecho ni CMM está sujeto a un modelo particular, ni constituye un modelo de proceso de ingeniería de software en el sentido estricto de la palabra; es un canon de evaluación que establece criterios para calificar la madurez de un proyecto en cinco niveles que van de lo caótico a la optimización continua [Pau02].

Ha habido una injusticia histórica con Winston Royce [Roy70] y su modelo en cascada: en su versión original, documentada en la imagen, era claramente iterativo [Ber03]. Por alguna razón pasó a la posteridad como la encarnación de una metodología secuencial.



El modelo original en cascada, basado en [PP03]

Hay unos cuantos modelos más o menos clásicos que no hemos considerado. Steve McConnell [McC96], por ejemplo, describe un modelo *code-and-fix*, un modelo en cascada con subproyectos, otro modelo en cascada con reducción de riesgo, un modelo de diseño sujeto a agenda y un modelo de prototipado evolutivo (el antepasado de Evo) que aquí hemos transferido al otro lado de la divisoria entre métodos pesados y ligeros. El tema de nuestro estudio, empero, no es la taxonomía de los modelos. La mejor referencia para las formas modélicas clásicas anteriores a 1990 sigue siendo el libro de Peter DeGrace y Leslie Stahl *Wicked problems, righteous solutions* [DS90], un verdadero catálogo de los paradigmas metodológicos principales en ingeniería de software. Lo importante aquí es que toda esta diversidad se va visto subsumida de golpe en un solo conjunto. Por diversas razones, legítimas o espurias, todo comenzó a verse bajo una nueva luz. Cuando estas cosas suceden no cabe sino hablar de una revolución.

Orígenes de la heterodoxia

No hace falta comulgar con las premisas más extremas de los MAs para encontrar aspectos cuestionables en todos y cada uno de los modelos clásicos. Aun los manuales introductorios de ingeniería de software incluyen la lista de limitaciones bien conocidas de la mayor parte de ellos, con el modelo en cascada, lejos, como el más castigado. Se le objeta su rigidez, su obstinación por exigir una estipulación previa y completa de los requerimientos, su modelo inapropiado de correcciones, su progresivo distanciamiento de la visión del cliente y su morosidad para liberar piezas ejecutables. En los últimos dos o

tres años de la década de 1990, las críticas a la metodología convencional aumentaron en intensidad y detonaron por todas partes.

Aunque la gestación de la rebelión fue larga y tortuosa, pareció que de pronto, para muchos, los modelos consagrados se tornaron inadmisibles. Circularon también unas cuantas evaluaciones que revelaron serias fallas en proyectos regidos por ellos, y se refrescó la idea de la crisis del software, que en realidad es un asunto que arrastra una historia de más de cuarenta años. Para colmo, en 1995 se publicó la edición de plata del clásico de Fred Brooks *The Mythical Man-Month* [Bro75], que con años de anticipación había revelado la estrechez de la mentalidad lineal. En *MMM* Brooks se había expedido en contra del método en cascada:

> Gran parte de los procedimientos actuales de adquisición de software reposan en el supuesto de que se puede especificar de antemano un sistema satisfactorio, obtener requisitos para su construcción, construirlo e instalarlo. Pienso que este supuesto es fundamentalmente equivocado y que muchos de los problemas de adquisición se originan en esta falacia.

En la vieja edición Brooks había dicho "Haga un plan para tirarlo. Lo hará de todos modos". En la nueva edición, fingió retractarse para atacar con más fuerza: "No construya uno para tirar. El modelo en cascada es erróneo!". Comenzaron a publicarse, una tras otra, historias de fracasos del método, o se refrescó el recuerdo de las que habían sido editadas. El Departamento de Defensa, que en la década de 1980 había promovido el ciclo de vida en cascada DOD-STD-2167, de infausta memoria, comenzó a asociarlo a una serie de reveses continuos. En 1987, se recomendó la implementación de métodos iterativos y evolutivos, de lo que resultó el MIL-STD-498. La NATO, la FDA y otros organismos han experimentado historias semejantes [Lar04]. En la segunda mitad de la década de 1990, también el reputado ISO 9000 cayó bajo fuego; se encontraron correlaciones negativas entre la adscripción a los estándares y la calidad de los productos, y la autoridad británica que regula la publicidad mandó que el British Standard Institute se abstuviera de proclamar que la adhesión a ISO mejoraría la calidad o la productividad de las soluciones [ESPI96].

Ante esa evidencia, los expertos, en masa, aconsejaron el abandono de los métodos en cascada y las normativas fuertes; eran nombres influyentes: Harlan Mills, Tom Gilb, Barry Boehm, James Martin, Tom DeMarco, Ed Yourdon y muchos más [Lar04]. Ya eran conocidos por diversos logros y muchos de ellos estaban, además, elaborando sus propias alternativas dinámicas, iterativas, evolutivas y en suma, ágiles. El surgimiento de los MAs no fue, mal que le pese a críticos como Steven Rakitin [Rak01], Gerold Keefer [Kee03], Stephen Mellor [Mel03], Edward Berard [Ber03], Matt Stephens o Doug Rosenberg [SR03], una revuelta de aficionados adictos al código sin destreza en el desarrollo corporativo. A los MAs les pueden caber muchos cuestionamientos, pero de ningún modo resulta razonable hacerles frente con argumentos *ad hominem*.

En las vísperas del estallido del Manifiesto ágil, todos los autores sensibles al encanto de los métodos incrementales que comenzaban a forjarse o a resurgir tenían un enemigo común. Aunque CMM no es estrictamente un método, ni un método en cascada en particular, lo cierto es que se lo comenzó a identificar como el modelo no-iterativo de peso pesado por excelencia. Hay razones para ello. CMM fue leído con espíritu de cascada a fines de los 80 y durante buena parte de la década siguiente. Aunque una

solución producida con una estrategia iterativa podría llegar a calificar como CMM-madura, las discusiones iniciales de CMM tienen un fuerte tono de planeamiento predictivo, desarrollo basado en planes y orientado por procesos [SEI03]. Craig Larman [Lar04] sostiene que muchos de los certificadores y consultores en CMM tienen una fuerte formación en los valores del modelo en cascada y en procesos prescriptivos, con poca experiencia en métodos iterativos e incrementales.

Los MAs difieren en sus particularidades, pero coinciden en adoptar un modelo iterativo que la literatura más agresiva opone dialécticamente al modelo ortodoxo dominante. El modelo iterativo en realidad es bastante antiguo, y se remonta a ideas plasmadas por Tom Gilb en los 60 y por Vic Basili y Joe Turner en 1975 [BT75]. Como quiera que se los llame, la metodología de la corriente principal se estima opuesta en casi todos los respectos a las nuevas, radicales, heterodoxas, extremas, adaptativas, minimalistas o marginales. El antagonismo entre ambas concepciones del mundo es, según los agilistas, extrema y abismal. Escribe Bob Charette, uno de los creadores de Lean Development: "La burocracia e inflexibilidad de organizaciones como el Software Engineering Institute y de prácticas como CMM las hacen cada vez menos relevantes para las circunstancias actuales del desarrollo de software" [en High00b]. Ken Orr, del Cutter Consortium, dice que la diferencia entre un asaltante de bancos y un metodólogo de estilo CMM es que con un asaltante se puede negociar [Orr03]. La certificación en CMM, agrega, depende más de una buena presentación en papel que de la calidad real de una pieza de software; tiene que ver más con el seguimiento a ciegas de una metodología que con el desarrollo y puesta en producción de un sistema de estado del arte.

La mayoría de los agilistas no cree que CMM satisfaga sus necesidades. Algunos de ellos lo exponen gráficamente: Turner y Jain afirman que si uno pregunta a un ingeniero de software típico si cree que CMM es aplicable a los MAs, la respuesta probablemente oscile entre un una mirada de sorpresa y una carcajada histérica [TJ02]. Una razón para ello es que CMM constituye una creencia en el desarrollo de software como un proceso definido que puede ser especificado en detalle, que dispone de algoritmos precisos y que los resultados pueden ser medidos con exactitud, usando las medidas para refinar el proceso hasta que se alcanza el óptimo. Para proyectos con algún grado de exploración e incertidumbre, los desarrolladores ágiles simplemente no creen que esos supuestos sean válidos. A Fred Brooks no le habría cabido la menor duda: en un método clásico puede haber bucles e iteraciones, pero la mentalidad que está detrás de su planificación es fatalmente lineal. Hay entre ambas posturas una división fundamental que muchos creen que no puede reconciliarse mediante el expediente de recurrir a algún cómodo término medio [Hig02a], aunque tanto desde CMM [Pau01] como desde el territorio ágil [Gla01] [Boe02a] se han intentado fórmulas de apaciguamiento.

Como arquetipo de la ortodoxia, CMM tiene compañía en su cuadrante. El Project Management Institute (PMI), la segunda *bête noire* de los iconoclastas, ha tenido influencia en niveles gerenciales a través de su prestigioso Body of Knowledge (PMBOK) [PMB04]. Aunque este modelo es una contribución valiosa y reconoce el mérito de los nuevos métodos, los contenidos más tempranos de PMBOK son también marcadamente prescriptivos y depositan mucha confianza en la elaboración del plan y el desenvolvimiento de un trabajo conforme a él.

A comienzos del siglo XXI estos modelos estaban ya desacreditados y se daban las condiciones para que irrumpiera el Manifiesto de los métodos ágiles. Kent Beck, el líder de XP, comentó que sería difícil encontrar una reunión de anarquistas organizacionales más grande que la de los 17 proponentes de MAs que se juntaron en marzo de 2001 en Salt Lake City para discutir los nuevos métodos. Lo que surgió de su reunión fue el Manifiesto Ágil de Desarrollo de Software [BBB+01a]. Fue todo un acontecimiento, y aunque ha sido muy reciente tiene la marca de los sucesos que quedan en la historia. Bob Charette y Ken Orr, independientemente, han comparado el sentido histórico del Manifiesto con las Tesis de Lutero; ambos desencadenaron guerras dogmáticas [Orr03].

En el encuentro había representantes de modelos muy distintos, pero todos compartían la idea de que era necesaria una alternativa a los métodos consagrados basados en una gestión burocrática, una documentación exhaustiva y procesos de peso pesado. Resumían su punto de vista argumentando que el movimiento Ágil no es una anti-metodología:

> De hecho, muchos de nosotros deseamos restaurar credibilidad a la palabra metodología. Queremos recuperar un equilibrio. Promovemos el modelado, pero no con el fin de archivar algún diagrama en un polvoriento repositorio corporativo. Promovemos la documentación, pero no cientos de páginas en tomos nunca mantenidos y rara vez usados. Planificamos, pero reconocemos los límites de la planificación en un entorno turbulento [BBB+01b].

El Manifiesto propiamente dicho [BBB+01a] reza como sigue:

> Estamos poniendo al descubierto formas mejores de desarrollo de software, haciéndolo y ayudando a otros a que lo hagan. A través de este trabajo hemos llegado a valorar:
>
> - Los individuos y la interacción por encima de los procesos y herramientas.
> - El software que funciona por encima de la documentación abarcadora.
> - La colaboración con el cliente por encima de la negociación contractual.
> - La respuesta al cambio por encima del seguimiento de un plan.
>
> Aunque hay valor en los elementos a la derecha, valorizamos más los de la izquierda.

Los firmantes del Manifiesto fueron Kent Beck (XP), Mike Beedle, Arie van Bennekum (DSDM), Alistair Cockburn (Crystal), Ward Cunnimgham (XP), Martin Fowler (XP), James Grenning (XP), Jim Highsmith (ASD), Andrew Hunt (Pragmatic Programming), Ron Jeffries (XP), Jon Kern (FDD), Brian Marick, Robert C. Martin (XP), Steve Mellor, Ken Schwaber (Scrum), Jeff Sutherland (Scrum) y Dave Thomas (Pragmatic Programming). Hay predominio demográfico de XP (6 sobre 17) y, considerando nuestra selección de diez MAs, se percibe la falta de delegados de Evo, Agile Modeling, Lean Development y RUP.

Al lado del Manifiesto se han especificado los principios que rigen a la comunidad de MAs [BBB+01b]:

1. Nuestra prioridad más alta es satisfacer al cliente a través de la entrega temprana y continua de software valioso.
2. Los requerimientos cambiantes son bienvenidos, incluso cuando llegan tarde en el desarrollo. Los procesos ágiles se pliegan al cambio en procura de una ventaja competitiva para el cliente.

3. Entregar con frecuencia software que funcione, desde un par de semanas hasta un par de meses, con preferencia por las escalas de tiempo más breves.
4. La gente de negocios y los desarrolladores deben trabajar juntos cotidianamente a través de todo el proyecto.
5. Construir proyectos en torno de individuos motivados. Darles la oportunidad y el respaldo que necesitan y procurarles confianza para que realicen la tarea.
6. La forma más eficiente y efectiva de comunicar información de ida y vuelta dentro de un equipo de desarrollo es mediante la conversación cara a cara.
7. El software que funciona es la medida primaria de progreso.
8. Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante indefinidamente.
9. La atención continua a la excelencia técnica enaltece la agilidad.
10. La simplicidad (el arte de maximizar la cantidad de trabajo que no se hace) es esencial.
11. Las mejores arquitecturas, requerimientos y diseños emergen de equipos que se auto-organizan.
12. A intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo, y ajusta su conducta en consecuencia.

El mero hecho de que los MAs se expresen a través de un Manifiesto urdido en una tertulia ocasional y no de un estándar elaborado durante años en un organismo, está señalando una tajante diferencia de actitud. No será la única. Mientras la terminología de CMM, EUP y otros métodos ortodoxos es prolija y circunspecta, las nomenclaturas de muchos de los MAs están plagadas de figuras del lenguaje procedentes de los dominios semánticos más diversos: "Scrum", por empezar, y luego "gallinas", "cerdos", "corridas" (*sprints*), pre-juego, juego, pos-juego y montaje (*staging*) en Scrum; "púas" (*spikes*) en Scrum y XP; "irradiadores" en Crystal.

Al lado de esos lexemas unitarios, en los nombres de principios, estrategias y técnicas abundan los aforismos y máximas como "El Minimalismo es Esencial", "Todo se puede cambiar", "80% hoy en vez de 100% mañana" en Lean Development, "Mirando basura" en LSD; el "Cono del Silencio" o el "Esqueleto Ambulante" en Crystal Clear; "Ligero de equipaje" en AM; "Documentos para tí y para mí" en ASD; "Gran Jefe" y la más publicitada de todas, "YAGNI" (acrónimo de "No vas a necesitarlo") en XP. También hay una producción destacada de citas citables o "agilismos" como éste de Ron Jeffries: "Lo lamento por su vaca; no sabía que era sagrada". Podría argumentarse que esta búsqueda de peculiaridad de expresión conspira contra la homogeneización de la terminología que ha sido uno de los motores de las iniciativas y lenguajes unificados como RUP o UML; pero también es un valor diferencial: nadie podrá jamás confundir un método ágil paradigmático con otro que no lo es.

Hay que subrayar que no todos los MAs comparten esta forma simbólica de locución, reminiscente de la neolengua de George Orwell; pues si bien la mayoría de ellos rubricó el Manifiesto, converge en el espíritu de ciertas ideas, toma prestadas prácticas de métodos aliados y hasta emprende una buena proporción de sus proyectos combinándose con otros, el conjunto ágil presenta ejemplares de naturaleza muy dispar. Hay en ese

conjunto dos mundos: uno es auto-organizativo, anárquico, igualitario o emergente de abajo hacia arriba; el otro es la expresión de una disciplina de *management*, innovadora por cierto, pero muchas veces más formal que transgresora, más rigurosa que temeraria. Se volverá a esta discusión en las conclusiones, pero se anticipa la idea ahora, para que en la lectura que sigue no se confunda lo más radical con lo más representativo.

Los Métodos Ágiles

Existen unos cuantos MAs reconocidos por los especialistas. En este estudio se analizarán con algún detenimiento los que han alcanzado a figurar en la bibliografía mayor y cuentan en su haber con casos sustanciales documentados. Ha habido unos cuantos más que típicamente se presentaron en ponencias de simposios de la comunidad y en seguida se esfumaron. Hacia el final se dedicará un párrafo para caracterizar a este género colateral, más que para describir sus ejemplares. Lo mismo se aplica a los métodos híbridos y a los dialectos (XBreed, XP@Scrum, dX, Grizzly, Enterprise XP, IndustrialXP). La idea no es hacer un censo de todo el campo, sino tratar las metodologías de real importancia.

Consideraremos entonces los diez MAs que han adquirido masa crítica: Extreme Programming, Scrum, Evo, Crystal Methods, Feature Driven Development, RUP, Dynamic Systems Development Method, Adaptive Software Development, Agile Modeling y Lean Development. Aunque hemos adoptado un foco restringido, de los *surveys* existentes a la fecha el presente estudio es el que contempla el mayor número de variedades. Los demás [Gla01] [ASR+02] [Hig02a] [CLC03] [Lar04] siempre omiten algunos importantes.

Dejaremos fuera de consideración un conjunto de formulaciones posmodernas, centradas en la acción, fractales, humanistas o caopléjicas porque entendemos que no han estipulado debidamente sus heurísticas ni las definiciones operacionales requeridas en la ingeniería de soluciones intensivas. Excluiremos Agile Model Driven Development de Scott Ambler (una versión ágil de MDD) y también el modelo en que se basa. Si se incluyera, se admitiría que un conjunto enorme de estrategias de desarrollo "guiadas" de alguna manera[1] (**-driven development*) se maquillaran un poco y se redefinieran como MAs, extendiendo la población de los géneros existentes más allá de lo que puede manejarse. AMDD es, por otra parte, una variante de AMD. Se excluirá también Pragmatic Programming, por más que sus responsables, Andrew Hunt y David Thomas, hayan firmado el Manifiesto; aunque en [ASR+02] se le consagra un capítulo, no se trata de una metodología articulada que se haya elaborado como método ágil, sino de un conjunto de *best practices* ágiles publicadas en *The Pragmatic Programmer* [HT00]. El mismo temperamento se aplicará a Internet-Speed Development (ISD). Por último, se hará caso omiso de Agile Database Methodology, porque es una metodología circunscripta y no un método de propósito general.

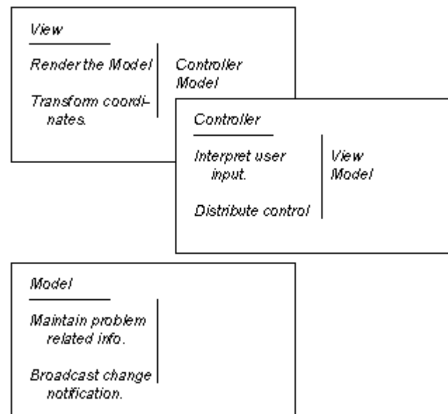
---

[1] Contract-Driven Development, Model Driven Development, User-Driven Development, Requirements-Driven Development, Design-Driven Development, Platform Driven Development, Use Case-Driven Development, Domain-Driven Development, Customer-Driven Development, Application-Driven Development, Pattern-Driven Development.

Con unas pocas excepciones, los *surveys* de MAs disponibles poseen estructuras demasiado complejas y heterogéneas para que puedan visualizarse con comodidad las regularidades del conjunto. Dado que el presente examen no es un estudio comparativo a fondo sino una visión preliminar, expondremos cada método de acuerdo con un plan estable que contemple su historia, sus valores, sus roles, sus artefactos, sus prácticas y su ciclo de vida. Cuando haya que representar diagramas de procesos, se tratará de reflejar el estilo iconográfico de los textos o páginas originales, apenas homogeneizado en su nivel de detalle para su mejor correlación.

eXtreme Programming (XP)

La Programación Extrema es sin duda alguna el método ágil que primero viene a la mente cuando se habla de modelos heterodoxos y el más transgresor entre ellos. Es, de acuerdo con el *survey* de Cutter Consortium, también el más popular entre los MAs: 38% del mercado ágil contra 23% de su competidor más cercano, FDD. Luego vienen Adaptive Software Development con 22%, DSDM con 19%, Crystal con 8%, Lean Development con 7% y Scrum con 3%. Todos los demás juntos suman 9% [Cha04].



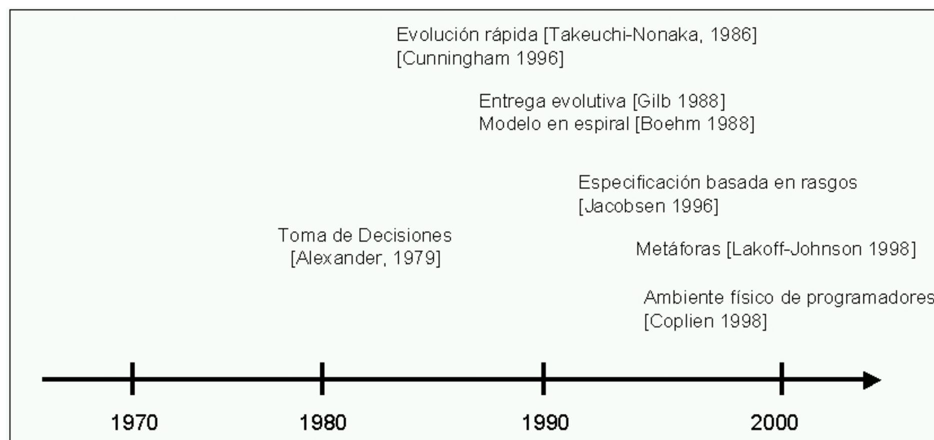
Tarjetas CRC del patrón MVC según [BC89]

A mediados de la década de 1980, Kent Beck y Ward Cunningham trabajaban en un grupo de investigación de Tektronix; allí idearon las tarjetas CRC y sentaron las bases de lo que después serían los patrones de diseño y XP. Los patrones y XP no necesitan presentación, pero las tarjetas CRC tal vez sí. Se trata de simples tarjetas de papel para fichado, de 4x6 pulgadas; CRC denota "Clase-Responsabilidad-Colaboración", y es una técnica que reemplaza a los diagramas en la representación de modelos. En las tarjetas se escriben las Responsabilidades, una descripción de alto nivel del propósito de una clase y las dependencias primarias. En su economía sintáctica y en su abstracción, prefiguran a lo que más tarde serían las tarjetas de historias de XP. Beck y Cunningham prohibían escribir en las tarjetas más de lo que en ellas cabía [BC89]. Nos ha parecido útil ilustrar las tarjetas correspondientes al Modelo, la Vista y el Controlador del patrón MVC de Smalltalk, tal como lo han hecho los autores en el texto original. Hemos descripto ese patrón (o estilo) en el documento sobre Estilos en Arquitectura de Software.

Luego de esa tarea conjunta, Beck comenzó a desarrollar XP mientras Cunningham inventó el concepto de Wiki Web, el sistema de hipertexto de autoría colectiva antecesor

de los Weblogs. En la década siguiente, Beck fue empleado por Chrysler para colaborar en el sistema de liquidación de sueldos en Smalltalk que se conoció como el proyecto C3 (Chrysler Comprehensive Compensation). Más tarde se unieron al proyecto Ron Jeffries (luego ardiente partidario de LSD y del framework .NET de Microsoft) y Martin Fowler, ulterior líder de Cutter Consortium. Previsiblemente, terminaron desarrollando C3 en una especie de proto-XP y el proyecto terminó exitosamente. Hay una leyenda negra que afirma que aunque XP surgió de allí, el proyecto C3 fracasó [Lar04] [Kee03]; lo cierto es que estuvo en producción durante un tiempo, pero fue reemplazado por otro sistema y los equipos originales de programación y mantenimiento se disolvieron cuando Chrysler cambió de nombre y de dueños.

En la gestación de C3 se encuentra no sólo la raíz de XP, sino el núcleo primitivo de la comunidad ágil. Beck, Cunningham y Jeffries son llamados "*the three extremos*", por analogía con "*the three amigos*" de UML [Hig00b]. Muchos autores escriben "eXtreme Programming" para ser consistentes con la sigla, pero Kent Beck siempre lo mayusculiza como se debe. XP se funda en cuatro valores: comunicación, simplicidad, *feedback* y coraje. Pero tan conocidos como sus valores son sus prácticas. Beck sostiene que se trata más de lineamientos que de reglas:



Gestación de XP, basado en [ASR+02]

1. **Juego de Planeamiento**. Busca determinar rápidamente el alcance de la versión siguiente, combinando prioridades de negocio definidas por el cliente con las estimaciones técnicas de los programadores. Éstos estiman el esfuerzo necesario para implementar las historias del cliente y éste decide sobre el alcance y la agenda de las entregas. Las historias se escriben en pequeñas fichas, que algunas veces se tiran. Cuando esto sucede, lo único restante que se parece a un requerimiento es una multitud de pruebas automatizadas, las pruebas de aceptación.
2. **Entregas pequeñas y frecuentes**. Se "produccioniza" [sic] un pequeño sistema rápidamente, al menos uno cada dos o tres meses. Pueden liberarse nuevas versiones diariamente (como es práctica en Microsoft), pero al menos se debe liberar una cada mes. Se agregan pocos rasgos cada vez.
3. **Metáforas del sistema**. El sistema de define a través de una metáfora o un conjunto de metáforas, una "historia compartida" por clientes, *managers* y programadores que orienta todo el sistema describiendo como funciona. Una metáfora puede interpretarse como una arquitectura simplificada. La concepción de metáfora que se aplica en XP

deriva de los estudios de Lakoff y Johnson, bien conocidos en lingüística y psicología cognitiva.

4. **Diseño simple**. El énfasis se deposita en diseñar la solución más simple susceptible de implementarse en el momento. Se eliminan complejidades innecesarias y código extra, y se define la menor cantidad de clases posible. No debe duplicarse código. En un oxímoron deliberado, se urge a "decir todo una vez y una sola vez". Nadie en XP llega a prescribir que no haya diseño concreto, pero el diseño se limita a algunas tarjetas elaboradas en sesiones de diseño de 10 a 30 minutos. Esta es la práctica donde se impone el minimalismo de YAGNI: no implementar nada que no se necesite ahora; o bien, nunca implementar algo que vaya a necesitarse más adelante; minimizar diagramas y documentos.
5. **Prueba continua**. El desarrollo está orientado por las pruebas. Los clientes ayudan a escribir las pruebas funcionales *antes* que se escriba el código. Esto es *test-driven development*. El propósito del código real no es cumplir un requerimiento, sino pasar las pruebas. Las pruebas y el código son escritas por el mismo programador, pero la prueba debería realizarse sin intervención humana, y es a todo o nada. Hay dos clases de prueba: la prueba unitaria, que verifica una sola clase, o un pequeño conjunto de clases; la prueba de aceptación verifica todo el sistema, o una gran parte.
6. **Refactorízación[2] continua**. Se refactoriza el sistema eliminando duplicación, mejorando la comunicación y agregando flexibilidad sin cambiar la funcionalidad. El proceso consiste en una serie de pequeñas transformaciones que modifican la estructura interna preservando su conducta aparente. La práctica también se conoce como **Mejora Continua de Código** o **Refactorízación implacable**. Se lo ha parafraseado diciendo: "Si funciona bien, arréglelo de todos modos". Se recomiendan herramientas automáticas. En sus comentarios a [Hig00b] Ken Orr recomienda GeneXus de ARTech, Uruguay, virtuoso ejecutor de las mejores promesas incumplidas de las herramientas CASE.
7. **Programación en pares**. Todo el código está escrito por pares de programadores. Dos personas escriben código en una computadora, turnándose en el uso del ratón y el teclado. El que no está escribiendo, piensa desde un punto de vista más estratégico y realiza lo que podría llamarse revisión de código en tiempo real. Los roles pueden cambiarse varias veces al día. Esta práctica no es en absoluto nueva. Hay

---

[2] El término *refactoring*, introducido por W. F. Opdyke en su tesis doctoral [Opd92], se refiere "al proceso de cambiar un sistema de software [orientado a objetos] de tal manera que no se altere el comportamiento exterior del código, pero se mejore su estructura interna". Normalmente se utilizan herramientas automáticas para hacerlo (DMS, GeNexus), y/o se implementan técnicas tales como aserciones (invariantes, pre- y poscondiciones) para expresar propiedades que deben conservarse antes y después de la refactoría. Otras técnicas son transformaciones de grafos, métricas de software, refinamiento de programas y análisis formal de conceptos [MVD03]. Al igual que sucede con los patrones, existe un amplio catálogo de refactorizaciones más comunes: reemplazo de iteración por recursión; sustitución de un algoritmo por otro más claro; extracción de clase, interface o método; descomposición de condicionales; reemplazo de herencia por delegación, etcétera. La Biblia del método es *Refactoring* de Martin Fowler, Kent Beck, John Brant, William Opdyke y Don Roberts [FBB+99].

antecedentes de programación en pares anteriores a XP [Hig00b]. Steve McConnell la proponía en 1993 en su *Code Complete* [McC93: 528].

8. **Propiedad colectiva del código**. Cualquiera puede cambiar cualquier parte del código en cualquier momento, siempre que escriba antes la prueba correspondiente. Algunas veces los practicantes aplican el patrón organizacional *CodeOwnership* de Coplien [Cop95].
9. **Integración continua**. Cada pieza se integra a la base de código apenas está lista, varias veces al día. Debe correrse la prueba antes y después de la integración. Hay una máquina (solamente) dedicada a este propósito.
10. **Ritmo sostenible**, trabajando un máximo de 8 horas por día. Antes se llamaba a esta práctica **Semana de 40 horas**. Mientras en RAD las horas extras eran una *best practice* [McC96], en XP todo el mundo debe irse a casa a las cinco de la tarde. Dado que el desarrollo de software se considera un ejercicio creativo, se estima que hay que estar fresco y descansado para hacerlo eficientemente; con ello se motiva a los participantes, se evita la rotación del personal y se mejora la calidad del producto. Deben minimizarse los héroes y eliminar el "proceso neurótico". Aunque podrían admitirse excepciones, no se permiten dos semanas seguidas de tiempo adicional. Si esto sucede, se lo trata como problema a resolver.
11. **Todo el equipo en el mismo lugar**. El cliente debe estar presente y disponible a tiempo completo para el equipo. También se llama **El Cliente en el Sitio**. Como esto parecía no cumplirse (si el cliente era muy *junior* no servía para gran cosa, y si era muy *senior* no deseaba estar allí), se especificó que el representante del cliente debe ser preferentemente un analista. (Tampoco se aclara analista de qué; seguramente se definirá en una próxima versión).
12. **Estándares de codificación**. Se deben seguir reglas de codificación y comunicarse a través del código. Según las discusiones en Wiki, algunos practicantes se desconciertan con esta regla, prefiriendo recurrir a la tradición oral. Otros la resuelven poniéndose de acuerdo en estilos de notación, indentación y nomenclatura, así como en un valor apreciado en la práctica, el llamado "código revelador de intenciones". Como en XP rige un cierto purismo de codificación, los comentarios no son bien vistos. Si el código es tan oscuro que necesita comentario, se lo debe reescribir o refactorizar.
13. **Espacio abierto**. Es preferible una sala grande con pequeños cubículos o, mejor todavía, sin divisiones. Los pares de programadores deben estar en el centro. En la periferia se ubican las máquinas privadas. En un encuentro de espacio abierto la agenda no se establece verticalmente.
14. **Reglas justas**. El equipo tiene sus propias reglas a seguir, pero se pueden cambiar en cualquier momento. En XP se piensa que no existe un proceso que sirva para todos los proyectos; lo que se hace habitualmente es adaptar un conjunto de prácticas simples a la características de cada proyecto.

Las prácticas se han ido modificando con el tiempo. Originariamente eran doce; de inmediato, trece. Las versiones más recientes enumeran diecinueve prácticas, agrupadas en cuatro clases.

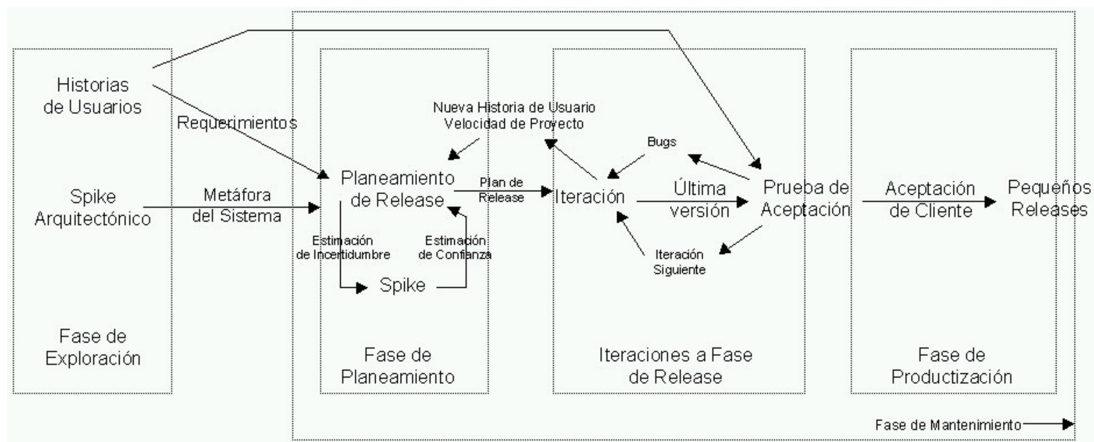
Prácticas conjuntas	Iteraciones Vocabulario Común – Reemplaza a Metáforas Espacio de trabajo abierto Retrospectivas
Prácticas de Programador	Desarrollo orientado a pruebas Programación en pares Refactorización Propiedad colectiva Integración continua YAGNI ("No habrás de necesitarlo") – Equivale a Diseño Simple
Prácticas de Management	Responsabilidad aceptada Cobertura aérea para el equipo Revisión trimestral Espejo – El *manager* debe comunicar un fiel reflejo del estado de cosas Ritmo sostenible
Prácticas de Cliente	Narración de historias Planeamiento de entrega Prueba de aceptación Entregas frecuentes

Mientras esto se escribe, Ward Cunningham sugiere rebautizar las prácticas como Xtudios (*Xtudes*). Oficialmente las prácticas siguen siendo doce y su extensión se basa en documentos inéditos de Kent Beck. Las prácticas no tratan en detalle el proceso de entrega, que es casi un no-evento.

Al lado de los valores y las prácticas, XP ha abundado en acrónimos algo extravagantes pero eficaces, que sus seguidores intercambian con actitud de complicidad. YAGNI significa "You Aren't Gonna Need It"; TETCPB, "Test Everything That Can Possibly Broke"; DTSTTCPW, "Do The Simplest Thing That Can Possibly Work", etcétera. Del mismo modo, BUFD ("Big Up Front Design") es el mote peyorativo que se aplica a los grandes diseños preliminares de los métodos en cascada, "el Libro Verde" es *Planning Extreme Programming* de Kent Beck y Martin Fowler [BF00], y "el Libro Blanco" es *Extreme Programming Explained* de Kent Beck [Beck99a]. Todo el mundo en XP sabe, además, qué significan GoF, *PLOPD* o *POSA*. Al menos en la concepción de Cunningham y Wiki, parecería que quien conoce más mantras y acrónimos, gana.

Beck [Beck99a] sugiere adoptar XP paulatinamente: "Si quiere intentar con XP, por Dios le pido que no se lo trague todo junto. Tome el peor problema de su proceso actual y trate de resolverlo a la manera de XP". Las prácticas también pueden adaptarse a las características de los sistemas particulares. Pero no todo es negociable y la integridad del método se sabe frágil. La particularidad de XP radica en no requerir ninguna herramienta fuera del ambiente de programación y prueba. Al contrario de otros métodos, que permiten modelado, XP demanda comunicación oral tanto para los requerimientos como para el diseño. Beck [Beck99b] ha llegado a decir: "También yo creo en el modelado; sólo que lo llamo por su nombre propio, 'mentir', y trato de convertirlo en arte".

El ciclo de vida es, naturalmente, iterativo. El siguiente diagrama describe su cuerpo principal:



Ciclo de vida de XP, adaptado de [Beck99a]

Algunos autores sugieren implementar *spikes* (o sea "púas" o "astillas") para estimar la duración y dificultad de una tarea inmediata [JAH01]. Un *spike* es un experimento dinámico de código, realizado para determinar cómo podría resolverse un problema. Es la versión ágil de la idea de prototipo. Se lo llama así porque "va de punta a punta, pero es muy fino" y porque en el recorrido de un árbol de opciones implementaría una opción de búsqueda *depth-first* (http://c2.com/cgi/wiki?SpikeSolution).

Entre los artefactos que se utilizan en XP vale la pena mencionar las tarjetas de historias (*story cards*); son tarjetas comunes de papel en que se escriben breves requerimientos de un rasgo, jamás casos de uso; pueden adoptar el esquema CRC. Las tarjetas tienen una granularidad de diez o veinte días. Las tarjetas se usan para estimar prioridades, alcance y tiempo de realización; en caso de discrepancia, gana la estimación más optimista. Otros productos son listas de tareas en papel o en una pizarra (jamás en computadora) y gráficos visibles pegados en la pared. Martin Fowler admite que la preferencia por esta clase de herramientas ocasiona que el mensaje entre líneas sea "Los XPertos no hacen diagramas" [Fow01].

Los roles de XP son pocos. Un cliente que escribe las historias y las pruebas de aceptación; programadores en pares; verificadores (que ayudan al cliente a desarrollar las pruebas); consultores técnicos; y, como parte del *management*, un *coach* o consejero que es la conciencia del grupo, interviene y enseña, y un seguidor de rastros (*tracker*) que colecta las métricas y avisa cuando hay una estimación alarmante, además de un Gran Jefe. El *management* es la parte menos satisfactoriamente caracterizada en la bibliografía, como si fuera un mal necesario; Beck comenta, por ejemplo, que la función más relevante del *coach* es la adquisición de juguetes y comida [Hig00b]; otros dicen que está para servir café. Lo importante es que el coach se vea como un facilitador, antes que como quien dá las órdenes. Los equipos de XP son típicamente pequeños, de tres a veinte personas, y en general no se cree que su escala se avenga al desarrollo de grandes sistemas de misión crítica con tanta comodidad como FDD.

Algunas empresas han creado sus propias versiones de XP, como es el caso de Standard & Poor's, que ha elaborado plantillas para proyectos futuros. XP se ha combinado con Evo, a pesar que ambos difieren en su política de especificaciones; también se estima compatible con Scrum, al punto que existe una forma híbrida, inventada por Mike Beedle, que se llama XBreed [http://www.xbreed.net] y otra bautizada XP@Scrum

[http://www.controlchaos.com/xpScrum.htm] en la que Scrum se usa como envoltorio de gestión alrededor de prácticas de ingeniería de XP. Se lo ha combinado muchas veces con UP, aunque éste recomienda pasar medio día de discusión de pizarra antes de programar y XP no admite más de 10 o 20 minutos. La diferencia mayor concierne a los casos de uso, que son norma en UP y que en XP se reemplazan por tarjetas de papel con historias simples que se refieren a rasgos. En las herramientas de RUP ahora hay *plug-ins* para XP.

Los obstáculos más comunes surgidos en proyectos XP son la "fantasía" [Lar04] de pretender que el cliente se quede en el sitio y la resistencia de muchos programadores a trabajar en pares. Craig Larman señala como factores negativos la ausencia de énfasis en la arquitectura durante las primeras iteraciones (no hay arquitectos en XP) y la consiguiente falta de métodos de diseño arquitectónico. Un estudio de casos de Bernhard Rumpe y Astrid Schröder sobre 45 ejemplares de uso reveló que las prácticas menos satisfactorias de XP han sido la presencia del usuario en el lugar de ejecución y las metáforas, que el 40% de los encuestados no comprende para qué sirven o cómo usarlas [RS01]. Las metáforas han sido reemplazadas por un Vocabulario Común (equivalente al "Sistema de Nombres" de Cunningham) en las últimas revisiones del método.

XP ha sido, de todos los MAs, el que más resistencia ha encontrado [Rak01] [McB02] [Baer03] [Mel03] [SR03]. Algunos han sugerido que esa beligerancia es un efecto de su nombre, que debería ser menos intimidante. Jim Highsmith [Hig02a] argumenta, sin embargo, que es improbable que muchos se entusiasmen por un libro que se titule, por ejemplo, *Programación Moderada*. Los nuevos mercados, tecnologías e ideas –piensa Highsmith– no se forjan a fuerza de moderación sino con giros radicales y con el coraje necesario para desafiar al status quo; XP, en todo caso, ha abierto el camino.

Scrum

Esta es, después de XP, la metodología ágil mejor conocida y la que otros métodos ágiles recomiendan como complemento, aunque su porción del mercado (3% según el Cutter Consortium) es más modesta que el ruido que hace. La primera referencia a la palabra "*scrum*" en la literatura aparece en un artículo de Hirotaka Takeuchi e Ikujiro Nonaka, "The New Product Development Game" en el que se presentaron diversas *best practices* de empresas innovadoras de Japón que siempre resultaban ser adaptativas, rápidas y con capacidad de auto-organización [TN86].

Otra palabra del mismo texto relacionada con modelos japoneses es *Sashimi*, el cual se inspira en una estrategia japonesa de desarrollo de hardware con fases solapadas utilizada por Fuji-Xerox. La palabra Scrum, empero, nada tiene que ver con Japón, sino que procede de la terminología del juego de rugby, donde designa al acto de preparar el avance del equipo en unidad pasando la pelota a uno y otro jugador (aunque hay otras acepciones en circulación). Igual que el juego, Scrum es adaptativo, ágil, auto-organizante y con pocos tiempos muertos.

Como metodología ágil específicamente referida a ingeniería de software, Scrum fue aplicado por Jeff Sutherland y elaborado más formalizadamente por Ken Schwaber [Sch95]. Poco después Sutherland y Schwaber se unieron para refinar y extender Scrum [BDS+98] [SB02]. En Scrum se aplicaron principios de procesos de control industrial, junto con experiencias metodológicas de Microsoft, Borland y Hewlett-Packard.

Schwaber, en particular, había trabajado con científicos de Du Pont para comprender mejor los procesos definidos de antemano, y ellos le dijeron que a pesar que CMM se concentraba en hacer que los procesos de desarrollo se tornaran repetibles, definidos y predecibles, muchos de ellos eran formalmente impredecibles e irrepetibles porque cuando se está planificando no hay primeros principios aplicables, los procesos recién comienzan a ser comprendidos y son complejos por naturaleza. Schwaber se dió cuenta entonces de que un proceso necesita aceptar el cambio, en lugar de esperar predictibilidad.

Al igual que Agile Modeling, Scrum no está concebido como método independiente, sino que se promueve como complemento de otras metodologías, incluyendo XP, MSF o RUP. Como método, Scrum enfatiza valores y prácticas de gestión, sin pronunciarse sobre requerimientos, implementación y demás cuestiones técnicas; de allí su deliberada insuficiencia y su complementariedad. Scrum se define como un proceso de *management* y control que implementa técnicas de control de procesos; se lo puede considerar un conjunto de patrones organizacionales, como se verá en un capítulo posterior (p. 60).

Los valores de Scrum son:

- Equipos auto-dirigidos y auto-organizados. No hay *manager* que decida, ni otros títulos que "miembros del equipo" o "cerdos"; la excepción es el *Scrum Master* que debe ser 50% programador y que resuelve problemas, pero no manda. Los observadores externos se llaman "gallinas"; pueden observar, pero no interferir ni opinar.
- Una vez elegida una tarea, no se agrega trabajo extra. En caso que se agregue algo, se recomienda quitar alguna otra cosa.
- Encuentros diarios con las tres preguntas indicadas en la figura. Se realizan siempre en el mismo lugar, en círculo. El encuentro diario impide caer en el dilema señalado por Fred Brooks: "¿Cómo es que un proyecto puede atrasarse un año?: Un día a la vez" [Bro95].
- Iteraciones de treinta días; se admite que sean más frecuentes.
- Demostración a participantes externos al fin de cada iteración.
- Al principio de cada iteración, planeamiento adaptativo guiado por el cliente.

El nombre de los miembros del equipo y los externos se deriva de una típica fábula agilista: un cerdo y una gallina discutían qué nombre ponerle a un nuevo restaurant; la gallina propuso "Jamón y Huevos". "No, gracias", respondió el cerdo, "Yo estaré comprometido, pero tú sólo involucrada".

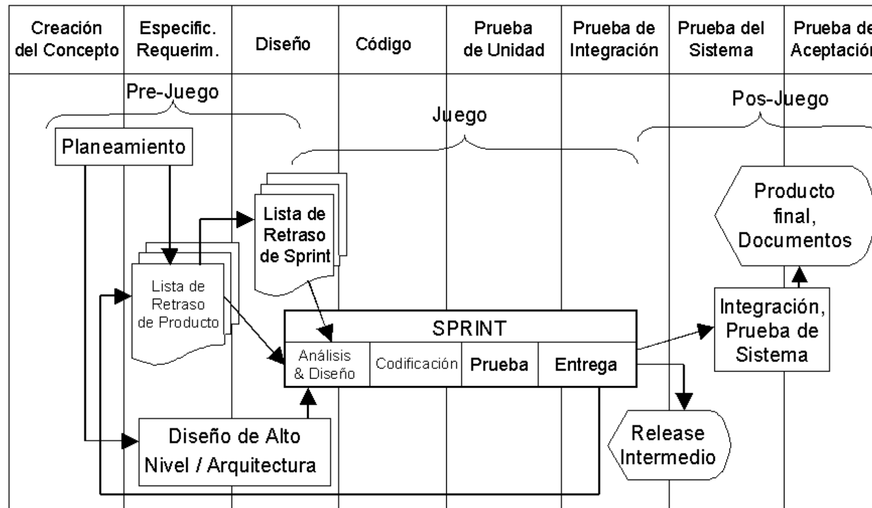
Scrum define seis roles:

1. **El Scrum Master**. Interactúa con el cliente y el equipo. Es responsable de asegurarse que el proyecto se lleve a cabo de acuerdo con las prácticas, valores y reglas de Scrum y que progrese según lo previsto. Coordina los encuentros diarios, formula las tres preguntas canónicas y se encarga de eliminar eventuales obstáculos. Debe ser miembro del equipo y trabajar a la par.
2. **Propietario del Proyecto**. Es el responsable oficial del proyecto, gestión, control y visibilidad de la lista de acumulación o lista de retraso del producto (*product backlog*). Es elegido por el Scrum Master, el cliente y los ejecutivos a cargo. Toma

las decisiones finales de las tareas asignadas al registro y convierte sus elementos en rasgos a desarrollar.

3. **Equipo de Scrum**. Tiene autoridad para reorganizarse y definir las acciones necesarias o sugerir remoción de impedimentos. El equipo posee la misma estructura del “equipo quirúrgico” desarrollado por IBM y comentado en el *MMM* de Brooks [Bro75], aunque Schwaber y Beedle [SB02] destacan que su naturaleza auto-organizadora la hace distinta.
4. **Cliente**. Participa en las tareas relacionadas con los ítems del registro.
5. **Management**. Está a cargo de las decisiones fundamentales y participa en la definición de los objetivos y requerimientos. Por ejemplo, selecciona al Dueño del Producto, evalúa el progreso y reduce el registro de acumulación junto con el Scrum Master.
6. **Usuario**.

La dimensión del equipo total de Scrum no debería ser superior a diez ingenieros. El número ideal es siete, más o menos dos, una cifra canónica en ciencia cognitiva [Mil56]. Si hay más, lo más recomendable es formar varios equipos. No hay una técnica oficial para coordinar equipos múltiples, pero se han documentado experiencias de hasta 800 miembros, divididos en Scrums de Scrum, definiendo un equipo central que se encarga de la coordinación, las pruebas cruzadas y la rotación de los miembros. El texto que relata esa experiencia es *Agile Software Development Ecosystems*, de Jim Highsmith [Hig02b].



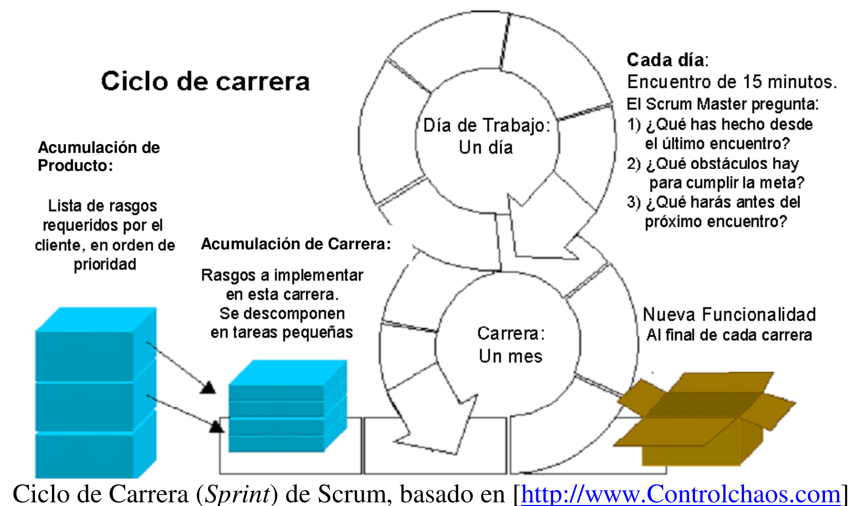
Ciclo de Scrum, basado en [Abr02]

El ciclo de vida de Scrum es el siguiente:

1. **Pre-Juego: Planeamiento**. El propósito es establecer la visión, definir expectativas y asegurarse la financiación. Las actividades son la escritura de la visión, el presupuesto, el registro de acumulación o retraso (*backlog*) del producto inicial y los ítems estimados, así como la arquitectura de alto nivel, el diseño exploratorio y los prototipos. El registro de acumulación es de alto nivel de abstracción.

2. **Pre-Juego: Montaje** (*Staging*). El propósito es identificar más requerimientos y priorizar las tareas para la primera iteración. Las actividades son planificación, diseño exploratorio y prototipos.
3. **Juego o Desarrollo**. El propósito es implementar un sistema listo para entrega en una serie de iteraciones de treinta días llamadas "corridas" (*sprints*). Las actividades son un encuentro de planeamiento de corridas en cada iteración, la definición del registro de acumulación de corridas y los estimados, y encuentros diarios de Scrum.
4. **Pos-Juego: Liberación**. El propósito es el despliegue operacional. Las actividades, documentación, entrenamiento, mercadeo y venta

Usualmente los registros de acumulación se llevan en planillas de cálculo comunes, antes que en una herramienta sofisticada de gestión de proyectos. Los elementos del registro pueden ser prestaciones del software, funciones, corrección de *bugs*, mejoras requeridas y actualizaciones de tecnología. Hay un registro total del producto y otro específico para cada corrida de 30 días. En la jerga de Scrum se llaman "paquetes" a los objetos o componentes que necesitan cambiarse en la siguiente iteración.



Ciclo de Carrera (*Sprint*) de Scrum, basado en [http://www.Controlchaos.com]

La lista de Acumulación del Producto contiene todos los rasgos, tecnología, mejoras y lista de *bugs* que, a medida que se desenvuelven, constituyen las futuras entregas del producto. Los rasgos más urgentes merecerán mayor detalle, los que pueden esperar se tratarán de manera más sumaria. La lista se origina a partir de una variedad de fuentes. El grupo de mercadeo genera los rasgos y la función; la gente de ventas genera elementos que harán que el producto sea más competitivo; los de ingeniería aportarán paquetes que lo harán más robusto; el cliente ingresará debilidades o problemas que deberán resolverse.

El propietario de la administración y el control del *backlog* en productos comerciales bien puede ser el *product manager*; para desarrollos *in-house* podría ser el *project manager*, o alguien designado por él. Se recomienda que una sola persona defina la prioridad de una tarea; si alguien tiene otra opinión, deberá convencer al responsable. Se estima que priorizar adecuadamente una lista de producto puede resultar dificultoso al principio del desarrollo, pero deviene más fácil con el correr del tiempo.

Acumulación de Producto:		Fecha:
		Estimado:
Tipo: Nuevo __ Mejora __ Arreglo: __	Fuente:	
Descripción		
Notas		

Product backlog de Scrum, basado en [http://www.controlchaos.com/pbacklog.htm]

La lista de acumulación de corrida sugerida tiene este formato:

Acumulación de Corrida:	Fecha:
Propietario:	Trabajo Pendiente/Fecha
Status: Pendiente___ Activo___Completo___	
Descripción:	
Notas:	

Sprint backlog de Scrum, basado en [http://www.controlchaos.com/sbacklog.htm]

El registro incluye los valores que representan las horas de trabajo pendiente; en función de esos valores se acostumbra elaborar un gráfico de quemado, cuyo modelo y fundamentación se encuentra en [http://www.controlchaos.com/burndown.htm]. Los gráficos de quemado, usados también en Crystal Methods, se describen en la página 33.

Al fin de cada iteración de treinta días hay una demostración a cargo del Scrum Master. Las presentaciones en PowerPoint están prohibidas. En los encuentros diarios, las gallinas deben estar fuera del círculo. Todos tienen que ser puntuales; si alguien llega tarde, se le cobra una multa que se destinará a obras de caridad. Es permitido usar artefactos de los métodos a los que Scrum acompañe, por ejemplo Listas de Riesgos si se utiliza UP, Planguage si el método es Evo, o los Planes de Proyecto sugeridos en la disciplina de Gestión de Proyectos de Microsoft Solutions Framework [MS02b]. No es legal, en cambio, el uso de instrumentos tales como diagramas PERT, porque éstos parten del supuesto de que las tareas de un proyecto se pueden identificar y ordenar; en Scrum el supuesto dominante es que el desarrollo es semi-caótico, cambiante y tiene demasiado ruido como para que se le aplique un proceso definido.

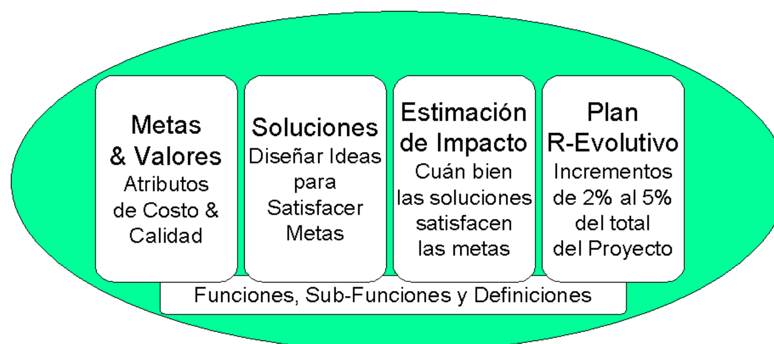
Algunos textos sobre Scrum establecen una arquitectura global en la fase de pre-juego; otros dicen que no hay una arquitectura global en Scrum, sino que la arquitectura y el diseño emanan de múltiples corridas [SB02]. No hay una ingeniería del software prescripta para Scrum; cada quien puede escoger entonces las prácticas de automatización, inspección de código, prueba unitaria, análisis o programación en pares que le resulten adecuadas.

Es habitual que Scrum se complemente con XP; en estos casos, Scrum suministra un marco de *management* basado en patrones organizacionales, mientras XP constituye la práctica de programación, usualmente orientada a objetos y con fuerte uso de patrones de diseño. Uno de los nombres que se utiliza para esta alianza es XP@Scrum. También son viables los híbridos con otros MAs.

Evolutionary Project Management (Evo)

Evo, creado por Tom Gilb, es el método iterativo ágil más antiguo. Se lo llama también Evolutionary Delivery, Evolutionary Management, Requirements Driven Project Management y Competitive Engineering. Fue elaborado inicialmente en Europa. En 1976 Gilb trató temas de desarrollo iterativo y gestión evolutiva en su clásico *Software metrics* [Gilb76], el texto que acuñó el concepto e inauguró el campo de las métricas de software. Luego desarrolló en profundidad esos temas en una serie de columnas en *Computer Weekly UK*. En 1981 publicó "Evolutionary Development" en *ACM Software Engineering Notes* y "Evolutionary Devivery versus the 'Waterfall Model'" en *ACM Sigsoft Software Requirements Engineering Notes*.

En la década de 1980 Gilb cayó bajo la influencia de los valores de W. Edward Deming y del método Planear-Hacer-Estudiar-Actuar (PDSA) de Walter Shewhart, que se constituyeron en modelos conceptuales subyacentes a Evo. Deming y Schewhart son, incidentalmente, considerados los padres del control estadístico de calidad; sus ideas, desarrolladas en las décadas de 1940 y 1950, se han aprovechado, por ejemplo, en la elaboración de estrategias como Six Sigma, en Lean Development o en la industria japonesa. En los 90s Gilb continuó el desarrollo de Evo, que tal vez fue más influyente por las ideas que éste proporcionara a XP, Scrum e incluso UP que por su éxito como método particular. El texto de Microsoft Press *Rapid Development* de Steve McConnell [McC96], que examina prácticas iterativas e incrementales en desarrollo de software, menciona ideas de Gilb en 14 secciones [Lar04]; todas esas referencias están ligadas a *best practices*.



Elementos de Evo [Gilb03b]

En las breves iteraciones de Evo, se efectúa un progreso hacia las máximas prioridades definidas por el cliente, liberando algunas piezas útiles para algunos participantes y solicitando su *feedback*. Esta es la práctica que se ha llamado Planeamiento Adaptativo Orientado al Cliente y Entrega Evolutiva. Otra idea distintiva de Evo es la clara definición, cuantificación, estimación y medida de los requerimientos de performance que necesitan mejoras. La performance incluye requisitos de calidad tales como robustez y tolerancia a fallas, al lado de estipulaciones cuantitativas de capacidad de carga y de ahorro de recursos. En Evo se espera que cada iteración constituya una re-evaluación de las soluciones en procura de la más alta relación de valor contra costo, teniendo en cuenta tanto el *feedback* como un amplio conjunto de estimaciones métricas. Evo requiere, igual que otros MAs, activa participación de los clientes. Todo debe cuantificarse; se

desalientan las apreciaciones cualitativas o subjetivas como “usable”, “mantenible” o “ergonómico”. A diferencia de otros MAs como Agile Modeling, donde la metodología es puntillosa pero discursiva, en Evo hay una especificación semántica y una pragmática rigurosa, completamente alejadas del sentido común, pero con la fundamentación que les presta derivarse de prácticas productivas suficientemente probadas.

Los diez principios fundamentales de Evo son:

1. Se entregarán temprano y con frecuencia resultados verdaderos, de valor para los participantes reales.
2. El siguiente paso de entrega de Evo será el que proporcione el mayor valor para el participante en ese momento.
3. Los pasos de Evo entregan los requerimientos especificados de manera evolutiva.
4. No podemos saber cuáles son los requerimientos por anticipado, pero podemos descubrirlos más rápidamente intentando proporcionar valor real a participantes reales.
5. Evo es ingeniería de sistemas holística (todos los aspectos necesarios del sistema deben ser completos y correctos) y con entrega a un ambiente de participantes reales (no es sólo sobre programación; es sobre satisfacción del cliente).
6. Los proyectos de Evo requieren una arquitectura abierta, porque habremos de cambiar las ideas del proyecto tan a menudo como se necesite hacerlo, para entregar realmente valor a nuestros participantes.
7. El equipo de proyecto de Evo concentrará su energía como equipo hacia el éxito del paso actual. En este paso tendrán éxito o fracasarán todos juntos. No gastarán energías en pasos futuros hasta que hayan dominado los pasos actuales satisfactoriamente.
8. Evo tiene que ver con aprendizaje a partir de la dura experiencia, tan rápido como se pueda: qué es lo que verdaderamente funciona, qué es lo que realmente entrega valor. Evo es una disciplina que nos hace confrontar nuestros problemas tempranamente, pero que nos permite progresar rápido cuando probadamente lo hemos hecho bien.
9. Evo conduce a una entrega temprana, a tiempo, tanto porque se lo ha priorizado así desde el inicio, y porque aprendemos desde el principio a hacer las cosas bien.
10. Evo debería permitirnos poner a prueba nuevos procesos de trabajo y deshacernos tempranamente de los que funcionan mal.

El modelo de Evo consiste en cinco elementos mayores [Gilb03b]:

1. **Metas, Valores y Costos** – Cuánto y cuántos recursos. Las Metas y Valores de los Participantes se llaman también, según la cultura, objetivos, metas estratégicas, requerimientos, propósitos, fines, ambiciones, cualidades e intenciones.
2. **Soluciones** – Banco de ideas sobre la forma de alcanzar Metas y Valores dentro del rango de los Costos.
3. **Estimación de Impacto** – Mapear las Soluciones a Metas y Costos para averiguar si se tienen ideas adecuadas para lograr las Metas dentro de los Costos.

4. **Plan Evolutivo** – Inicialmente una idea general de la secuencia a desarrollar y evolucionar hacia las Metas. Los detalles necesarios evolucionan junto con el resto del plan a medida que se desarrolla el producto/servicio.
5. **Funciones** – Describen qué hace el sistema. Son extremadamente secundarias, más de lo que se piensa, y deben mantenerse al mínimo.

A diferencia de lo que es el caso en IEEE 1471, donde todos, clientes y técnicos, son Participantes (*Stakeholders*), en Evo se llama Participante sólo al cliente. Cuando se inicia el ciclo, primero se definen los Valores y Metas del Participante; esta es una lista tradicional de recursos tales como dinero, tiempo y gente. Una vez que se comprende hacia dónde se quiere ir y cuándo se podría llegar ahí, se definen Soluciones para lograrlo. Utilizando una Tabla de Estimación de Impacto, se realiza la ingeniería de las Soluciones para satisfacer óptimamente las Metas y Valores de los Participantes. Se desarrolla un plan paso a paso llamado Entrega Evolutiva para entregar no soluciones, sino mejoras a dichas Metas y Valores. Inicialmente las Soluciones y el Plan de Entrega Evolutiva se delinean a un alto nivel de abstracción. Tomando ideas de las Soluciones y del Plan se detallan, desarrollan, verifican y entregan a los Participantes reales o a quién se encuentre tan cerca de ellos como se pueda llegar.

A medida que se desenvuelve el proyecto, se obtiene *feedback* en tiempo real sobre las mejoras que implica la Entrega Evolutiva sobre las Metas y Valores del Participante, así como sobre el consumo de Recursos. Esta información se usa para establecer qué es lo que está bien y lo que no, cuáles son los desafíos y qué es lo que no se sabía desde un principio; también se aprende sobre las nuevas tecnologías y técnicas que no estaban disponibles cuando el proyecto empezó. Se ajusta luego todo según se necesite, pero sin detallar las Soluciones o las Entregas Evolutivas hasta que se esté próximo a la entrega. Por último vienen las Funciones y Sub-Funciones, de las que se razona teniendo en cuenta que en rigor, en un nivel puro, son de hecho Soluciones a Metas.

El hijo de Tom, Kai Gilb, ha sido crítico sobre la necesidad de comprender bien las metas y no mezclarlas con otra cosa. Bajo el nombre de requerimientos se suelen mezclar soluciones y finalidades sin que nadie se sienta culpable. En Evo, cada una de las categorías ha sido objeto de un cuidadoso análisis semántico que busca establecer, además, por qué la gente suele comunicarse sobre el "Cómo" de una solución y no sobre "Cuán bien". En este marco, esa clase de distinciones (demasiado escrupulosas para detallarlas aquí) suele ser esencial. En esa lectura, por ejemplo, una Meta es un nivel de Calidad que se ha decidido alcanzar; como en inglés hay confusión entre calidad y cualidad (que se designan ambos con la misma palabra) Gilb luego dice que cada producto tiene muchas cualidades, y las considera atributos del sistema. Cuando se desarrolla un sistema, entonces, se decide qué niveles de calidad desearíamos que tuviera el sistema, y se las llama Metas.

Igual tratamiento se concede en Evo a las Funciones y Sub-Funciones, que se conciben más bien como Soluciones para Metas de los Participantes. Aunque hay un proceso bien definido para establecerlas y tratar con ellas, en Evo se recomienda mantener las funciones al mínimo, porque se estima además que una especificación funcional es una concepción obsoleta. La diferencia entre un procesador de texto como Word y la escritura con lápiz y papel, ilustra Gilb, *no* es funcional; en ambos casos se puede hacer lo mismo, sólo que se lo hace de distinta manera. Lo que el Participante requiere no debe ser

interpretado como la adquisición de nueva funcionalidad; por el contrario, el desastre de la industria de software, se razona aquí, se origina en que ha dependido demasiado de la funcionalidad. En Evo, sin duda, se debe razonar de otro modo: todo tiene que ver con las Metas y Valores de los Participantes, expresadas en términos tales que una Solución (o como se la llama también Diseño, Estrategia, Táctica, Proceso, Funcionalidad, Arquitectura y Método) pueda definirse como un medio para un fin, a través del cual se lleve de la situación en la que se está a otra situación que se desea.

Una herramienta desarrollada finamente en Evo es la que se llama "Herramienta-?"; consiste en preguntar "por qué" a cada meta o requerimiento aparente, haciéndolo además iterativamente sobre las respuestas que se van dando, a fin de deslindar el requisito real por detrás de la fachada de la contestación inicial. Unos pocos ejemplos demuestran la impensada eficacia de un principio que a primera vista parecería ser circular [Gilb03b]. Tras presentar otras herramientas de notación y detallar sus usos, llega la hora de poner en marcha un recurso fundamental de Evo, que es el Planguage. Se trata de un lenguaje estructurado de especificación que sirve para formalizar el requerimiento. El lenguaje es simple pero rico, así como son persuasivas las pautas que orientan su uso. Este es un ejemplo sencillo de la especificación de una meta de satisfacción del cliente en Planguage:

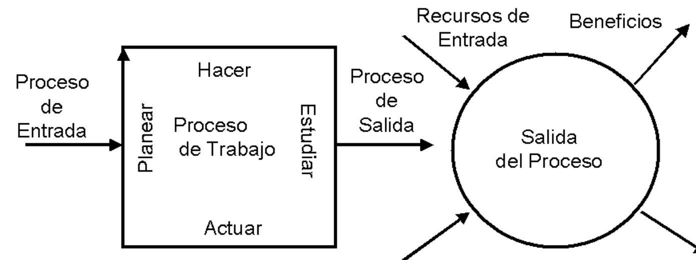
```
CUSTOMER.SATISFACTION
SCALE: evaluación promedio de la satisfacción de un cliente, de 1 a 5,
       siendo 1 la peor y 5 la mejor
PAST [2003] 2.5
GOAL [2004] 3.5
```

Planguage integra todas las disciplinas de solución de problemas. Mediante él, y usando el Método Planguage (PL), se pueden sujetar los medios a los fines. El método consiste en un lenguaje para Planificación y un conjunto de procesos de trabajo, el Lenguaje de Procesos. Proporciona un conjunto rico de formas de expresar propósito, restricciones, estrategia, diseño, bondad, inadecuación, riesgo, logro y credibilidad. Se inspira ampliamente en el ciclo Planear-Hacer-Estudiar-Actuar (PDSA) que Walter Shewhart aplicara desde 1930 y su discípulo Walter Deming impusiera en Japón.

En [Gilb03b] Planguage es elaborado y enseñado a medida que se describe el desarrollo de la metodología. En la descripción de los pasos de una solución, Gilb va cuestionando idea por idea el concepto tradicional de método, modelo, diseño y solución, así como las herramientas que se suponen inevitablemente ligadas a ellos. Un tratamiento crítico ejemplar merece, por ejemplo, el modelo en cascada, cuya idea subyacente de "flujo" atraviesa incluso a los modelos que creen basarse en una concepción distinta. Gilb lo ilustra con una caso real: "Los ingenieros de Microsoft no usan su propia herramienta tipo PERT [MS Project], porque éstas se basan en el modelo en cascada; no pueden hacerlo, porque ellos tienen proyectos reales, con desafíos, incógnitas, requerimientos cambiantes, nuevas tecnologías, competidores, etcétera. Ellos usan métodos que proporcionan *feedback* y aceptan cambios" [Gilb03b: 72].

Esencial en el tratamiento metodológico de Evo es la concepción que liga la evolución con el aprendizaje, lo que por otra parte es consistente con las modernas teorías evolutivas de John Holland [Hol95] y con las teorías que rigen los sistemas adaptativos complejos. Los ciclos de aprendizaje de Evo son exactamente lo mismo que el ciclo

Planear-Hacer-Estudiar-Actuar. Los principios que orientan la idea de la Entrega Evolutiva del Proyecto son:



Conceptos del Método Planguage, basado en [Gilb03a]

1. **Aprendizaje**: La Entrega Evolutiva es un ciclo de aprendizaje. Se aprende de la realidad y la experiencia; se aprende lo que funciona y lo que no.
2. **Temprano**: Aprender lo suficiente para cambiar lo que necesita cambiarse antes que sea tarde.
3. **Pequeño**: Pequeños pasos acarrean pequeños riesgos. Manteniendo el ciclo de entrega breve, es poco lo que se pierde cuando algo anda mal.
4. **Más simple**: Lo complejo se hace más simple y más fácil de manejar cuando se lo descompone en pequeñas partes.

El carácter de aprendizaje que tiene el modelo de ciclos es evidente en el ejemplo:

Plan a: Hacer un plan de alto nivel.

Plan b: Descomponer ese plan en incrementos. Seleccionar un incremento a realizar primero.

Hacer: Hacer el primer incremento. Entregarlo a los Participantes en un breve período de tiempo.

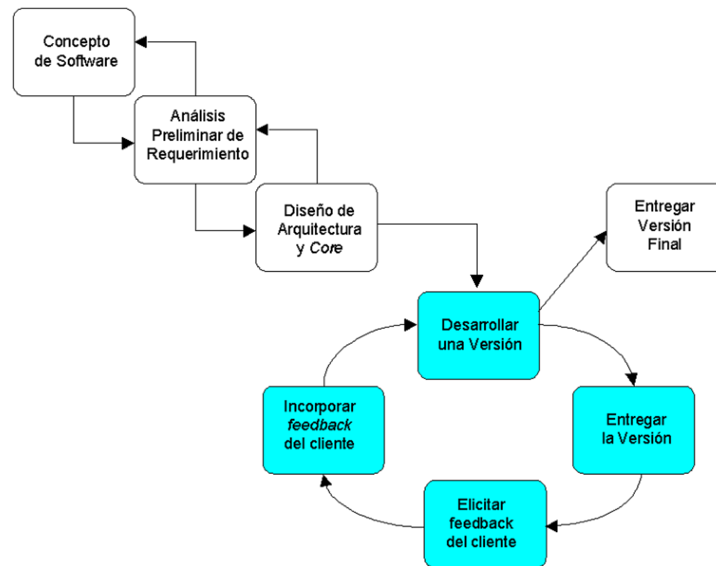
Estudio: Mediar y estudiar cuán bien se comportó el incremento, comparándolo con las expectativas y aprendiendo de la experiencia.

Actuar: Basado en lo que se aprendió, mantener el incremento o desecharlo, o introducir los cambios necesarios.

Otro marco que ha establecido la analogía entre evolución (o adaptación) y aprendizaje es, como se verá, el Adaptive Software Development de Jim Highsmith [Hig00a].

La evaluación que se realiza en el estudio debe apoyarse en una herramienta objetiva de inspección o control de calidad (un tema muy amplio de ingeniería de software); Evo implementa para ello Specification Quality Control (SQC), un método probado durante décadas, capaz de descubrir fallas que otros métodos pasan por alto. Se estima que SQC puede captar el 95% de las fallas y reducir tiempos de proyecto en un 50% [Gilb03a]; el uso de SQC, así como su rendimiento, está documentado en docenas de proyectos de misión crítica. Si bien SQC como lenguaje de especificación es complejo, como se trata de un estándar de industria existen herramientas automatizadas de alto nivel, como SQC for Excel™, un *add*-in desarrollado por BaRaN Systems (http://www.baran-systems.com/New/Products/SQC_For_Excel/index.htm), el *shareware* Premium SQC for Excel, SPC for MS Excel de Business Process Improvement (BPI) y otros productos similares.

Tom Gilb distingue claramente entre los Pasos de la Entrega Evolutiva y las iteraciones propias de los modelos iterativos tradicionales o de algún método ágil como RUP. Las iteraciones siguen un modelo de flujo, tienen un diseño prestablecido y son una secuencia de construcciones definidas desde el principio; los pasos evolutivos se definen primero de una manera genérica y luego se van refinando en cada ciclo, adoptando un carácter cada vez más formal. Las iteraciones no se realizan sólo para corregir los errores de código mediante refinamiento convencional, sino en función de la aplicación de SQC u otras herramientas con capacidades métricas.



Modelo de entrega evolutiva – Basado en [McC96]

En proyectos evolutivos, las Metas se desarrollan tratando de comprender de quiénes vienen (Participantes), qué es lo que son (medios y fines) y cómo expresarlas (cuantificables, medibles y verificables). Se procura pasar el menor tiempo posible en tareas de documentación. En las formas más simples y relajadas de Entrega Evolutiva, a veces llamada Entrega Incremental, liberar parte de una solución es un Paso de Entrega Evolutiva. En la Entrega Evolutiva más pura y más rica, sólo las mejoras en las Metas de los Participantes se consideran un Paso de Entrega Evolutiva. Hay que distinguir bien, asimismo, entre los Medios y los Fines, por un lado, y las Metas y las Soluciones por el otro. Las Metas deben separarse de las Soluciones; las Metas deben ser sagradas, y deben alcanzarse por cualquier medio; las Soluciones son sólo posibles, contingentes: son caballos de trabajo, y deben cambiarse cuando se consigue un caballo mejor. Evo incluye, al lado de Planguage, lineamientos de métrica (todo es cuantificable en él), procedimientos formales y orientaciones para descomponer procesos grandes en pasos, políticas de planeamiento y un conjunto de plantillas y tablas de Estimación de Impacto.

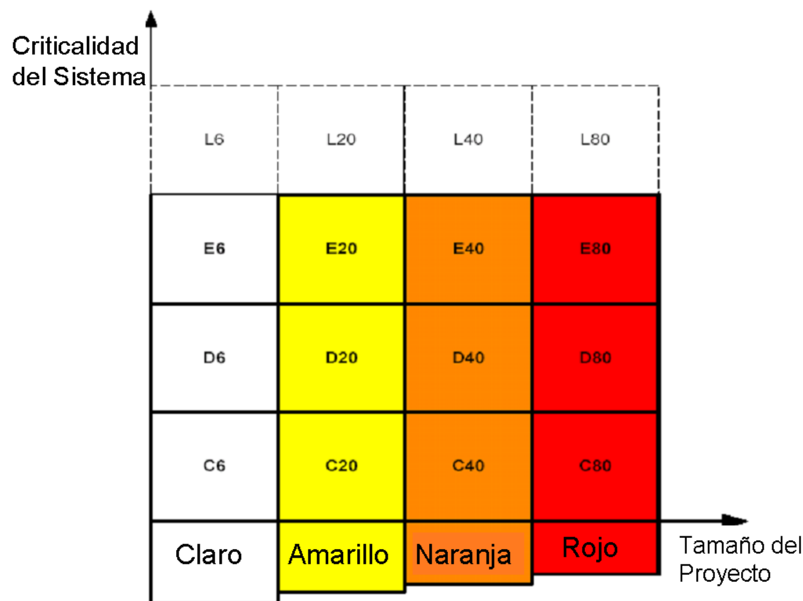
A diferencia de otras MAs que son más experimentales y que no tienen mucho respaldo de casos sistemáticamente documentados, Evo es una metodología probada desde hace mucho tiempo en numerosos clientes corporativos: la NASA, Lockheed Martin, Hewlett Packard, Douglas Aircraft, la Marina británica. El estándar MIL-STD-498 del Departamento de Defensa y su correspondiente estándar civil IEEE doc 12207 homologan el uso del modelo de entrega evolutiva. Sólo DSDM y RUP han logrado un reconocimiento comparable. Tom Gilb considera que las metodologías de Microsoft

reveladas en el best-seller *Microsoft Secrets* de Michael Cusumano y Richard Selby [CS95a], con su ciclo de construcción cotidiano a las 5 de la tarde y demás prácticas organizacionales, demuestra la vigencia de un método evolutivo. Todos los ejemplos de métodos evolutivos de uno de los libros mayores de Gilb [Gilb97] se ilustran con ejemplos de prácticas de Microsoft. En Evo no hay, por último, ni la sombra del carácter lúdico y la pintura de brocha gorda que se encuentran con frecuencia en Scrum o en XP. Se trata, por el contrario, de una metodología que impresiona por la precisión, relevancia e imaginación de sus fundamentaciones.

Crystal Methods

Las metodologías Crystal fueron creadas por el "antropólogo de proyectos" Alistair Cockburn, el autor que ha escrito los que tal vez sean los textos más utilizados, influyentes y recomendables sobre casos de uso, *Writing effective Use Cases* [Coc01] [Lar03]. Cockburn (quien siempre insiste que su apellido debe pronunciarse "Coburn" a la manera escocesa), escribe que mucha gente piensa que el desarrollo de software es una actividad de ingeniería. Esa comparación, piensa, es de hecho más perniciosa que útil, y nos lleva en una dirección equivocada.

Comparar el software con la ingeniería nos conduce a preguntarnos sobre "especificaciones" y "modelos" del software, sobre su completitud, corrección y vigencia. Esas preguntas son inconducentes, porque cuando pasa cierto tiempo no nos interesa que los modelos sean completos, que coincidan con el mundo "real" (sea ello lo que fuere) o que estén al día con la versión actual del lenguaje. Intentar que así sea es una pérdida de tiempo [Coc00]. En contra de esa visión ingenieril a la manera de un Bertrand Meyer, Cockburn ha alternado diversas visiones despreocupadamente contradictorias que alternativamente lo condujeron a adoptar XP en el sentido más radical, a sinergizarse con DSDM o LSD, a concebir el desarrollo de software como una forma comunitaria de poesía [Coc97a] o a elaborar su propia familia de Metodologías Crystal.



Familia de Crystal Methods [Crystalmethodologies.org]

La familia Crystal dispone un código de color para marcar la complejidad de una metodología: cuanto más oscuro un color, más “pesado” es el método. Cuanto más crítico es un sistema, más rigor se requiere. El código cromático se aplica a una forma tabular elaborada por Cockburn que se usa en muchos MAs para situar el rango de complejidad al cual se aplica una metodología. En la figura se muestra una evaluación de las pérdidas que puede ocasionar la falla de un sistema y el método requerido según este criterio. Los parámetros son Comodidad (C), Dinero Discrecional (D), Dinero Esencial (E) y Vidas (L). En otras palabras, la caída de un sistema que ocasione incomodidades indica que su nivel de criticalidad es C, mientras que si causa pérdidas de vidas su nivel es L. Los números del cuadro indican el número de personas afectadas a un proyecto, ±20%.

Los métodos se llaman Crystal evocando las facetas de una gema: cada faceta es otra versión del proceso, y todas se sitúan en torno a un núcleo idéntico. Hay cuatro variantes de metodologías: Crystal Clear (“Claro como el cristal”) para equipos de 8 o menos integrantes; Amarillo, para 8 a 20; Naranja, para 20 a 50; Rojo, para 50 a 100. Se promete seguir con Marrón, Azul y Violeta. La más exhaustivamente documentada es Crystal Clear (CC), y es la que se ha de describir a continuación. CC puede ser usado en proyectos pequeños de categoría D6, aunque con alguna extensión se aplica también a niveles E8 a D10. El otro método elaborado en profundidad es el Naranja, apto para proyectos de duración estimada en 2 años, descripto en *Surviving Object-Oriented Projects* [Coc97b]. Los otros dos aún se están desarrollando. Como casi todos los otros métodos, CC consiste en valores, técnicas y procesos.

Los siete valores o propiedades de CC [Coc02] son:

1. **Entrega frecuente**. Consiste en entregar software a los clientes con frecuencia, no solamente en compilar el código. La frecuencia dependerá del proyecto, pero puede ser diaria, semanal, mensual o lo que fuere. La entrega puede hacerse sin despliegue, si es que se consigue algún usuario cortés o curioso que suministre *feedback*.
2. **Comunicación osmótica**. Todos juntos en el mismo cuarto. Una variante especial es disponer en la sala de un diseñador *senior*; eso se llama **Experto al Alcance de la Oreja**. Una reunión separada para que los concurrentes se concentren mejor es descripta como **El Cono del Silencio**.
3. **Mejora reflexiva**. Tomarse un pequeño tiempo (unas pocas horas cada algunas semanas o una vez al mes) para pensar bien qué se está haciendo, cotejar notas, reflexionar, discutir.
4. **Seguridad personal**. Hablar cuando algo molesta: decirle amigablemente al *manager* que la agenda no es realista, o a un colega que su código necesita mejorarse, o que sería conveniente que se bañase más seguido. Esto es importante porque el equipo puede descubrir y reparar sus debilidades. No es provechoso encubrir los desacuerdos con gentileza y conciliación. Técnicamente, estas cuestiones se han caracterizado como una importante variable de *confianza* y han sido estudiadas con seriedad en la literatura.
5. **Foco**. Saber lo que se está haciendo y tener la tranquilidad y el tiempo para hacerlo. Lo primero debe venir de la comunicación sobre dirección y prioridades, típicamente con el Patrocinador Ejecutivo. Lo segundo, de un ambiente en que la gente no se vea compelida a hacer otras cosas incompatibles.

6. **Fácil acceso a usuarios expertos**. Una comunicación de Keil a la ACM demostró hace tiempo, sobre una amplia muestra estadística, la importancia del contacto directo con expertos en el desarrollo de un proyecto. No hay un dogma de vida o muerte sobre esto, como sí lo hay en XP. Un encuentro semanal o semi-semanal con llamados telefónicos adicionales parece ser una buena pauta. Otra variante es que los programadores se entrenen para ser usuarios durante un tiempo. El equipo de desarrollo, de todas maneras, incluye un *Experto en Negocios*.
7. **Ambiente técnico con prueba automatizada, *management* de configuración e integración frecuente**. Microsoft estableció la idea de los *builds* cotidianos, y no es una mala práctica. Muchos equipos ágiles compilan e integran varias veces al día.

Crystal Clear no requiere ninguna estrategia o técnica, pero conviene tener unas cuantas a mano para empezar. Las estrategias documentadas favoritas, comunes a otros MAs, son:

1. **Exploración de 360°**. Verificar o tomar una muestra del valor de negocios del proyecto, los requerimientos, el modelo de dominio, la tecnología, el plan del proyecto y el proceso. La exploración es preliminar al desarrollo y equivale al período de incepción de RUP. Mientras en RUP esto puede demandar algunas semanas o meses, en Crystal Clear debe insumir unos pocos días; como mucho dos semanas si se requiere usar una tecnología nueva o inusual. El muestreo de valor de negocios se puede hacer verbalmente, con casos de uso u otros mecanismos de listas, pero debe resultar en una lista de los casos de uso esenciales del sistema. Respecto de la tecnología conviene correr unos pocos experimentos en base a lo que Cunningham y Beck han llamado *Spikes* (http://c2.com/cgi/wiki?SpikeSolution).
2. **Victoria temprana**. Es mejor buscar pequeños triunfos iniciales que aspirar a una gran victoria tardía. La fundamentación de esta estrategia proviene de algunos estudios sociológicos específicos. Usualmente la primera victoria temprana consiste en la construcción de un Esqueleto Ambulante. Conviene no utilizar la técnica de "lo peor primero" de XP, porque puede bajar la moral. La preferencia de Cockburn es "lo más fácil primero, lo más difícil segundo".
3. **Esqueleto ambulante**. Es una transacción que debe ser simple pero completa. Podría ser una rutina de consulta y actualización en un sistema cliente-servidor, o la ejecución de una transacción en un sistema transaccional de negocios. Un Esqueleto Ambulante no suele ser robusto; sólo *camina*, y carece de la carne de la funcionalidad de la aplicación real, que se agregará incrementalmente. Es diferente de un *Spike*, porque éste es "la más pequeña implementación que demuestra un éxito técnico plausible" y después se tira, porque puede ser desprolijo y peligroso; un *Spike* sólo se usa para saber si se está en la dirección correcta. El Esqueleto debe producirse con buenos hábitos de producción y pruebas de regresión, y está destinado a crecer con el sistema.
4. **Rearquitectura incremental**. Se ha demostrado que no es conveniente interrumpir el desarrollo para corregir la arquitectura. Más bien la arquitectura debe evolucionar en etapas, manteniendo el sistema en ejecución mientras ella se modifica.
5. **Radiadores de información**. Es una lámina pegada en algún lugar que el equipo pueda observar mientras trabaja o camina. Tiene que ser comprensible para el observador casual, entendida de un vistazo y renovada periódicamente para que valga

la pena visitarla. Puede estar en una página Web, pero es mejor si está en una pared, porque en una máquina se acumulan tantas cosas que ninguna llama la atención. Podría mostrar el conjunto de la iteración actual, el número de pruebas pasadas o pendientes, el número de casos de uso o historias entregado, el estado de los servidores, los resultados del último Taller de Reflexión. Una variante creativa es un sistema de semáforos implementado por Freeman, Benson y Borning.

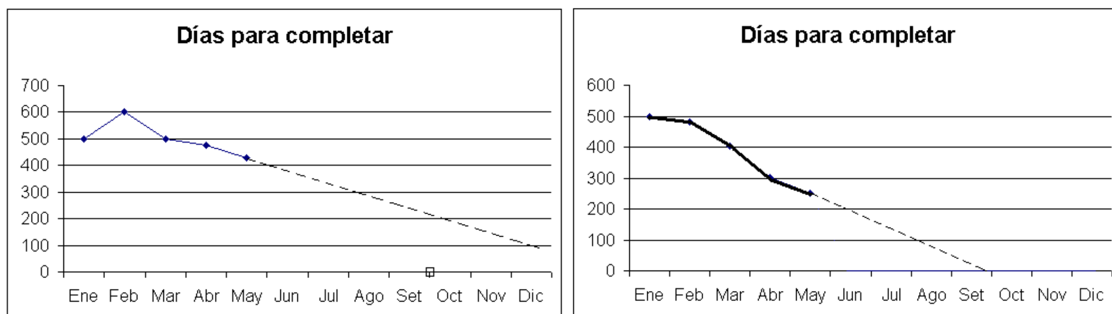
En cuanto a las técnicas, se favorecen:

1. **Entrevistas de proyectos**. Se suele entrevistar a más de un responsable para tener visiones más ricas. Cockburn ha elaborado una útil plantilla de dos páginas con entrevistas que probó ser útil en diversos proyectos. La idea es averiguar cuales son las prioridades, obtener una lista de rasgos deseados, saber cuáles son los requerimientos más críticos y cuáles los más negociables. Si se trata de una actualización o corrección, saber cuáles son las cosas que se hicieron bien y merecen preservarse y los errores que no se quieren repetir.
2. **Talleres de reflexión**. El equipo debe detenerse treinta minutos o una hora para reflexionar sobre sus convenciones de trabajo, discutir inconvenientes y mejoras y planear para el período siguiente. De aquí puede salir material para poner en un poster como Radiador de Información.
3. **Planeamiento Blitz**. Una técnica puede ser el Juego de Planeamiento de XP. En este juego, se ponen tarjetas indexadas en una mesa, con una historia de usuario o función visible en cada una. El grupo finge que no hay dependencias entre tarjetas, y las alinea en secuencias de desarrollo preferidas. Los programadores escriben en cada tarjeta el tiempo estimado para desarrollar cada función. El patrocinador o embajador del usuario escribe la secuencia de prioridades, teniendo en cuenta los tiempos referidos y el valor de negocio de cada función. Las tarjetas se agrupan en períodos de tres semanas llamados iteraciones que se agrupan en entregas (*releases*), usualmente no más largas de tres meses [Beck01]. Pueden usarse tarjetas CRC. Cockburn propone otras variantes del juego, como la Jam Session de Planeamiento del Proyecto[3] [Coc02]. Las diferencias entre la versión de Cockburn y el juego de XP son varias: en XP las tarjetas tienen historias, en CC listas de tareas; el juego de XP asume que no hay dependencias, el de CC que sí las hay; en XP hay iteraciones de duración fija, en CC no presupone nada sobre la longitud de la iteración.
4. **Estimación Delphi con estimaciones de pericia**. La técnica se llama así por analogía con el oráculo de Delfos; se la describió por primera vez en el clásico *Surviving Object-Oriented Projects* de Cockburn [Coc97b], reputado como uno de los mejores libros sobre el paradigma de objetos. En el proceso Delphi se reúnen los expertos responsables y proceden como en un remate para proponer el tamaño del sistema, su tiempo de ejecución, la fecha de las entregas según dependencias técnicas y de

[3] Hay aquí un juego de palabras deliberado que contrasta esta “Jam Session” con las “JAD Sessions” propias del RAD tradicional. Éstas eran reuniones intensivas donde participaba todo el mundo (técnicos y gente de negocios); duraban todo un día y se hacían lejos de la oficina. Estas excursiones lejos del lugar de trabajo no son comunes en los MAs. Las sesiones JAD son un proceso complejo y definido, sobre el cual hay abundante literatura [WS95].

negocios y para equilibrar las entregas en paquetes de igual tamaño. La descripción del remate está en [Coc02] e insume tres o cuatro páginas .

5. **Encuentros diarios de pie**. La palabra clave es “brevedad”, cinco a diez minutos como máximo. No se trata de discutir problemas, sino de identificarlos. Los problemas sólo se discuten en otros encuentros posteriores, con la gente que tiene que ver en ellos. La técnica se origina en Scrum. Se deben hacer de pie para que la gente no escriba en sus *notebooks*, garabatee papeles o se quede dormida.
6. **Miniatura de procesos**. La “Hora Extrema” fue inventada por Peter Merel para introducir a la gente en XP en 60 minutos (http://c2.com/cgi/wiki?ExtremeHour) y proporciona lineamientos canónicos que pueden usarse para articular esta práctica. Una forma de presentar Crystal Clear puede insumir entre 90 minutos y un día. La idea es que la gente pueda “degustar” la nueva metodología.
7. **Gráficos de quemado**. Su nombre viene de los gráficos de quemado de calorías de los regímenes dietéticos; se usan también en Scrum. Se trata de una técnica de graficación para descubrir demoras y problemas tempranamente en el proceso, evitando que se descubra demasiado tarde que todavía no se sabe cuánto falta. Para ello se hace una estimación del tiempo faltante para programar lo que resta al ritmo actual, lo cual sirve para tener dominio de proyectos en los cuales las prioridades cambian bruscamente y con frecuencia. Esta técnica se asocia con algunos recursos ingeniosos, como la *Lista Témpana*, llamada así porque se refiere al agregado de ítems con alta prioridad en el tope de las listas de trabajos pendientes, esperando que los demás elementos se hundan bajo la línea de flotación; los elementos que están sobre la línea se entregarán en la iteración siguiente, los que están por debajo en las restantes. En otros MAs la *Lista Témpana* no es otra cosa que un gráfico de retraso. Los gráficos de quemado ilustran la velocidad del proceso, analizando la diferencia entre las líneas proyectadas y efectivas de cada entrega, como se ilustra en las figuras.
8. **Programación lado a lado**. Mucha gente siente que la programación en pares de XP involucra una presión excesiva; la versión de Crystal Clear establece proximidad, pero cada quien se aboca a su trabajo asignado, prestando un ojo a lo que hace su compañero, quien tiene su propia máquina. Esta es una ampliación de la Comunicación Osmótica al contexto de la programación.

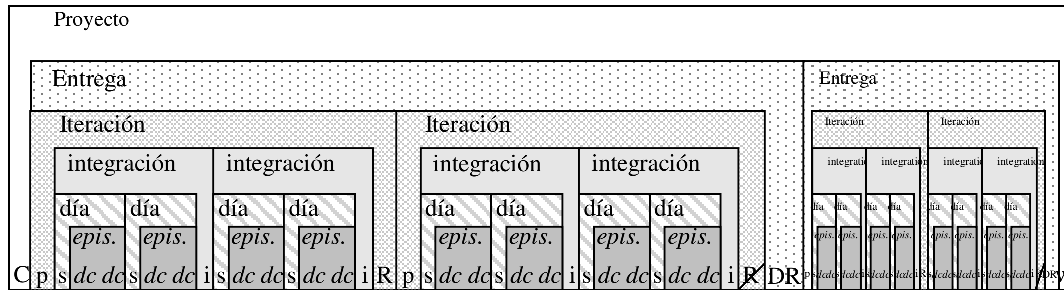


Gráficos de quemado – Con necesidad de recortar retrasos (izq.) y con entrega proyectada en término. Medición realizada en mayo – La fecha de entrega proyectada es el 1° de octubre

El proceso de CC se basa en una exploración refinada de los inconvenientes de los modelos clásicos. Dice Cockburn que la mayoría de los modelos de proceso propuestos

entre 1970 y 2000 se describían como secuencias de pasos. Aún cuando se recomendaran iteraciones e incrementos (que no hacían más que agregar confusión a la interpretación) los modelos parecían dictar un proceso en cascada, por más que los autores aseguraran que no era así. El problema con estos procesos es que realmente están describiendo un *workflow* requerido, un grafo de dependencia: el equipo no puede entregar un sistema hasta que está integrado y corre. No puede integrar y verificar hasta que el código no está escrito y corriendo. Y no puede diseñar y escribir el código hasta que se le dice cuáles son los requerimientos. Un grafo de dependencia se interpreta necesariamente en ese sentido, aunque no haya sido la intención original.

En lugar de esta interpretación lineal, CC enfatiza el proceso como un conjunto de ciclos anidados. En la mayoría de los proyectos se perciben siete ciclos: (1) el proyecto, (2) el ciclo de entrega de una unidad, (3) la iteración (nótese que CC requiere múltiples entregas por proyecto pero no muchas iteraciones por entrega), (4) la semana laboral, (5) el período de integración, de 30 minutos a tres días, (6) el día de trabajo, (7) el episodio de desarrollo de una sección de código, de pocos minutos a pocas horas.



Ciclos anidados de Crystal Clear [Coc02]

Cockburn subraya que interpretar no linealmente un modelo de ciclos es difícil; la mente se resiste a hacerlo. Él mismo asegura que estuvo diez años tratando de explicar el uso de un proceso cíclico y sólo recientemente ha logrado intuir cómo hacerlo. La figura muestra los ciclos y las actividades conectadas a ellos. Las letras denotan *Chartering*, planeamiento de iteración, reunión diaria de pie (*standup*), desarrollo, *check-in*, integración, taller de Reflexión, Entrega (*Delivery*), y empaquetado del proyecto (*Wrapup*).

Hay ocho roles nominados en CC: Patrocinador, Usuario Experto, Diseñador Principal, Diseñador-Programador, Experto en Negocios, Coordinador, Verificador, Escritor. En Crystal Naranja se agregan aun más roles: Diseñador de IU, Diseñador de Base de Datos, Experto en Uso, Facilitador Técnico, Analista/Diseñador de Negocios, Arquitecto, Mentor de Diseño, Punto de Reutilización. A continuación se describen en bastardilla los artefactos de los que son responsables los roles de CC, detalladamente descriptos en la documentación.

1. **Patrocinador**. Produce la *Declaración de Misión con Prioridades de Compromiso (Tradeoff)*. Consigue los recursos y define la totalidad del proyecto.
2. **Usuario Experto**. Junto con el Experto en Negocios produce la *Lista de Actores-Objetivos* y el *Archivo de Casos de Uso y Requerimientos*. Debe familiarizarse con el uso del sistema, sugerir atajos de teclado, modos de operación, información a visualizar simultáneamente, navegación, etcétera.

3. **Diseñador Principal**. Produce la *Descripción Arquitectónica*. Se supone que debe ser al menos un profesional de Nivel 3. (En MAs se definen tres niveles de experiencia: Nivel 1 es capaz de "seguir los procedimientos"; Nivel 2 es capaz de "apartarse de los procedimientos específicos" y encontrar otros distintos; Nivel 3 es capaz de manejar con fluidez, mezclar e inventar procedimientos). El Diseñador Principal tiene roles de coordinador, arquitecto, mentor y programador más experto.
4. **Diseñador-Programador**. Produce, junto con el Diseñador Principal, los *Borradores de Pantallas*, el *Modelo Común de Dominio*, las *Notas y Diagramas de Diseño*, el *Código Fuente*, el *Código de Migración*, las *Pruebas* y el *Sistema Empaquetado*. Cockburn no distingue entre diseñadores y programadores. Un programa en CC es "diseño y programa"; sus programadores son diseñadores-programadores. En CC un diseñador que no programe no tiene cabida.
5. **Experto en Negocios**. Junto con el Usuario Experto produce la *Lista de Actores-Objetivos* y el *Archivo de Casos de Uso y Requerimientos*. Debe conocer las reglas y políticas del negocio.
6. **Coordinador**. Con la ayuda del equipo, produce el *Mapa de Proyecto*, el *Plan de Entrega*, el *Estado del Proyecto*, la *Lista de Riesgos*, el *Plan y Estado de Iteración* y la *Agenda de Visualización*.
7. **Verificador**. Produce el *Reporte de Bugs*. Puede ser un programador en tiempo parcial, o un equipo de varias personas.
8. **Escritor**. Produce el *Manual de Usuario*.

El **Equipo como Grupo** es responsable de producir la *Estructura y Convenciones del Equipo* y los *Resultados del Taller de Reflexión*.

A pesar que no contempla el desarrollo de software propiamente dicho, CC involucra unos veinte productos de trabajo o artefactos. Mencionamos los más importantes:

1. Declaración de la misión. Documento de un párrafo a una página, describiendo el propósito.
2. Estructura del equipo. Lista de equipos y miembros.
3. Metodología. Comprende roles, estructura, proceso, productos de trabajo que mantienen, métodos de revisión.
4. Secuencia de entrega. Declaración o diagrama de dependencia; muestra el orden de las entregas y lo que hay en cada una.
5. Cronograma de visualización y entrega. Lista, planilla de hoja de cálculo o herramienta de gestión de proyectos.
6. Lista de riesgos. Descripción de riesgos por orden descendente de prioridad.
7. Estatus del proyecto. Lista hitos, fecha prevista, fecha efectiva y comentarios.
8. Lista de actores-objetivos. Lista de dos columnas, planilla de hoja de cálculo, diagrama de caso de uso o similar.
9. Casos de uso anotados. Requerimientos funcionales.
10. Archivo de requerimientos. Colección de información indicando qué se debe construir, quiénes han de utilizarlo, de qué manera proporciona valor y qué restricciones afectan al diseño.

Los métodos Crystal no prescriben las prácticas de desarrollo, las herramientas o los productos que pueden usarse, pudiendo combinarse con otros métodos como Scrum, XP y Microsoft Solutions Framework. En su comentario a [Hig00b], Cockburn confiesa que cuando imaginó CC pensaba proporcionar un método ligero; comparado con XP, sin embargo, CC resulta muy pesado. CC es más fácil de aprender e implementar; a pesar de su jerga chocante XP es más disciplinado, piensa Cockburn; pero si un equipo ligero puede tolerar sus rigores, lo mejor será que se mude a XP.

Feature Driven Development (FDD)

Feature Oriented Programming (FOP) es una técnica de programación guiada por rasgos o características (*features*) y centrada en el usuario, no en el programador; su objetivo es sintetizar un programa conforme a los rasgos requeridos [Bat03]. En un desarrollo en términos de FOP, los objetos se organizan en módulos o capas conforme a rasgos. FDD, en cambio, es un método ágil, iterativo y adaptativo. A diferencia de otros MAs, no cubre todo el ciclo de vida sino sólo las fases de diseño y construcción y se considera adecuado para proyectos mayores y de misión crítica. FDD es, además, marca registrada de una empresa, Nebulon Pty. Aunque hay coincidencias entre la programación orientada por rasgos y el desarrollo guidado por rasgos, FDD no necesariamente implementa FOP.

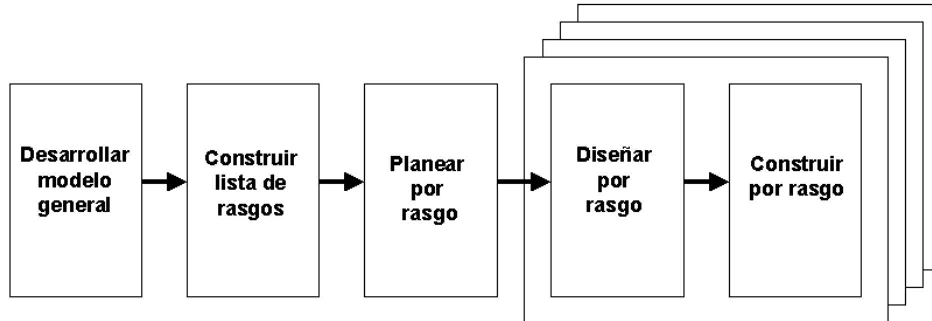
FDD no requiere un modelo específico de proceso y se complementa con otras metodologías. Enfatiza cuestiones de calidad y define claramente entregas tangibles y formas de evaluación del progreso. Se lo reportó por primera vez en un libro de Peter Coad, Eric Lefebvre y Jeff DeLuca *Java Modeling in Color with UML* [CLD00]; luego fue desarrollado con amplitud en un proyecto mayor de desarrollo por DeLuca, Coad y Stephen Palmer. Su implementación de referencia, análogo al C3 de XP, fue el Singapore Project; DeLuca había sido contratado para salvar un sistema muy complicado para el cual el contratista anterior había producido, luego de dos años, 3500 páginas de documentación y ninguna línea de código. Naturalmente, el proyecto basado en FDD fue todo un éxito, y permitió fundar el método en un caso real de misión crítica.

Los principios de FDD son pocos y simples:

- Se requiere un sistema para construir sistemas si se pretende escalar a proyectos grandes.
- Un proceso simple y bien definido trabaja mejor.
- Los pasos de un proceso deben ser lógicos y su mérito inmediatamente obvio para cada miembro del equipo.
- Vanagloriarse del proceso puede impedir el trabajo real.
- Los buenos procesos van hasta el fondo del asunto, de modo que los miembros del equipo se puedan concentrar en los resultados.
- Los ciclos cortos, iterativos, orientados por rasgos (*features*) son mejores.

Hay tres categorías de rol en FDD: roles claves, roles de soporte y roles adicionales. Los seis roles claves de un proyecto son: (1) administrador del proyecto, quien tiene la última palabra en materia de visión, cronograma y asignación del personal; (2) arquitecto jefe (puede dividirse en arquitecto de dominio y arquitecto técnico); (3) *manager* de desarrollo, que puede combinarse con arquitecto jefe o *manager* de proyecto; (4)

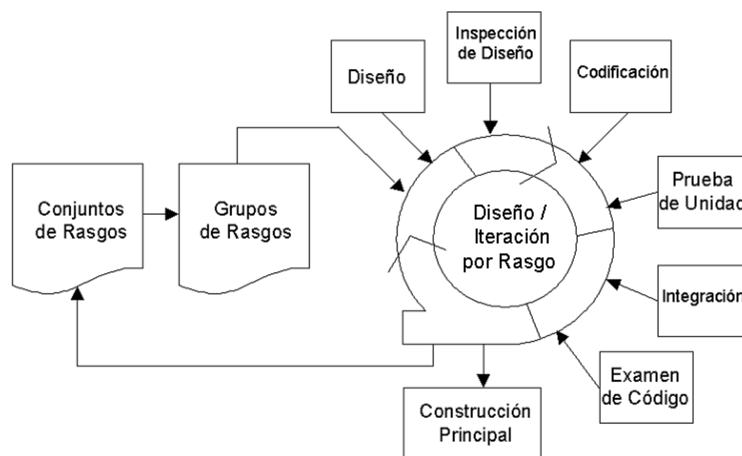
programador jefe, que participa en el análisis del requerimiento y selecciona rasgos del conjunto a desarrollar en la siguiente iteración; (5) propietarios de clases, que trabajan bajo la guía del programador jefe en diseño, codificación, prueba y documentación, repartidos por rasgos y (6) experto de dominio, que puede ser un cliente, patrocinador, analista de negocios o una mezcla de todo eso.



Proceso FDD, basado en [http://togethercommunities.com]

Los cinco roles de soporte comprenden (1) administrador de entrega, que controla el progreso del proceso revisando los reportes del programador jefe y manteniendo reuniones breves con él; reporta al manager del proyecto; (2) abogado/guru de lenguaje, que conoce a la perfección el lenguaje y la tecnología; (3) ingeniero de construcción, que se encarga del control de versiones de los *builds* y publica la documentación; (4) herramientista (*toolsmith*), que construye herramientas ad hoc o mantiene bases de datos y sitios Web y (5) administrador del sistema, que controla el ambiente de trabajo o productiza el sistema cuando se lo entrega.

Los tres roles adicionales son los de verificadores, encargados del despliegue y escritores técnicos. Un miembro de un equipo puede tener otros roles a cargo, y un solo rol puede ser compartido por varias personas.



Ciclo de FDD, basado en [ASR+02]

FDD consiste en cinco procesos secuenciales durante los cuales se diseña y construye el sistema. La parte iterativa soporta desarrollo ágil con rápidas adaptaciones a cambios en requerimientos y necesidades del negocio. Cada fase del proceso tiene un criterio de entrada, tareas, pruebas y un criterio de salida. Típicamente, la iteración de un rasgo insume de una a tres semanas. Las fases son:

1) **Desarrollo de un modelo general**. Cuando comienza este desarrollo, los expertos de dominio ya están al tanto de la visión, el contexto y los requerimientos del sistema a construir. A esta altura se espera que existan requerimientos tales como casos de uso o especificaciones funcionales. FDD, sin embargo, no cubre este aspecto. Los expertos de dominio presentan un ensayo (*walkthrough*) en el que los miembros del equipo y el arquitecto principal se informan de la descripción de alto nivel del sistema. El dominio general se subdivide en áreas más específicas y se define un ensayo más detallado para cada uno de los miembros del dominio. Luego de cada ensayo, un equipo de desarrollo trabaja en pequeños grupos para producir modelos de objeto de cada área de dominio. Simultáneamente, se construye un gran modelo general para todo el sistema.
2) **Construcción de la lista de rasgos**. Los ensayos, modelos de objeto y documentación de requerimientos proporcionan la base para construir una amplia lista de rasgos. Los rasgos son pequeños ítems útiles a los ojos del cliente. Son similares a las tarjetas de historias de XP y se escriben en un lenguaje que todas las partes puedan entender. Las funciones se agrupan conforme a diversas actividades en áreas de dominio específicas. La lista de rasgos es revisada por los usuarios y patrocinadores para asegurar su validez y exhaustividad. Los rasgos que requieran más de diez días se descomponen en otros más pequeños.
3) **Planeamiento por rasgo**. Incluye la creación de un plan de alto nivel, en el que los conjuntos de rasgos se ponen en secuencia conforme a su prioridad y dependencia, y se asigna a los programadores jefes. Las listas se priorizan en secciones que se llaman paquetes de diseño. Luego se asignan las clases definidas en la selección del modelo general a programadores individuales, o sea propietarios de clases. Se pone fecha para los conjuntos de rasgos.
4) **Diseño por rasgo y Construcción por rasgo**. Se selecciona un pequeño conjunto de rasgos del conjunto y los propietarios de clases seleccionan los correspondientes equipos dispuestos por rasgos. Se procede luego iterativamente hasta que se producen los rasgos seleccionados. Una iteración puede tomar de unos pocos días a un máximo de dos semanas. Puede haber varios grupos trabajando en paralelo. El proceso iterativo incluye inspección de diseño, codificación, prueba de unidad, integración e inspección de código. Luego de una iteración exitosa, los rasgos completos se promueven al *build* principal. Este proceso puede demorar una o dos semanas en implementarse.

FDD consiste en un conjunto de “mejores prácticas” que distan de ser nuevas pero se combinan de manera original. Las prácticas canónicas son:

- Modelado de objetos del dominio, resultante en un *framework* cuando se agregan los rasgos. Esta forma de modelado descompone un problema mayor en otros menores; el diseño y la implementación de cada clase u objeto es un problema pequeño a resolver. Cuando se combinan las clases completas, constituyen la solución al problema mayor. Una forma particular de la técnica es el modelado en colores [CLD00], que agrega una dimensión adicional de visualización. Si bien se puede modelar en blanco y negro, en FDD el modelado basado en objetos es imperativo.

- Desarrollo por rasgo. Hacer simplemente que las clases y objetos funcionen no refleja lo que el cliente pide. El seguimiento del progreso se realiza mediante examen de pequeñas funcionalidades descompuestas y funciones valoradas por el cliente. Un rasgo en FDD es una función pequeña expresada en la forma <acción> <resultado> <por | para | de | a> <objeto> con los operadores adecuados entre los términos. Por ejemplo, calcular el importe total de una venta; determinar la última operación de un cajero; validar la contraseña de un usuario.
- Propiedad individual de clases (código). Cada clase tiene una sola persona nominada como responsable por su consistencia, performance e integridad conceptual.
- Equipos de Rasgos, pequeños y dinámicamente formados. La existencia de un equipo garantiza que un conjunto de mentes se apliquen a cada decisión y se tomen en cuenta múltiples alternativas.
- Inspección. Se refiere al uso de los mejores mecanismos de detección conocidos. FDD es tan escrupuloso en materia de inspección como lo es Evo.
- *Builds* regulares. Siempre se tiene un sistema disponible. Los *builds* forman la base a partir de la cual se van agregando nuevos rasgos.
- Administración de configuración. Permite realizar seguimiento histórico de las últimas versiones completas de código fuente.
- Reporte de progreso. Se comunica a todos los niveles organizacionales necesarios.

FDD suministra un rico conjunto de artefactos para la planificación y control de los proyectos. En http://www.nebulon.com/articles/fdd/fddimplementations.html se encuentran diversos formularios y tablas con información real de implementaciones de FDD: Vistas de desarrollo, Vistas de planificación, Reportes de progreso, Reportes de tendencia, Vista de plan (<acción><resultado><objeto>), etcétera. Se han desarrollado también algunas herramientas que generan vistas combinadas o específicas.

Id	Descripción	Prog. Jefe.	Prop. Clase	Ensayo		Diseño		Inspección de Diseño		Código		Inspección de Código		Promover a Build	
				Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual
MD125	Validar los límites transaccionales de un CAO contra una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD126	Definir el estado de una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD127	Especificar el oficial de autorización de una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD128	Rechazar una instrucción de implementación para un conjunto de líneas	CP	ABC	STATUS: Inactivo NOTA: [agregado por CK: 3/2/99] No aplicable											
MD129	Confirmar una instrucción de implementación para un conjunto de líneas	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD130	Determinar si todos los documentos se han completado para un prestatario	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99		08/02/99		10/02/99	
				NOTA: [agregado por SL: 3/2/99] Bloqueado en AS											
MD131	Validar los límites transaccionales de un CAO contra una instrucción de desembolso	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99		08/02/99		10/02/99	
				NOTA: [agregado por: 3/2/99] Atrasado según estimaciones iniciales											
MD132	Enviar para autorización una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99	05/02/99	06/02/99	06/02/99	08/02/99	08/02/99
MD133	Validar fecha de baja de una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99	05/02/99	06/02/99	06/02/99	08/02/99	08/02/99

Plan de rasgo – Implementación – Basado en http://www.nebulon.com/articles/fdd/planview.html

La matriz muestra un ejemplo de vista de un plan de rasgo, con la típica codificación en colores.

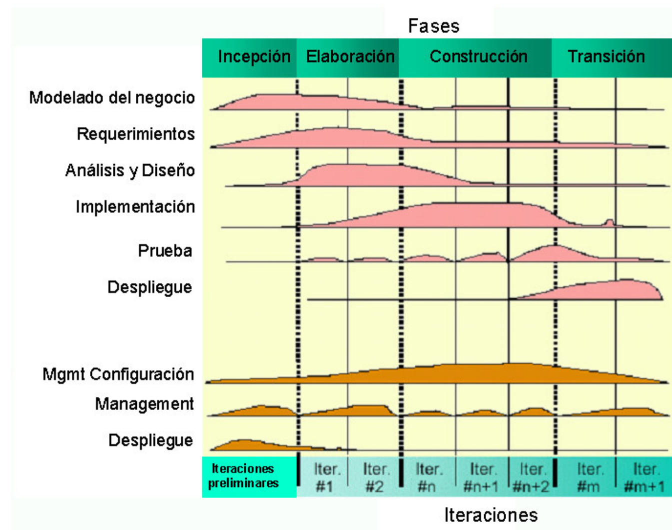
En síntesis, FDD es un método de desarrollo de ciclos cortos que se concentra en la fase de diseño y construcción. En la primera fase, el modelo global de dominio es elaborado por expertos del dominio y desarrolladores; el modelo de dominio consiste en diagramas de clases con clases, relaciones, métodos y atributos. Los métodos no reflejan conveniencias de programación sino rasgos funcionales.

Algunos agilistas sienten que FDD es demasiado jerárquico para ser un método ágil, porque demanda un programador jefe, quien dirige a los propietarios de clases, quienes dirigen equipos de rasgos. Otros críticos sienten que la ausencia de procedimientos detallados de prueba en FDD es llamativa e impropia. Los promotores del método aducen que las empresas ya tienen implementadas sus herramientas de prueba, pero subsiste el problema de su adecuación a FDD. Un rasgo llamativo de FDD es que no exige la presencia del cliente.

FDD se utilizó por primera vez en grandes aplicaciones bancarias a fines de la década de 1990. Los autores sugieren su uso para proyectos nuevos o actualizaciones de sistemas existentes, y recomiendan adoptarlo en forma gradual. En [ASR+02] se asegura que aunque no hay evidencia amplia que documente sus éxitos, las grandes consultoras suelen recomendarlo incluso para delicados proyectos de misión crítica.

Rational Unified Process (RUP)

Uno podría preguntarse legítimamente qué hace RUP en el contexto de los MAs ¿No es más bien representativo de la filosofía a la que el Manifiesto se opone? ¿Están tratando los métodos clásicos de cooptar a los métodos nuevos? El hecho es que existe una polémica aún en curso respecto de si los métodos asociados al Proceso Unificado, y en particular RUP, representan técnicas convencionales y pesadas o si por el contrario son adaptables al programa de los MAs.



Fases y *workflows* de RUP, basado en [BMP98]

Philippe Kruchten, impulsor tanto del famoso modelo de vistas 4+1 como de RUP, ha participado en el célebre "debate de los gurúes" [Hig01] afirmando que RUP es particularmente apto para ambas clases de escenarios. Kruchten ha sido en general conciliador; aunque el diseño orientado a objetos suele otorgar a la Arquitectura de Software académica un lugar modesto, el modelo de 4+1 [Kru95] se inspira explícitamente en el modelo arquitectónico fundado por Dewayne Perry y Alexander Wolf [PW92]. Recíprocamente, 4+1 tiene cabida en casi todos los métodos basados en arquitectura desarrollados recientemente por el SEI. Highsmith, comentando la movilidad de RUP de un campo a otro, señaló que nadie, en apariencia, está dispuesto a conceder "agilidad" a sus rivales; RUP pretende ser ágil, y tal vez lo sea. De hecho, los principales textos que analizan MAs acostumbran a incluir al RUP entre los más representativos, o como un método que no se puede ignorar [ASR+02] [Lar04].

Algunos autores no admiten esta concesión. En un interesante análisis crítico, Wolfgang Hesse, de la Universidad de Marburg [HesS/f] ha señalado que aunque el RUP se precia de sus cualidades iterativas, incrementales y centradas en la arquitectura, de hecho responde subterráneamente a un rígido modelo de fases, no posee capacidades recursivas suficientes y sus definiciones dinámicas son demasiado engorrosas y recargadas como para ser de utilidad en un contexto cambiante.

El proceso de ciclo de vida de RUP se divide en cuatro fases bien conocidas llamadas Incepción, Elaboración, Construcción y Transición. Esas fases se dividen en iteraciones, cada una de las cuales produce una pieza de software demostrable. La duración de cada iteración puede extenderse desde dos semanas hasta seis meses. Las fases son:

1. **Incepción**. Significa "comienzo", pero la palabra original (de origen latino y casi en desuso como sustantivo) es sugestiva y por ello la traducimos así. Se especifican los objetivos del ciclo de vida del proyecto y las necesidades de cada participante. Esto entraña establecer el alcance y las condiciones de límite y los criterios de aceptabilidad. Se identifican los casos de uso que orientarán la funcionalidad. Se diseñan las arquitecturas candidatas y se estima la agenda y el presupuesto de todo el proyecto, en particular para la siguiente fase de elaboración. Típicamente es una fase breve que puede durar unos pocos días o unas pocas semanas.
2. **Elaboración**. Se analiza el dominio del problema y se define el plan del proyecto. RUP presupone que la fase de elaboración brinda una arquitectura suficientemente sólida junto con requerimientos y planes bastante estables. Se describen en detalle la infraestructura y el ambiente de desarrollo, así como el soporte de herramientas de automatización. Al cabo de esta fase, debe estar identificada la mayoría de los casos de uso y los actores, debe quedar descripta la arquitectura de software y se debe crear un prototipo de ella. Al final de la fase se realiza un análisis para determinar los riesgos y se evalúan los gastos hechos contra los originalmente planeados.
3. **Construcción**. Se desarrollan, integran y verifican todos los componentes y rasgos de la aplicación. RUP considera que esta fase es un proceso de manufactura, en el que se debe poner énfasis en la administración de los recursos y el control de costos, agenda y calidad. Los resultados de esta fase (las versiones alfa, beta y otras versiones de prueba) se crean tan rápido como sea posible. Se debe compilar también una versión de entrega. Es la fase más prolongada de todas.

4. **Transición**. Comienza cuando el producto está suficientemente maduro para ser entregado. Se corrigen los últimos errores y se agregan los rasgos pospuestos. La fase consiste en prueba beta, piloto, entrenamiento a usuarios y despacho del producto a mercadeo, distribución y ventas. Se produce también la documentación. Se llama transición porque se transfiere a las manos del usuario, pasando del entorno de desarrollo al de producción.

A través de las fases se desarrollan en paralelo nueve *workflows* o disciplinas: Modelado de Negocios, Requerimientos, Análisis & Diseño, Implementación, Prueba, Gestión de Configuración & Cambio, Gestión del Proyecto y Entorno. Además de estos *workflows*, RUP define algunas prácticas comunes:

1. **Desarrollo iterativo de software**. Las iteraciones deben ser breves y proceder por incrementos pequeños. Esto permite identificar riesgos y problemas tempranamente y reaccionar frente a ellos en consecuencia.
2. **Administración de requerimientos**. Identifica requerimientos cambiantes y postula una estrategia disciplinada para administrarlos.
3. **Uso de arquitecturas basadas en componentes**. La reutilización de componentes permite asimismo ahorros sustanciales en tiempo, recursos y esfuerzo.
4. **Modelado visual del software.** Se deben construir modelos visuales, porque los sistemas complejos no podrían comprenderse de otra manera. Utilizando una herramienta como UML, la arquitectura y el diseño se pueden especificar sin ambigüedad y comunicar a todas las partes involucradas.
5. **Prueba de calidad del software**. RUP pone bastante énfasis en la calidad del producto entregado.
6. **Control de cambios y trazabilidad**. La madurez del software se puede medir por la frecuencia y tipos de cambios realizados.

Aunque RUP es extremadamente locuaz en muchos respectos, no proporciona lineamientos claros de implementación que puedan compararse, por ejemplo, a los métodos Crystal, en los que se detalla la documentación requerida y los roles según diversas escalas de proyecto. En RUP esas importantes decisiones de dejan a criterio del usuario. Se asegura [Kru00] que RUP puede implementarse “sacándolo de la caja”, pero dado que el número de sus artefactos y herramientas es inmenso, siempre se dice que hay que recortarlo y adaptarlo a cada caso. El proceso de implementación mismo es complejo, dividiéndose en seis fases cíclicas.

Existe una versión recortada de RUP, dX de Robert Martin, en la cual se han tomado en consideración experiencias de diversos MAs, reduciendo los artefactos de RUP a sus mínimos esenciales y (en un gesto heroico) usando tarjetas de fichado en lugar de UML. Es como si fuera RUP imitando los principios de XP; algunos piensan que dX es XP de cabo a rabo, sólo que con algunos nombres cambiados [ASR+02]. RUP se ha combinado con Evo, Scrum, MSF y cualquier metodología imaginable. Dado que RUP es suficientemente conocido y su estructura es más amplia y compleja que el de cualquier otro método ágil, su tratamiento en este texto concluye en este punto.

Dynamic Systems Development Method (DSDM)

Originado en los trabajos de Jennifer Stapleton, directora del DSDM Consortium, DSDM se ha convertido en el framework de desarrollo rápido de aplicaciones (RAD) más popular de Gran Bretaña [Sta97] y se ha llegado a promover como el estándar de facto para desarrollo de soluciones de negocios sujetas a márgenes de tiempo estrechos. Se calcula que uno de cada cinco desarrolladores en Gran Bretaña utiliza DSDM y que más de 500 empresas mayores lo han adoptado.

Además de un método, DSDM proporciona un framework completo de controles para RAD y lineamientos para su uso. DSDM puede complementar metodologías de XP, RUP o Microsoft Solutions Framework, o combinaciones de todas ellas. DSDM es relativamente antiguo en el campo de los MAs y constituye una metodología madura, que ya va por su cuarta versión. Se dice que ahora las iniciales DSDM significan Dynamic Solutions Delivery Method. Ya no se habla de sistemas sino de soluciones, y en lugar de priorizar el desarrollo se prefiere enfatizar la entrega. El libro más reciente que sintetiza las prácticas se llama *DSDM: Business Focused Development* [DS03].

La idea dominante detrás de DSDM es explícitamente inversa a la que se encuentra en otras partes, y al principio resulta contraria a la intuición; en lugar de ajustar tiempo y recursos para lograr cada funcionalidad, en esta metodología tiempo y recursos se mantienen como constantes y se ajusta la funcionalidad de acuerdo con ello. Esto se expresa a través de reglas que se conocen como “reglas MoSCoW” por las iniciales de su estipulación en inglés. Las reglas se refieren a rasgos del requerimiento:

1. **Must have: Debe tener**. Son los requerimientos fundamentales del sistema. De éstos, el subconjunto mínimo ha de ser satisfecho por completo.
2. **Should have: Debería tener**. Son requerimientos importantes para los que habrá una resolución en el corto plazo.
3. **Could have: Podría tener**. Podrían quedar fuera del sistema si no hay más remedio.
4. **Want to have but won’t have this time around: Se desea que tenga, pero no lo tendrá esta vuelta**. Son requerimientos valorados, pero pueden esperar.

DSDM consiste en cinco fases:

1. Estudio de viabilidad.
2. Estudio del negocio.
3. Iteración del modelo funcional.
4. Iteración de diseño y versión.
5. Implementación.

Las últimas tres fases son iterativas e incrementales. De acuerdo con la iniciativa de mantener el tiempo constante, las iteraciones de DSDM son cajas de tiempo. La iteración acaba cuando el tiempo se consume. Se supone que al cabo de la iteración los resultados están garantizados. Una caja de tiempo puede durar de unos pocos días a unas pocas semanas.

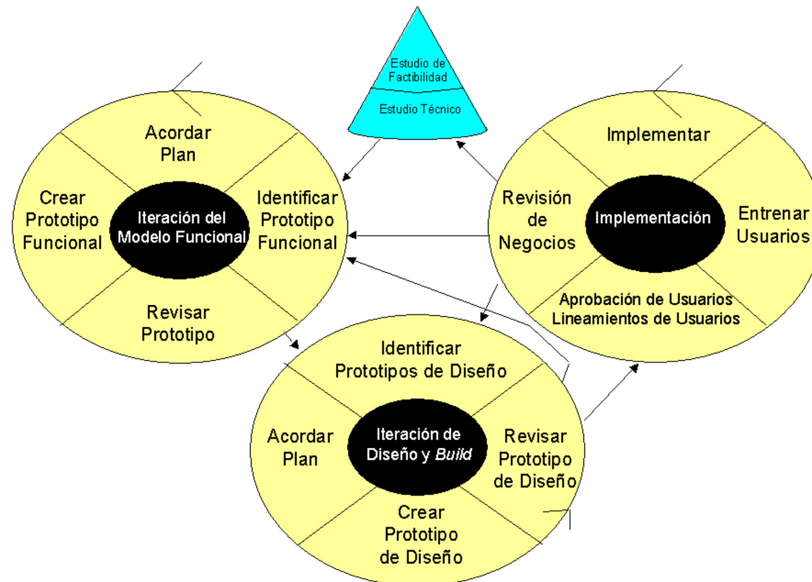
A diferencia de otros AMs, DSDM ha desarrollado sistemáticamente el problema de su propia implantación en una empresa. El proceso de Examen de Salud (*Health Check*) de DSDM se divide en dos partes que se interrogan, sucesivamente, sobre la capacidad de

una organización para adoptar el método y sobre la forma en que éste responde a las necesidades una vez que el proyecto está encaminado. Un Examen de Salud puede insumir entre tres días y un mes de trabajo de consultoría.

1. **Estudio de factibilidad**. Se evalúa el uso de DSDM o de otra metodología conforme al tipo de proyecto, variables organizacionales y de personal. Si se opta por DSDM, se analizan las posibilidades técnicas y los riesgos. Se preparan como productos un reporte de viabilidad y un plan sumario para el desarrollo. Si la tecnología no se conoce bien, se hace un pequeño prototipo para ver qué pasa. No se espera que el estudio completo insuma más de unas pocas semanas. Es mucho para un método ágil, pero menos de lo que demandan algunos métodos clásicos.
2. **Estudio del negocio**. Se analizan las características del negocio y la tecnología. La estrategia recomendada consiste en el desarrollo de talleres, donde se espera que los expertos del cliente consideren las facetas del sistema y acuerden sus prioridades de desarrollo. Se describen los procesos de negocio y las clases de usuario en una Definición del Área de Negocios. Se espera así reconocer e involucrar a gente clave de la organización en una etapa temprana. La Definición utiliza descripciones de alto nivel, como diagramas de entidad-relación o modelos de objetos de negocios. Otros productos son la Definición de Arquitectura del Sistema y el Plan de Bosquejo de Prototipado. La definición arquitectónica es un primer bosquejo y se admite que cambie en el curso del proyecto DSDM. El plan debe establecer la estrategia de prototipado de las siguientes etapas y un plan para la gestión de configuración.
3. **Iteración del modelo funcional**. En cada iteración se planea el contenido y la estrategia, se realiza la iteración y se analizan los resultados pensando en las siguientes. Se lleva a cabo tanto el análisis como el código; se construyen los prototipos y en base a la experiencia se mejoran los modelos de análisis. Los prototipos no han de ser descartados por completo, sino gradualmente mejorados hacia la calidad que debe tener el producto final. Se produce como resultado un Modelo Funcional, conteniendo el código del prototipo y los modelos de análisis. También se realizan pruebas constantemente. Hay otros cuatro productos emergentes: (1) Funciones Priorizadas es una lista de funciones entregadas al fin de cada iteración; (2) los Documentos de Revisión del Prototipado Funcional reúnen los comentarios de los usuarios sobre el incremento actual para ser considerados en iteraciones posteriores; (3) los Requerimientos Funcionales son listas que se construyen para ser tratadas en fases siguientes; (4) el Análisis de Riesgo de Desarrollo Ulterior es un documento importante en la fase de iteración del modelo, porque desde la fase siguiente en adelante los problemas que se encuentren serán más difíciles de tratar.
4. **Iteración de diseño y construcción**. Aquí es donde se construye la mayor parte del sistema. El producto es un Sistema Probado que cumplimenta por lo menos el conjunto mínimo de requerimientos acordados conforme a las reglas MoSCoW. El diseño y la construcción son iterativos y el diseño y los prototipos funcionales son revisados por usuarios. El desarrollo ulterior se atiene a sus comentarios.
5. **Despliegue**. El sistema se transfiere del ambiente de desarrollo al de producción. Se entrena a los usuarios, que ponen las manos en el sistema. Eventualmente la fase puede llegar a iterarse. Otros productos son el Manual de Usuario y el Reporte de Revisión del Sistema. A partir de aquí hay cuatro cursos de acción posibles: (1) Si el

sistema satisface todos los requerimientos, el desarrollo ha terminado. (2) Si quedan muchos requerimientos sin resolver, se puede correr el proceso nuevamente desde el comienzo. (3) Si se ha dejado de lado alguna prestación no crítica, el proceso se puede correr desde la iteración funcional del modelo en adelante. (4) si algunas cuestiones técnicas no pudieron resolverse por falta de tiempo se puede iterar desde la fase de diseño y construcción.

La configuración del ciclo de vida de DSDM se representa con un diagrama característico (del cual hay una evocación en el logotipo del consorcio) que vale la pena reproducir:



Proceso de desarrollo DSDM, basado en [http://www.dsdm.org]

DSDM define quince roles, algo más que el promedio de los MAs. Los más importantes son:

1. **Programadores y Programadores Senior**. Son los únicos roles de desarrollo. El título de Senior indica también nivel de liderazgo dentro del equipo. Equivale a Nivel 3 de Cockburn. Ambos títulos cubren todos los roles de desarrollo, incluyendo analistas, diseñadores, programadores y verificadores.
2. **Coordinador técnico**. Define la arquitectura del sistema y es responsable por la calidad técnica del proyecto, el control técnico y la configuración del sistema.
3. **Usuario embajador**. Proporciona al proyecto conocimiento de la comunidad de usuarios y disemina información sobre el progreso del sistema hacia otros usuarios. Se define adicionalmente un rol de **Usuario Asesor** (*Advisor*) que representa otros puntos de vista importantes; puede ser alguien del personal de IT o un auditor funcional.
4. **Visionario**. Es un usuario participante que tiene la percepción más exacta de los objetivos del sistema y el proyecto. Asegura que los requerimientos esenciales se cumplan y que el proyecto vaya en la dirección adecuada desde el punto de vista de aquéllos.

5. **Patrocinador Ejecutivo**. Es la persona de la organización que detenta autoridad y responsabilidad financiera, y es quien tiene la última palabra en las decisiones importantes.
6. **Facilitador**. Es responsable de administrar el progreso del taller y el motor de la preparación y la comunicación.
7. **Escriba**. Registra los requerimientos, acuerdos y decisiones alcanzadas en las reuniones, talleres y sesiones de prototipado.

En DSDM las prácticas se llaman Principios, y son nueve:

1. Es imperativo el compromiso activo del usuario.
2. Los equipos de DSDM deben tener el poder de tomar decisiones.
3. El foco radica en la frecuente entrega de productos.
4. El criterio esencial para la aceptación de los entregables es la adecuación a los propósitos de negocios.
5. Se requiere desarrollo iterativo e incremental.
6. Todos los cambios durante el desarrollo son reversibles.
7. La línea de base de los requerimientos es de alto nivel. Esto permite que los requerimientos de detalle se cambien según se necesite y que los esenciales se capten tempranamente.
8. La prueba está integrada a través de todo el ciclo de vida. La prueba también es incremental. Se recomienda particularmente la prueba de regresión, de acuerdo con el estilo evolutivo de desarrollo.
9. Es esencial una estrategia colaborativa y cooperativa entre todos los participantes. Las responsabilidades son compartidas y la colaboración entre usuario y desarrolladores no debe tener fisuras.

Desde mediados de la década de 1990 hay abundantes estudios de casos, sobre todo en Gran Bretaña, y la adecuación de DSDM para desarrollo rápido está suficientemente probada [ASR+02]. El equipo mínimo de DSDM es de dos personas y puede llegar a seis, pero puede haber varios equipos en un proyecto. El mínimo de dos personas involucra que un equipo consiste de un programador y un usuario. El máximo de seis es el valor que se encuentra en la práctica. DSDM se ha aplicado a proyectos grandes y pequeños. La precondición para su uso en sistemas grandes es su partición en componentes que pueden ser desarrollados por equipos normales.

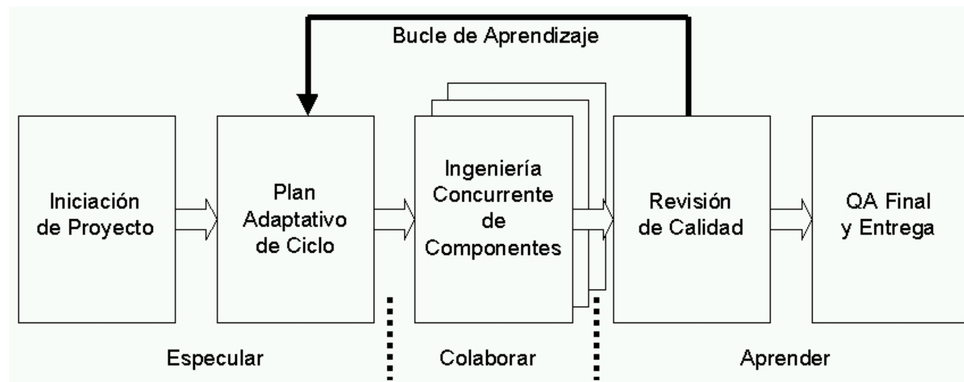
Se ha elaborado en particular la combinación de DSDM con XP y se ha llamado a esta mixtura EnterpriseXP, término acuñado por Mike Griffiths de Quadrus Developments (http://www.enterprisexp.org). Se atribuye a Kent Beck haber afirmado que la comunidad de DSDM ha construido una imagen corporativa mejor que la del mundo XP y que sería conveniente aprender de esa experiencia. También hay documentos conjuntos de DSDM y Rational, con participación de Jennifer Stapleton, que demuestran la compatibilidad del modelo DSDM con RUP, a despecho de sus fuertes diferencias terminológicas.

En el sitio del DSDM Consortium hay documentos específicos sobre la convivencia de la metodología con Microsoft Solutions Framework; se tratará detalladamente el particular en el capítulo sobre MSF y los MAs (pág. 55). También hay casos de éxito

(particularmente el de Fujitsu European Repair Centre) en que se emplearon Visual Basic como lenguaje, SQL Server como base de datos y Windows como plataforma de desarrollo e implementación (http://www.dsdm.org). Los métodos y manuales completos de DSDM sólo están disponibles para miembros del consorcio; la membrecía involucra el pago de un canon que oscila entre 550 € y 4400 € anuales.

Adaptive Software Development

James Highsmith III, consultor de Cutter Consortium, desarrolló ASD hacia el año 2000 con la intención primaria de ofrecer una alternativa a la idea, propia de CMM Nivel 5, de que la optimización es la única solución para problemas de complejidad creciente. Este método ágil pretende abrir una tercera vía entre el “desarrollo monumental de software” y el “desarrollo accidental”, o entre la burocracia y la adhocracia. Deberíamos buscar más bien, afirma Highsmith, “el rigor estrictamente necesario”; para ello hay que situarse en coordenadas apenas un poco fuera del caos y ejercer menos control que el que se cree necesario [Hig00a].



Fases del ciclo de vida de ASD, basado en Highsmith [Hig00a: 84]

La estrategia entera se basa en el concepto de emergencia, una propiedad de los sistemas adaptativos complejos que describe la forma en que la interacción de las partes genera una propiedad que no puede ser explicada en función de los componentes individuales. El pensador de quien Highsmith toma estas ideas es John Holland, el creador del algoritmo genético y probablemente el investigador actual más importante en materia de procesos emergentes [Hol95]. Holland se pregunta, entre otras cosas, cómo hace un macro-sistema extremadamente complejo, no controlado de arriba hacia abajo en todas las variables intervinientes (como por ejemplo la ciudad de Nueva York o la Web) para mantenerse funcionando en un aparente equilibrio sin colapsar.

La respuesta, que tiene que ver con la auto-organización, la adaptación al cambio y el orden que emerge de la interacción entre las partes, remite a examinar analogías con los sistemas adaptativos complejos por excelencia, esto es: los organismos vivientes (o sus análogos digitales, como las redes neuronales auto-organizativas de Teuvo Kohonen y los autómatas celulares desde Von Neumann a Stephen Wolfram). Para Highsmith, los proyectos de software son sistemas adaptativos complejos y la optimización no hace más que sofocar la emergencia necesaria para afrontar el cambio. Llevando más allá la analogía, Highsmith interpreta la organización empresarial que emprende un desarrollo como si fuera un ambiente, sus miembros como agentes y el producto como el resultado

emergente de relaciones de competencia y cooperación. En los sistemas complejos no es aplicable el análisis, porque no puede *deducirse* el comportamiento del todo a partir de la conducta de las partes, ni sumarse las propiedades individuales para determinar las características del conjunto: el oxígeno es combustible, el hidrógeno también, pero cuando se combinan se obtiene agua, la cual no tiene esa propiedad.

Highsmith indaga además la economía del retorno creciente según Brian Arthur. Esta economía se caracteriza por la alta velocidad y la elevada tasa de cambio. Velocidad y cambio introducen una complejidad que no puede ser manipulada por las estrategias convencionales. En condiciones de complejidad el mercado es formalmente impredecible y el proceso de desarrollo deviene imposible de planificar bajo el supuesto de que después se tendrá control del proceso. El razonamiento simple de causa y efecto no puede dar cuenta de la situación y mucho menos controlarla. En este contexto, los sistemas adaptativos complejos suministran los conceptos requeridos: agentes autónomos que compiten y cooperan, los ambientes mutables y la emergencia. Tomando como punto de partida estas ideas, Highsmith elabora un modelo de gestión que llama Modelo de Liderazgo-Colaboración Adaptativo (L-C), el cual tiene puntos en común con el modelado basado en agentes autónomos; en este modelo se considera que la adaptabilidad no puede ser comandada, sino que debe ser nutrida: nutriendo de conducta adaptativa a cada agente, el sistema global deviene adaptativo.

ASD presupone que las necesidades del cliente son siempre cambiantes. La iniciación de un proyecto involucra definir una misión para él, determinar las características y las fechas y descomponer el proyecto en una serie de pasos individuales, cada uno de los cuales puede abarcar entre cuatro y ocho semanas. Los pasos iniciales deben verificar el alcance del proyecto; los tardíos tienen que ver con el diseño de una arquitectura, la construcción del código, la ejecución de las pruebas finales y el despliegue.

Aspectos claves de ASD son:

1. Un conjunto no estándar de “artefactos de misión” (documentos para tí y para mí), incluyendo una visión del proyecto, una hoja de datos, un perfil de misión del producto y un esquema de su especificación
2. Un ciclo de vida, inherentemente iterativo.
3. Cajas de tiempo, con ciclos cortos de entrega orientados por riesgo.

Un ciclo de vida es una iteración; este ciclo se basa en componentes y no en tareas, es limitado en el tiempo, orientado por riesgos y tolerante al cambio. Que se base en componentes implica concentrarse en el desarrollo de software que trabaje, construyendo el sistema pieza por pieza. En este paradigma, el cambio es bienvenido y necesario, pues se concibe como la oportunidad de aprender y ganar así una ventaja competitiva; de ningún modo es algo que pueda ir en detrimento del proceso y sus resultados.

Highsmith piensa que los procesos rigurosos (repetibles, visibles, medibles) son encomiables porque proporcionan estabilidad en un entorno complejo, pero muchos procesos en el desarrollo (por ejemplo, el diseño del proyecto) deberían ser flexibles. La clave para mantener el control radica en los “estados de trabajo” (la colección de los productos de trabajo) y no en el flujo de trabajo (*workflow*). Demasiado rigor, por otra parte, acarrea *rigor mortis,* el cual impide cambiar el producto cuando se introducen las inevitables modificaciones. En la moderna teoría económica del retorno creciente, ser

capaz de adaptarse es significativamente más importante que ser capaz de optimizar [Hig00a].

La idea subyacente a ASD (y de ahí su particularidad) radica en que no proporciona un método para el desarrollo de software sino que más bien suministra la forma de implementar una cultura adaptativa en la empresa, con capacidad para reconocer que la incertidumbre y el cambio son el estado natural. El problema inicial es que la empresa *no sabe que no sabe,* y por tal razón debe aprender. Los cuatro objetivos de este proceso de aprendizaje son entonces:

1. Prestar soporte a una cultura adaptativa o un conjunto mental para que se espere cambio e incertidumbre y no se tenga una falsa expectativa de orden.
2. Introducir marcos de referencia para orientar el proceso iterativo de gestión del cambio.
3. Establecer la colaboración y la interacción de la gente en tres niveles: interpersonal, cultural y estructural.
4. Agregar rigor y disciplina a una estrategia RAD, haciéndola escalable a la complejidad de los emprendimientos de la vida real.

ASD se concentra más en los componentes que en las tareas; en la práctica, esto se traduce en ocuparse más de la calidad que en los procesos usados para producir un resultado. En los ciclos adaptativos de la fase de Colaboración, el planeamiento es parte del proceso iterativo, y las definiciones de los componentes se refinan continuamente. La base para los ciclos posteriores (el bucle de Aprendizaje) se obtiene a través de repetidas revisiones de calidad con presencia del cliente como experto, constituyendo un grupo de foco de cliente. Esto ocurre solamente al final de las fases, por lo que la presencia del cliente se suplementa con sesiones de desarrollo conjunto de aplicaciones (JAD). Hemos visto que una sesión JAD, común en el antiguo RAD, es un taller en el que programadores y representantes del cliente se encuentran para discutir rasgos del producto en términos no técnicos, sino de negocios.

El modelo de Highsmith es, naturalmente, complementario a cualquier concepción dinámica del método; no podría ser otra cosa que adaptable, después de todo, y por ello admite y promueve integración con otros modelos y marcos. Un estudio de Dirk Riehle [Rie00] compara ASD con XP, encontrando similitudes y diferencias de principio que pueden conciliarse con relativa facilidad, al lado de otras variables que son incompatibles. La actitud de ambos métodos frente a la redundancia de código, por ejemplo, es distinta; en XP se debe hacer todo "una vez y sólo una vez", mientras que en ASD la redundancia puede ser un subproducto táctico inevitable en un ambiente competitivo y debe aceptarse en tanto el producto sea "suficientemente bueno". En materia de técnicas, ASD las considera importantes pero no más que eso; para XP, en cambio, los patrones y la refactorización son balas de plata [Bro87].

Hay ausencia de estudios de casos del método adaptativo, aunque las referencias literarias a sus principios son abundantes. Como ASD no constituye un método de ingeniería de ciclo de vida sino una visión cultural o una epistemología, no califica como framework suficiente para articular un proyecto. Más visible es la participación de Highsmith en el respetado Cutter Consortium, del cual es director del Agile Project Management Advisory Service. Entre las empresas que han requerido consultoría adaptativa se cuentan

AS Bank de Nueva Zelanda, CNET, GlaxoSmithKline, Landmark, Nextel, Nike, Phoenix International Health, Thoughworks y Microsoft. Los consultores que se vinculan a los lineamientos ágiles de Cutter Consortium, por su parte, son innumerables. Jim Highsmith es citado en un epígrafe referido a los métodos ágiles en la documentación de Microsoft Solutions Framework [MS03].

Agile Modeling

Agile Modeling (AM) fue propuesto por Scott Ambler [Amb02a] no tanto como un método ágil cerrado en sí mismo, sino como complemento de otras metodologías, sean éstas ágiles o convencionales. Ambler recomienda su uso con XP, Microsoft Solutions Framework, RUP o EUP. En el caso de XP y MSF los practicantes podrían definir mejor los procesos de modelado que en ellos faltan, y en el caso de RUP y EUP el modelado ágil permite hacer más ligeros los procesos que ya usan. AM es una estrategia de modelado (de clases, de datos, de procesos) pensada para contrarrestar la sospecha de que los métodos ágiles no modelan y no documentan. Se lo podría definir como un proceso de software basado en prácticas cuyo objetivo es orientar el modelado de una manera efectiva y ágil.

Los principales objetivos de AM son:

1. Definir y mostrar de qué manera se debe poner en práctica una colección de valores, principios y prácticas que conducen al modelado de peso ligero.
2. Enfrentar el problema de la aplicación de técnicas de modelado en procesos de desarrollo ágiles.
3. Enfrentar el problema de la aplicación de las técnicas de modelado independientemente del proceso de software que se utilice.

Los valores de AM incluyen a los de XP: comunicación, simplicidad, *feedback* y coraje, añadiendo humildad. Una de las mejores caracterizaciones de los principios subyacentes a AM está en la definición de sus alcances:

1. AM es una actitud, no un proceso prescriptivo. Comprende una colección de valores a los que los modeladores ágiles adhieren, principios en los que creen y prácticas que aplican. Describe un estilo de modelado; no es un recetario de cocina.
2. AM es suplemento de otros métodos. El primer foco es el modelado y el segundo la documentación.
3. AM es una tarea de conjunto de los participantes. No hay "yo" en AM.
4. La prioridad es la efectividad. AM ayuda a crear un modelo o proceso cuando se tiene un propósito claro y se comprenden las necesidades de la audiencia; contribuye a aplicar los artefactos correctos para afrontar la situación inmediata y a crear los modelos más simples que sea posible.
5. AM es algo que funciona en la práctica, no una teoría académica. Las prácticas han sido discutidas desde 2001 en comunidad (http://www.agilemodeling.com/feedback.htm).
6. AM no es una bala de plata.
7. AM es para el programador promedio, pero no reemplaza a la gente competente.

8. AM no es un ataque a la documentación. La documentación debe ser mínima y relevante.
9. AM no es un ataque a las herramientas CASE.
10. AM no es para cualquiera.

Los principios de AM especificados por Ambler [Amb02a] incluyen:

1. **Presuponer simplicidad**. La solución más simple es la mejor.
2. **El contenido es más importante que la representación**. Pueden ser notas, pizarras o documentos formales. Lo que importa no es el soporte físico o la técnica de representación, sino el contenido.
3. **Abrazar el cambio**. Aceptar que los requerimientos cambian.
4. **Habilitar el esfuerzo siguiente**. Garantizar que el sistema es suficientemente robusto para admitir mejoras ulteriores; debe ser un objetivo, pero no el primordial.
5. **Todo el mundo puede aprender de algún otro**. Reconocer que nunca se domina realmente algo.
6. **Cambio incremental**. No esperar hacerlo bien la primera vez.
7. **Conocer tus modelos**. Saber cuáles son sus fuerzas y sus debilidades.
8. **Adaptación local**. Producir sólo el modelo que resulte suficiente para el propósito.
9. **Maximizar la inversión del cliente**.
10. **Modelar con un propósito**. Si no se puede identificar para qué se está haciendo algo ¿para qué molestarse?
11. **Modelos múltiples**. Múltiples paradigmas en convivencia, según se requiera.
12. **Comunicación abierta y honesta**.
13. **Trabajo de calidad**.
14. **Realimentación rápida**. No esperar que sea demasiado tarde.
15. **El software es el objetivo primario**. Debe ser de alta calidad y coincidir con lo que el usuario espera.
16. **Viajar ligero de equipaje**. No crear más modelos de los necesarios.
17. **Trabajar con los instintos de la gente**.

Lo más concreto de AM es su rico conjunto de prácticas [Amb02b], cada una de las cuales se asocia a lineamientos decididamente narrativos, articulados con minuciosidad, pero muy lejos de los rigores del aparato cuantitativo de Evo:

1. Colaboración activa de los participantes.
2. Aplicación de estándares de modelado.
3. Aplicación adecuada de patrones de modelado.
4. Aplicación de los artefactos correctos.
5. Propiedad colectiva de todos los elementos.
6. Considerar la verificabilidad.
7. Crear diversos modelos en paralelo.
8. Crear contenido simple.
9. Diseñar modelos de manera simple.

10. Descartar los modelos temporarios.
11. Exhibir públicamente los modelos.
12. Formalizar modelos de contrato.
13. Iterar sobre otro artefacto.
14. Modelo en incrementos pequeños.
15. Modelar para comunicar.
16. Modelar para comprender.
17. Modelar con otros.
18. Poner a prueba con código.
19. Reutilizar los recursos existentes.
20. Actualizar sólo cuando duele.
21. Utilizar las herramientas más simples (CASE, o mejor pizarras, tarjetas, *post-its*).

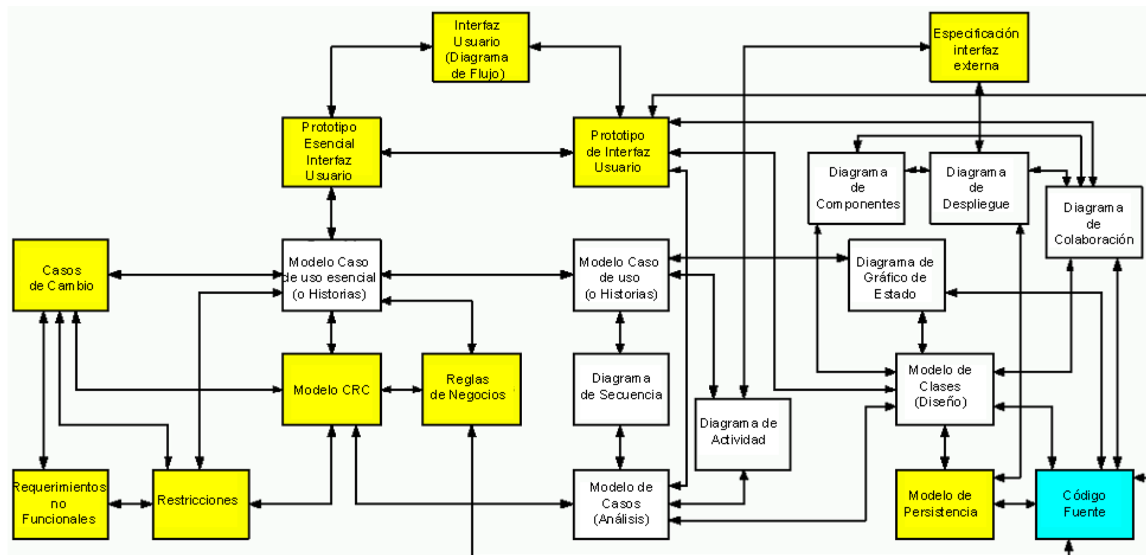


Diagrama de artefactos de Agile Modeling / EUP, basado en [Amb02c]

Como AM se debe usar como complemento de otras metodologías, nada se especifica sobre métodos de desarrollo, tamaño del equipo, roles, duración de iteraciones, trabajo distribuido y criticalidad, todo lo cual dependerá del método que se utilice. Los principios y prácticas que hemos citado puede dar impresión de puro sentido común, sólo que en variante caórdica; en realidad los aspectos atinentes a uso de herramientas de modelado y documentación, así como la articulación con metodologías específicas, las técnicas de bases de datos y el uso de artefactos están especificados cuidadosamente, como se puede constatar en el sitio de AM y en los textos más extensos de Ambler [Amb02a] [Amb02b] [Amb03] [Amb04].

Los diagramas de UML y los artefactos del Proceso Unificado, por ejemplo, han sido explorados en extremo detalle describiendo cómo debería ser su tratamiento en un proceso ágil EUP, regido por principios caórdicos. EUP es simplemente UP+AM. Se han documentado algunos estudios de casos de AM en proyectos de mediano calado [Amb02b]. Aún cuando uno no pretenda integrarse al bando de los MAs, puede que valga

la pena echar una mirada a ese material y considerar la razonabilidad de algunas de sus sugerencias.

Lean Development (LD) y Lean Software Development (LSD)

Lean Development (LD) es el método menos divulgado entre los reconocidamente importantes. La palabra "*lean*" significa magro, enjuto; en su sentido técnico apareció por primera vez en 1990 en el libro de James Womack *La Máquina que Cambió al Mundo* [WTR91]. LD, iniciado por Bob Charette [Cha01], se inspira en el éxito del proceso industrial Lean Manufacturing, bien conocido en la producción automotriz y en manufactura desde la década de 1980. Este proceso tiene como precepto la eliminación de residuos a través de la mejora constante, haciendo que el producto fluya a instancias del cliente para hacerlo lo más perfecto posible.

Los procesos a la manera americana corrían con sus máquinas al 100% de capacidad y mantenían inventarios gigantescos de productos y suministros; Toyota , en contra de la intuición, resultaba más eficiente manteniendo suministros en planta para un solo día, y produciendo sólo lo necesario para cubrir las órdenes pendientes. Esto es lo que se llama *Just in Time Production*. Con JIT se evita además que el inventario degrade o se torne obsoleto, o empiece a actuar como un freno para el cambio. Toyota implementaba además las técnicas innovadoras del Total Quality Management de Edward Deming, que sólo algunos matemáticos y empresarios de avanzada conocían en Estados Unidos. Hasta el día de hoy la foto de Deming en Toyota es más grande y esplendorosa que la del fundador, Toyoda Sakichi.

Otros aspectos esenciales de Lean Manufacturing son la relación participativa con el empleado y el trato que le brinda la compañía, así como una especificación de principios, disciplinas y métodos iterativos, adaptativos, auto-organizativos e interdependientes en un patrón de ciclos de corta duración que tiene algo más que un aire de familia con el patrón de procesos de los MAs (http://www.strategosinc.com/principles.htm). Existe unanimidad de intereses, consistencia de discurso y complementariedad entre las comunidades Lean de manufactura y desarrollo de software.

Mientras que otros MAs se concentran en el proceso de desarrollo, Charette sostenía que para ser verdaderamente ágil se debía conocer además el negocio de punta a punta. LD se inspira en doce valores centrados en estrategias de gestión [Hig02b]:

1. Satisfacer al cliente es la máxima prioridad.
2. Proporcionar siempre el mejor valor por la inversión.
3. El éxito depende de la activa participación del cliente.
4. Cada proyecto LD es un esfuerzo de equipo.
5. Todo se puede cambiar.
6. Soluciones de dominio, no puntos.
7. Completar, no construir.
8. Una solución al 80% hoy, en vez de una al 100% mañana.
9. El minimalismo es esencial.
10. La necesidad determina la tecnología.
11. El crecimiento del producto es el incremento de sus prestaciones, no de su tamaño.

12. Nunca empujes LD más allá de sus límites.

Dado que LD es más una filosofía de *management* que un proceso de desarrollo no hay mucho que decir del tamaño del equipo, la duración de las iteraciones, los roles o la naturaleza de sus etapas. Últimamente LD ha evolucionado como Lean Software Development (LSD); su figura de referencia es Mary Poppendieck [Pop01].

Uno de los sitios primordiales del modelo son las páginas consagradas a LSD que mantiene Darrell Norton [Nor04], donde se promueve el desarrollo del método aplicando el framework .NET de Microsoft. Norton ha reformulado los valores de Charette reduciéndolos a siete y suministrando más de veinte herramientas análogas a patrones organizacionales para su implementación en ingeniería de software. Los nuevos principios son:

1. Eliminar basura (las herramientas son *Seeing Waste, Value Stream Mapping*). Basura es todo lo que no agregue valor a un producto, desde la óptica del sistema de valores del cliente. Este principio equivale a la reducción del inventario en manufactura. El inventario del desarrollo de software es el conjunto de artefactos intermedios. Un estudio del Standish Group reveló que en un sistema típico, las prestaciones que se usan siempre suman el 7%, las que se usan a menudo el 13%, "algunas veces" el 16%, "raras veces" el 19% y "nunca" el 45%. Esto es un claro 80/20: el 80% del valor proviene del 20% de los rasgos. Concentrarse en el 20% útil es una aplicación del mismo principio que subyace a la idea de YAGNI.
2. Amplificar el conocimiento (*Feedback, Iterations, Synchronization, Set-based Development*). El desarrollo se considera un ejercicio de descubrimiento.
3. Decidir tan tarde como sea posible (*Options Thinking, The Last Responsible Moment, Making Decisions*). Las prácticas de desarrollo que proporcionan toma de decisiones tardías son efectivas en todos los dominios que involucran incertidumbre porque brindan una estrategia basada en opciones fundadas en la realidad, no en especulaciones. En un mercado que cambia, la decisión tardía, que mantiene las opciones abiertas, es más eficiente que un compromiso prematuro. En términos metodológicos, este principio se traduce también en la renuencia a planificarlo todo antes de comenzar. En un entorno cambiante, los requerimientos detallados corren el riesgo de estar equivocados o ser anacrónicos.
4. Entregar tan rápido como sea posible (*Pull Systems, Queueing Theory, Cost of Delay*). Se deben favorecer ciclos cortos de diseño → implementación → *feedback* → mejora. El cliente recibe lo que necesita hoy, no lo que necesitaba ayer.
5. Otorgar poder al equipo (*Self Determination, Motivation, Leadership, Expertise*). Los desarrolladores que mejor conocen los elementos de juicio son los que pueden tomar las decisiones más adecuadas.
6. Integridad incorporada (*Perceived Integrity, Conceptual Integrity, Refactoring, Testing*). La integridad conceptual significa que los conceptos del sistema trabajan como una totalidad armónica de arquitectura coherente. La investigación ha demostrado que la integridad viene con el liderazgo, la experiencia relevante, la comunicación efectiva y la disciplina saludable. Los procesos, los procedimientos y las medidas no son substitutos adecuados.

7. Ver la totalidad (*Measurements, Contracts*). Uno de los problemas más intratables del desarrollo de software convencional es que los expertos en áreas específicas (por ejemplo, bases de datos o GUIs) maximizan la corrección de la parte que les interesa, sin percibir la totalidad.

Otra preceptiva algo más amplia es la de Mary Poppendieck [Pop01], cuidadosamente decantadas del Lean Manufacturing y de Total Quality Management (TQM), que sólo coincide con la de Norton en algunos puntos:

1. Eliminar basura – Entre la basura se cuentan diagramas y modelos que no agregan valor al producto.
2. Minimizar inventario – Igualmente, suprimir artefactos tales como documentos de requerimiento y diseño.
3. Maximizar el flujo – Utilizar desarrollo iterativo.
4. Solicitar demanda – Soportar requerimientos flexibles.
5. Otorgar poder a los trabajadores.
6. Satisfacer los requerimientos del cliente – Trabajar junto a él, permitiéndole cambiar de ideas.
7. Hacerlo bien la primera vez – Verificar temprano y refactorizar cuando sea preciso.
8. Abolir la optimización local – Alcance de gestión flexible.
9. Asociarse con quienes suministran – Evitar relaciones de adversidad.
10. Crear una cultura de mejora continua.

Las herramientas, junto con el prolijo desarrollo de la metodología, se detallan en un texto de Mary y Tom Poppendieck [PP03], consistentemente encomiado por sus lectores.

Igual que Agile Modeling, que cubría sobre todo aspectos de modelado y documentación, LD y LSD han sido pensados como complemento de otros métodos, y no como una metodología excluyente a implementar en la empresa. LD prefiere concentrarse en las premisas y modelos derivados de Lean Production, que hoy constituyen lo que se conoce como el canon de la Escuela de Negocios de Harvard. Para las técnicas concretas de programación, LD promueve el uso de otros MAs que sean consistentes con su visión, como XP o sobre todo Scrum.

Aunque la formulación del método es relativamente reciente, la familiaridad de muchas empresas con los principios de Lean Production & Lean Manufacturing ha facilitado la penetración en el mercado de su análogo en ingeniería de software. LD se encuentra hoy en América del Norte en una situación similar a la de DSDM en Gran Bretaña, llegando al 7% entre los MAs a nivel mundial (algo menos que Crystal pero el doble que Scrum). Existen abundantes casos de éxito documentados empleando LD y LSD, la mayoría en Canadá. Algunos de ellos son los de Canadian Pacific Railway, Autodesk y PowerEx Corporation. Se ha aplicado prácticamente a todo el espectro de la industria.

Microsoft Solutions Framework y los Métodos Agiles

Los métodos ágiles de desarrollo tienen una larga tradición en las prácticas de Microsoft. Uno de los textos clásicos de RAD, *Rapid Development* de Steve McConnell [McC96], se remonta a una tradición más temprana que los MAs contemporáneos; al igual que

éstos, reconoce como "mejores prácticas" al modelo de ciclo de vida evolutivo, a los encuentros y talleres de equipo, las revisiones frecuentes, el diseño para el cambio, la entrega evolutiva, la reutilización, el prototipado evolutivo, la comunicación intensa, el desarrollo iterativo e incremental.

McConnell cita con generosidad el pensamiento de algunos de los precursores de los MAs: Barry Boehm, Frederick Brooks, Tom DeMarco, Harlan Mills. Cada vez que se utiliza la expresión "evolutivo" en RAD, se lo hace en el sentido que le imprimiera Tom Gilb, quien viene desarrollando el método Evo desde hace más de cuarenta años. El mismo McConnell, autor de *Code Complete* [McC93], es acreditado como una influencia importante en la literatura ágil de la actualidad [Lar04], aunque sus métodos carezcan de personalidad al lado de la estridente idiosincracia de algunos MAs. Algunas prácticas de RAD, sin embargo, divergen sensiblemente de lo que hoy se consideraría correcto en la comunidad ágil, como la recomendación de establecer metas fijas de antemano, contratar a terceros para realizar parte del código (*outsourcing*), trabajar en oficinas privadas u ofrecerse a permanecer horas extraordinarias.

Microsoft Solutions Framework es "un conjunto de principios, modelos, disciplinas, conceptos, lineamientos y prácticas probadas" elaborado por Microsoft. Como hemos entrevisto desde el epígrafe inicial, MSF se encuentra en estrecha comunión de principios con las metodologías ágiles, algunas de cuyas intuiciones y prácticas han sido anticipadas en las primeras versiones de su canon, hacia 1994. La documentación de MSF no deja sombra de dudas sobre esta concordancia; merece ser citada con amplitud y continuidad para apreciar el flujo de las palabras textuales y la intensidad de las referencias. Tras una cita del líder del movimiento ágil y ASD, Jim Highsmith, dice el documento principal de MSF 3.0:

> Las estrategias tradicionales de gestión de proyectos y los modelos de proceso en "cascada" presuponen un nivel de predictibilidad que no es tan común en proyectos de tecnología como puede serlo en otras industrias. A menudo, ni los productos ni los recursos para generarlos son bien entendidos y la exploración deviene parte del proyecto. Cuanto más busca una organización maximizar el impacto de negocios de la inversión tecnológica, más se aventura en nuevos territorios. Esta nueva base es inherentemente incierta y sujeta a cambios a medida que la exploración y la experimentación resultan en nuevas necesidades y métodos. Pretender o demandar certidumbre ante esta incertidumbre sería, en el mejor de los casos, irreal, y en el peor, disfuncional.
>
> MSF reconoce la naturaleza *caórdica* (una combinación de caos y orden, según lo acuñara Dee Hock, fundador y anterior CEO de Visa International) de los proyectos de tecnología. Toma como punto de partida el supuesto de que debe esperarse cambio continuo y de que es imposible aislar un proyecto de solución de estos cambios. Además de los cambios procedentes del exterior, MSF aconseja a los equipos que esperen cambios originados por los participantes e incluso por el propio equipo. Por ejemplo, admite que los requerimientos de un proyecto pueden ser difíciles de articular de antemano y que a menudo sufren modificaciones significativas a medida que los participantes van viendo sus posibilidades con mayor claridad.

> MSF ha diseñado tanto su Modelo de Equipo como su Modelo de Proceso para anticiparse al cambio y controlarlo. El Modelo de Equipo de MSF alienta la agilidad para hacer frente a nuevos cambios involucrando a todo el equipo en las decisiones fundamentales, asegurándose así que se exploran y revisan los elementos de juicio desde todas las perspectivas críticas. El Modelo de Proceso de MSF, a través de su estrategia iterativa en la construcción de productos del proyecto, suministra una imagen más clara del estado de los mismos en cada etapa sucesiva. El equipo puede identificar con mayor facilidad el impacto de cualquier cambio y lidiar con él efectivamente, minimizando los efectos colaterales negativos mientras optimiza los beneficios.
>
> En años recientes se han visto surgir estrategias específicas para desarrollar software que buscan maximizar el principio de agilidad y la preparación para el cambio. Compartiendo esta filosofía, MSF alienta la aplicación de esas estrategias donde resulte apropiado. […] Los métodos Ágiles, tales como Lean Development, eXtreme Programming y Adaptive Software Development, son estrategias de desarrollo de software que alientan prácticas que son adaptativas en lugar de predictivas, centradas en la gente y el equipo, iterativas, orientadas por prestaciones y entregas, de comunicación intensiva, y que requieren que el negocio se involucre en forma directa. Comparando esos atributos con los principios fundacionales de MSF, se encuentra que MSF y las metodologías ágiles están muy alineados tanto en los principios como en la práctica en ambientes que requieran alto grado de adaptabilidad [MS03].

Aunque MSF también se presta como marco para un desarrollo fuertemente documentado y a escala de misión crítica que requiere niveles más altos de estructura, como el que se articula en CMMI, PMBOK o ISO9000, sus disciplinas de ningún modo admiten la validez (o sustentan la vigencia) de modelos no iterativos o no incrementales.

La *Disciplina de Gestión de Riesgo* que se aplica a todos los miembros del *Modelo de Equipo* de MSF emplea el principio fundacional que literalmente se expresa como *permanecer ágil – esperar el cambio*. Esta ha sido una de las premisas de Jim Highsmith, quien tiene a Microsoft en un lugar prominente entre las empresas que han recurrido a su consultoría en materia de desarrollo adaptativo (ASD). También la utilización de la experiencia como oportunidad de aprendizaje está explícitamente en línea con las ideas de Highsmith. Los ocho principios fundacionales de MSF se asemejan a más de una de las declaraciones de principios que hemos visto expresadas en los manifiestos de los diversos MAs:

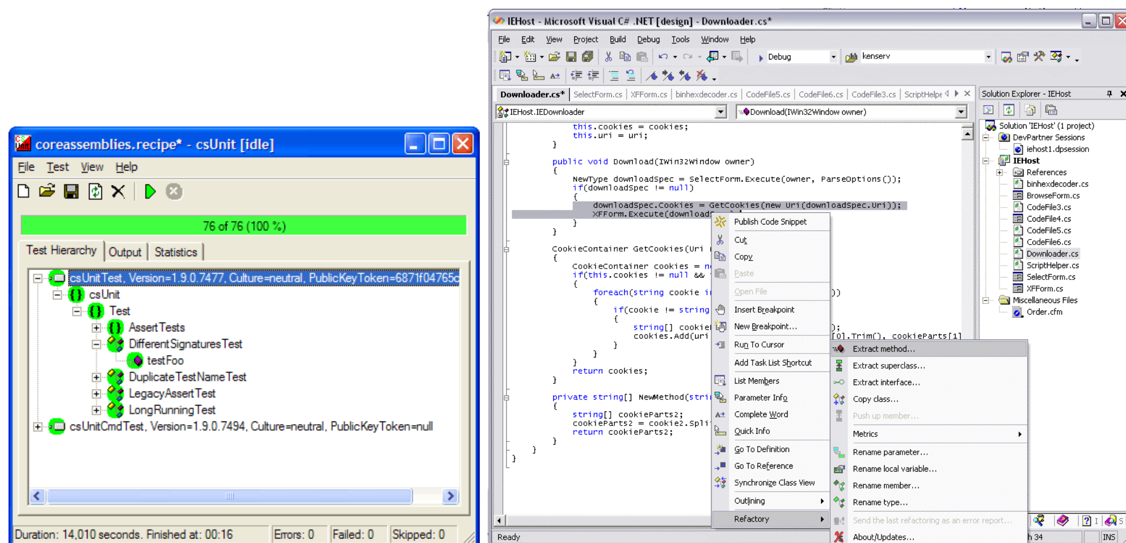
1. Alentar comunicaciones abiertas.
2. Trabajar hacia una visión compartida.
3. Otorgar poder a los miembros del equipo.
4. Establecer responsabilidad clara y compartida.
5. Concentrarse en la entrega de valor de negocios.
6. Permanecer ágil, esperar el cambio.
7. Invertir en calidad.
8. Aprender de todas las experiencias.

El primer principio, *Alentar la comunicación abierta*, utiliza como epígrafe para establecer el tono de sus ideas una referencia a la nueva edición de *The Mythical Man-Month* de Frederick Brooks [Bro95: 74-75], el ángel tutelar de todos los MAs y el disparador de las críticas al pensamiento de sentido común. Decía Brooks que este pensamiento (subyacente todavía en muchos usos de las *pivot-tables*) no titubearía en suscribir la idea de que nueve mujeres tardarían un mes en tener un hijo. Brooks ha sido uno de los primeros en estudiar las variables humanas en la producción de software, y su *MMM* es un libro de cabecera de todos los practicantes ágiles.

El segundo principio de MSF, *Trabajar hacia una visión compartida*, emplea para los mismos fines una cita de uno de los codificadores de la Programación Rápida, Steve McConnell [McC96]. *Permanecer ágil, esperar el cambio* comparte una vez más el pensamiento de Highsmith: "Los *managers* ágiles deben comprender que demandar certidumbre frente a la incertidumbre es disfuncional. Ellos deben establecer metas y restricciones que proporcionen límites dentro de los cuales puedan florecer la creatividad y la innovación".

En la *Disciplina de Management de Proyectos*, asimismo, se reconoce como uno de los cuerpos de conocimiento a los que se ha recurrido a Prince2 (***Pr****ojects* ***in*** ***C****ontrolled* ***E****nvironments*), un estándar de industria para *management* y control de proyectos. El *Modelo de Procesos de MSF*, asimismo, recomienda una estructura de proceso fundamentalmente iterativa similar a la de todos los MAs que han elaborado modelos de ciclo de vida [MS02a], y a lo largo de todo el marco se insiste en otro postulado común, el de la participación activa del cliente en todas las etapas del proceso.

Recíprocamente, muchos de los codificadores de MAs han procurado poner en claro sus vínculos con MSF, entendiendo que éste es uno de los *players* importantes en la arena metodológica, y que en estas prácticas es común que disciplinas de alto nivel se articulen y orquesten con metodologías más precisas, e incluso con más de una de ellas. Por ejemplo, Agile Modeling es, en las palabras de Scott Ambler, un complemento adecuado a la disciplina de MSF, en la misma medida en que es también armónico con RUP o XP.



csUnit probando unidad en un proyecto XP en C# (izq) – Refactorización con Xtreme Simplicity (der)

Mas allá de MSF, Ron Jeffries, uno de los padres de XP, mantiene en su sitio de MAs múltiples referencias al Framework .NET y a C#. Mucha gente ignora que uno de los "*three* extremos" que inventaron XP es el mismo que publicó recientemente *Extreme Programming Adventures in C#* [Jef04]. Incidentalmente, hay herramientas para trabajar en términos a la vez conformes a MSF y a las prácticas ágiles, como csUnit para .NET, un verificador de regresión del tipo que exigen los procesos de XP (figura), o NUnitAsp para Asp.NET [NV04]. Instrumentos de este tipo permiten implementar la práctica de prueba automática y diseño orientado por pruebas requeridos por los MAs. También hay disponibilidad de herramientas comerciales de refactorización para .NET (VB y C#), como Xtreme Simplicity (http://www.xtreme-simplicity.net) y .NET Refactoring (http://www.dotnetrefactoring.com), así como un proyecto de código abierto, Opnieuw (http://sourceforge.net/projects/opnieuw).

Tanto en los MAs como en la estrategia arquitectónica de Microsoft coinciden en su apreciación de los patrones. Toda la práctica de Patterns & Practices de Microsoft, igual que es el caso en Scrum y en XP, utiliza numerosos patrones organizativos y de diseño propuestos por Martin Fowler [MS04]. Por su parte, Tom Gilb considera que las prácticas metodológicas internas de Microsoft, desde hace ya mucho tiempo, aplican principios que son compatibles con los de su modelo de Evolutionary Project Management (Evo), al punto que sirven para ilustrarlos, uno por uno, en viñetas destacadas [Gilb97].

David Preedy (de Microsoft Consulting Services) y Paul Turner (miembro del DSDM Consortium) elaboraron los principios subyacentes a sus respectivos modelos y también encontraron convergencia y armonía entre DSDM y MSF [PT03]. En el área de aplicabilidad, MSF se caracteriza como un marco de desarrollo para una era caracterizada por tecnología cambiante y servicios distribuidos; de la misma manera, DSDM se aplica a los mismos escenarios, excluyendo situaciones de requerimientos fijos y tecnologías monolíticas.

La participación de los usuarios en el proyecto es considerada en parecidos términos en ambos marcos, siendo MSF un poco más permisivo respecto de su eventual no-participación. En ambos marcos se observan similares temperamentos en relación con la capacidad de decisión del equipo, la producción regular de entregables frecuentes, los criterios de aceptabilidad, el papel de los prototipos en el desarrollo iterativo, la posibilidad de volver atrás (línea temprana de base y congelamiento tardío en MSF, reversibilidad de cambios en DSDM) y el papel crítico de los verificadores durante todo el ciclo de vida.

Hay algunas diferencias menores en la denominación de los roles, que son 6 en MSF y 10 en DSDM, pero no es difícil conciliar sus especificaciones porque ambas contemplan complementos. Lo mismo vale para la prueba de control de calidad, aunque en DSDM no hay un rol específico para QA. Los hitos externos de MSF son cuatro y los de DSDM son cinco, pero su correspondencia es notable: el primero establece Visión & Alcance, Aprobación de la Especificación Funcional, Alcance Completo y Entrega; el segundo, Viabilidad, Estudio de Negocios, Modelo Funcional, Diseño & Construcción e Implementación.

En MSF se prioriza la gestión de riesgos, como una forma de tratar las áreas más riesgosas primero; en DSDM la prioridad se basa en el beneficio de negocios antes que en el riesgo. Pero son semejantes los criterios sobre el tamaño de los equipos, la

importancia que se concede a la entrega en fecha, la gestión de versiones con funcionalidad incremental, el desarrollo basado en componentes (en la versión 4.x de DSDM, no así en la 3), así como el papel que se otorga al usuario en los requerimientos [PT03]. En DSDM Consortium también se ha investigado y puesto a prueba la integración de DSDM, MSF y Prince2 en proyectos ágiles complejos. De la combinación de estos métodos y estándares ha dicho Paul Turner que es incluso superior a la suma de las partes [Tur03].

En suma, muchos de los principios de los MAs son consistentes no sólo con los marcos teóricos, sino con las prácticas de Microsoft, las cuales vienen empleando ideas adaptativas, desarrollos incrementales y metodologías de agilidad (algunas de ellas radicales) desde hace mucho tiempo. Se puede hacer entonces diseño y programación ágil en ambientes Microsoft, o en conformidad con MSF (ya sea en modalidad corporativa como en DSDM, o en modo *hacker* como en XP), sin desnaturalizar a ninguna de las partes.

Métodos y Patrones

Una búsqueda de "extreme programming" en Google devuelve hoy algo así como 433.000 punteros; "design patterns" retorna, por su parte, 982.000. Son grandes números, aún en contraste con los 3.940.000 de "object oriented", los 2.420.000 de "uml", los 1.720.000 de "visual c++" o los 4.510.000 de "visual basic", que vienen desarrollándose desde hace mucho más tiempo. Siendo la Programación Extrema en particular y los MAs en general dos de los tópicos más novedosos y expansivos en la actualidad, cabría preguntarse cuál es su relación con los patrones de diseño, que sin duda constituyen el otro tema de conversación dominante en tecnología de software.

Ante semejante pregunta no cabe una respuesta uniforme: los MAs de estilo *hacker*, como XP o Scrum, y en otro orden FDD, celebran a los patrones como una de sus herramientas cardinales, a tono con un paradigma de programación orientada a objetos; los MAs inclinados a una visión general del proceso, pensados como lecturas para el staff de *management* o candidatos a ser tenidos en cuenta en las metodologías corporativas, pocas veces o nunca se refieren a ellos. No hablan siquiera de programación concreta, en última instancia. La programación con patrones en ambientes XP se trata en infinitos sitios de la Web; una idea semejante en DSDM sería impensable. Esta situación corrobora que el concepto de MAs engloba dos niveles de análisis, dos lugares distintos en el ciclo de vida y dos mundos de *stakeholders* diferentes, por más que todos sus responsables se hayan unido alguna vez (y lo sigan haciendo) para elaborar el Manifiesto y promover la visión ágil en su conjunto.

En esta serie de estudios de Arquitectura de Software, los patrones se definen y estudian en un documento separado, por lo que obviamos aquí su historia y su definición. Hay diversas clases de patrones: de arquitectura, de diseño, organizacionales. En XP los patrones de diseño son la forma nativa de la refactorización, así como la orientación a objetos es la modalidad por defecto de su paradigma de programación. Highsmith estima que los patrones proporcionan a la programación actual lo que el diseño estructural de Larry Constantine y Ed Yourdon suministraban una generación antes: guías efectivas para la estructura de programas [Hig00b]. Highsmith piensa que los lineamientos

provistos por los patrones son, hoy por hoy, la técnica de programación más productiva y estructurada de que se dispone.

Ahora bien, si se piensa que la idea de patrones de la informática fue tomada de la arquitectura de edificios y ciudades (de donde procede también el concepto, a más alto nivel, de arquitectura de software), puede concluirse que hay patrones no solamente en la funcionalidad del código, sino, como decía Christopher Alexander [Ale77], en todas partes. Algunos de los que desarrollaron métodos percibieron lo mismo, y de eso trata el resto del capítulo: cuáles son los patrones propios de una metodología ágil.

En el mismo espíritu alexanderiano de Coplien [Cop95], que más abajo se examinará en detalle, Jim Highsmith propone reemplazar la idea de procesos por la de patrones en su Adaptive Management, un modelo anterior a la elaboración de su método ASD y a la firma del Manifiesto Ágil [Hig99]. A diferencia de la noción de proceso, que la imaginación vincula con circuitos cibernéticos de entrada-salida y control, los patrones proporcionan un marco para pensar antes que un conjunto de reglas para obedecer. En esa tesitura, Highsmith propone tres patrones de gestión de Liderazgo-Colaboración que llama *Encontrando el Equilibrio en el Filo del Caos*, *Encontrando el Equilibrio en el Filo del Tiempo*, y *Encontrando el Equilibrio en el Filo del Poder*.

- El primero sugiere el ejercicio del control justo para evitar precipitarse en el caos, pero no tan estrecho que obstaculice el juicio creativo. El caos es fácil: simplemente precipítese en él. La estabilidad es fácil: sólo tiene que seguir los pasos. El equilibrio, en cambio, es difícil y delicado.
- Encontrar el equilibrio en el filo del tiempo implica hacer una arquitectura no de las estructuras, sino del devenir. Si se concibe el mundo como estable y predecible, el cambio se verá como una aberración temporaria en la transformación de un estado al otro. Si se presume que todo es impredecible, sobrevendrá la inmovilidad. Los arquitectos del tiempo presuponen que todo cambia, pero en diferentes ciclos de tiempo. Separan las fuerzas estabilizadoras de las que desestabilizan y manejan el cambio en vez de ser manejados por él. Lo que hay que preguntarse en consecuencia no es "¿Cual es el mejor XYZ?", sino "¿Cuál es nuestra mejor estrategia cuando algo cambia?". Cuanto más turbulento el ambiente, más corto deberá ser el ciclo. La adaptación consiste en mejoras, migración e integración. Los arquitectos del tiempo no están inmovilizados en el presente; por el contrario, equilibran el aprendizaje del pasado con la anticipación del futuro y entienden cuál puede ser el manejo del tiempo más adecuado. Hay que aprender a crear un *timing* apropiado para manejar el tiempo y convertir un impedimento en un beneficio.
- La arquitectura del poder implica adaptabilidad. En opinión de Highsmith, los billones de dólares gastados en "*change management*" constituyen un despilfarro. Hay una sutil diferencia entre adaptarse y cambiar. Adaptarse involucra poner el foco en lo que pasa afuera (el ambiente, el mercado), en vez de dejarse llevar por las políticas internas. En tiempos turbulentos, esperar que el pináculo de la jerarquía de la empresa se digne a involucrarse es una receta para el fracaso. La adaptación necesita ocurrir en todos los niveles, lo que significa libre flujo de la información y toma de decisiones distribuida. Compartir el poder es la característica fundamental de las organizaciones adaptables.

En otra aplicación del concepto, Alistair Cockburn, experto reconocido en programación orientada a objetos y en casos de uso, desarrolló para Crystal un atractivo uso de los patrones. El argumento de Cockburn es que las formas tradicionales de organizar los artefactos, que en algunos casos son docenas, mezclan conceptos de factoría con propiedades del producto (visión, declaraciones de misión, prototipos conceptuales) e idealizan la asignación de roles a las personas. Las formas de expresar propiedades (diagramas HIPO, gráficos estructurales, diagramas de flujo, casos de uso, diagramas de clase) van y vienen con las modas y la tecnología. En las formas tradicionales, no se consideran ni las reglas culturales ni la situación ambiental.

Cockburn concibe su modelo alternativo como una factoría en la cual las reglas que describen la metodología se expresan mediante patrones. Estudiando la literatura de patrones, Cockburn encontró que ellos corresponden a dos grandes clases: propiedades y estrategias. Algunos elementos de CC pueden ser fácilmente tipificados: la Comunicación Osmótica y la Entrega Frecuente tienen que ver con propiedades, mientras que La Exploración de 360°, el Esqueleto Ambulante y la Rearquitectura Incremental son más bien estrategias. Se puede concebir entonces un modelo de procesos como un conjunto de patrones que expresan su naturaleza; de esa manera, el equipo de programación puede entender el modelo e involucrarse mejor [Coc02]. La idea no está desarrollada más allá de ese punto, pero parece ser un buen elemento para ponerse a pensar.

Podría sospecharse que el examen de Cockburn de la literatura de patrones no fue suficientemente amplia. No menciona, por ejemplo, los patrones organizacionales de James Coplien (de Lucent Technologies, ex-Bell Labs), que sistematizan la misma idea que Cockburn estaba tratando de articular [Cop95]. El principio de estos patrones es el mismo que subyace a las ideas de Christopher Alexander: hay propiedades y elementos que se repiten en diferentes ejemplares de un determinado dominio. Cada ejemplar se puede comprender como una organización particular, compuesta por miembros de un conjunto finito de patrones reutilizables. Un lenguaje de patrones consiste en un repertorio de patrones y reglas para organizarlos. Un patrón es también una instrucción que muestra cómo se puede usar una configuración una y otra vez para resolver determinado problema, dondequiera que el contexto lo haga relevante.

El lenguaje de patrones de Coplien expresa tanto procedimientos metodológicos (*IncrementalIntegation, WorkSplit*), como artefactos (*CRC-CardsAndRoles*), propiedades (*HolisticDiversity*) y circunstancias ambientales (*SmokeFilledRoom, CodeOwnership*). Los Patlets que complementan a los patrones son como patrones sumarios o de alto nivel y sirven también para encontrar los patrones que se necesitan. Los patrones de Coplien, por ende, permiten expresar la configuración de una metodología como una organización de patrones reutilizables, considerados como sus componentes. Idea poderosa, sin duda, que recién comienza a explorarse.

Algunos practicantes de Scrum de primera línea se han acostumbrado a pensar su propio marco como un conjunto o sistema de patrones organizacionales. Mike Beedle, Ken Schwaber, Jeff Sutherland y otros [BDS+98], por ejemplo, suplementaron algunos patrones organizacionales de Coplien (*Firewall, FormFollowsFunction*, *NamedStableBases*) y los del repositorio del sitio de patrones de organización de Lucent Technologies (http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?ProjectIndex)

con nuevas extensiones del lenguaje, a fin de suministrar una descripción total de Scrum orientada a patrones.

La vigencia de los patrones se percibe además a través de reformulaciones o casos especiales de algunas metodologías nativas, como es el caso de XP@Scrum y sobre todo de XBreed, la combinación de XP, Scrum y patrones ideada por Mike Beedle. XBreed suplanta el Juego de Planeamiento por Scrum, auspicia algo de YAGNI pero en pequeñas dosis, admite CRC para las historias simples pero recomienda casos de uso para las historias complicadas, define un rol de arquitecto e impulsa con entusiasmo, casi excluyentemente, un fuerte uso de patrones.

Como Craig McCormick [McC02] no parece haber leído tampoco a [BDS+98], el brillo de una idea muy parecida a las anteriores se empañó debido a un síndrome que Coplien llamaría *InvenciónIndependiente*. En efecto, McCormick sugiere considerar los MAs, al lado de las prácticas ortodoxas, como una especie de repositorio de patrones de proceso o componentes metodológicos (entregables, técnicas, flujos de proceso, etcétera), junto con lineamientos sobre la manera de ensamblarlos para configurar la metodología de un proyecto determinado. McCormick encuentra que más útil que embarcarse en el objetivo de forjar un método unificado para todos los escenarios, sería articular una serie de frameworks de proceso, una meta-metodología, basada en abstracciones de patrones de procesos reutilizables, fundamentados en la experiencia. Bajo ciertas condiciones de proyecto, Extreme Programming (o RUP, o DSDM, o CMMI) puede ser una instancia del framework. La idea ya existía, por cierto, pues si se lo piensa un poco es evidente que se trata de un patrón recurrente de razonamiento; pero resulta sugerente verla expresada de esta otra forma.

Agilidad, Caos y Complejidad

Muchas de las manifestaciones heterodoxas de los MAs tienen como fuente de inspiración los nuevos paradigmas epistemológicos cristalizados en las teorías del caos determinista de la década de 1980 y las teorías de la complejidad de los 90s. Esas teorías tratan de diversas clases de procesos, fenómenos y sistemas: autómatas celulares, redes booleanas aleatorias, redes neuronales, programación evolutiva, memética, dinámica no lineal, criticalidad auto-organizada, modelos basados en agentes autónomos, metaheurísticas, atractores extraños, gramáticas recursivas, series de Fibonacci, orden emergente, fractales. La cuestión merece tratarse en un estudio específico; aquí sólo brindaremos un puñado de indicios sobre esta peculiar convergencia.

- Algunos de los ideólogos de Scrum [ADM96] se inspiran en conceptos tales como filo del caos y control de procesos caóticos, tal como se los describe en el conocido libro *Chaos* de James Gleick. También afirman que Microsoft (igual que Scrum) implementa en sus prácticas de desarrollo una estrategia de caos controlado, aunque no proporcionan mucho detalle a su razonamiento. Los autores también se refieren al modelo de L. B. S. Racoon [Rac95] referido al caos y al ciclo de vida caótico.
- En MSF 3.0 hay, como se ha visto, un comentario sobre la naturaleza caórdica de los sistemas, en el sentido del concepto acuñado por Dee Hock [Hock00], quien llevara a la práctica este concepto en la gestión de su propia compañía, Visa. Hock sostiene que la ciencia de la complejidad, ligada a la comprensión de los sistemas

autocatalíticos, no lineales, complejos y adaptativos, habrá de ser la ciencia más importante del nuevo siglo. La visión caórdica de Hock (cuyas ideas sobre no linealidad tienen más de un punto en común con las de Fred Brooks) ha inspirado un movimiento caórdico de amplios alcances que va mucho más allá de las metodologías de desarrollo de software. El modelo de gestión Liderazgo-Colaboración de Jim Highsmith, por ejemplo, se funda en las ideas de Hock.

- En Scrum se habla de caos permanentemente. Haciendo honor al nombre, el sitio de Scrum está en http://www.controlchaos.com. La descripción de Craig Larman sobre Scrum [Lar04: 136] incluye como bibliografía de referencia para comprender procesos adaptativos y autoorganización los dos textos clásicos de John Holland, el padre del algoritmo genético, quien también ha ejercido influencia sobre Highsmith, como ya se ha visto. Jeff Sutherland [Sut01] ha señalado la analogía entre los saltos y discontinuidades de los procesos de desarrollo ágiles y el "equilibrio puntuado" de los procesos evolutivos complejos.
- En las páginas de su sitio empresarial y en su bibliografía, Jeff DeLuca establece la estrecha relación entre las premisas de FDD y las teorías del caos determinista y de la complejidad, vinculándolas con el pensamiento no lineal de Fred Brooks y con la concepción del control de caos de Jim Highsmith.
- En su presentación de Adaptive Software Development, Highsmith afirma que su práctica se basa en "estar en equilibrio en el filo del caos": proporcionar suficiente guía para evitar caer en el caos, pero no demasiada, para no suprimir la emergencia y la creatividad. Este modelo basado en la analogía entre empresas/equipos/proyectos y sistemas adaptativos complejos, es, por mucho, el más vinculado a las nuevas ciencias de la complejidad y el caos [Hig00a]. Un ejemplo de estos sistemas podría ser una bandada de pájaros que se sincroniza sin que exista un líder que ejerza control y determine su dirección; cada pájaro sigue reglas simples y locales, pero en la macro-escala la bandada exhibe orden y un claro comportamiento colectivo emergente. Estas ideas han inspirado un modelo de gestión basado en cooperación y competencia [Hig02b].
- Una visión semejante se presenta en el modelo de Mary y Tom Poppendieck de Lean Development [PP03], donde se considera el comportamiento de sistemas sociales de insectos como una manifestación de trabajo en equipo, soluciones adaptativas y capacidades emergentes. LD promueve un diseño semejante a los de XP y Scrum, enfatizando sus aspectos evolutivos.
- Duane Truex [TBT00] de la Universidad de Georgia en Atlanta, promotor de una especie de método ágil llamado Desarrollo Ametódico, se basa en conceptos de dinámica y emergencia organizacional. Su modelo, desarrollado tres años antes de la publicación del Manifiesto, anticipa ideas caórdicas que se encuentran implícitas en los MAs.
- En una entrevista a Kent Beck y Martin Fowler, el primero afirmó que mientras SEI se encuentra en el campo modernista, XP es más bien posmoderno. "Sus raíces filosóficas se encuentran en la teoría de los sistemas complejos" y sus capacidades se generan a través de procesos emergentes. [Beck01]. Es significativo que este juicio haya sido expresado por el creador de la Programación Extrema.

- Barry Boehm, promotor temprano de los MAs y creador de COCOMO y del modelo en espiral que prevalece en MSF [MS03], ha expresado que en el futuro la ingeniería de software deberá explorar los sistemas adaptativos, los agentes autónomos que se auto-organizan, los paisajes de adecuación evolutivos y el orden emergente. Cita al respecto el pensamiento de Stuart Kauffman, estudioso de los fenómenos adaptativos e inventor de las redes booleanas aleatorias, una variante de los sistemas adaptativos complejos a medio camino entre los autómatas celulares y los modelos de agentes [Boe02b].

En fin, hay más de una razón para elaborar en detalle las relaciones entre los MAs (incluyendo MSF) y el paradigma del caos y la complejidad. Lo haremos en otro documento de esta serie, donde también se expondrán los ricos fundamentos y los logros prácticos de esa epistemología.

Anti-agilidad: La crítica de los Métodos Ágiles

A pesar de que nadie en sus cabales se opondría a que se lo considere ágil, adaptable y sensible a los requerimientos del cliente, es natural que se generara oposición a los MAs y que en ocasiones la reacción frente a ellos adoptara una fuerte actitud combativa, para deleite de los cronistas que gustan hablar de "conflagraciones", "cruzadas" y "acometidas". Si los ágiles han sido implacables en su caricatura de las metodologías rigurosas, los adversarios que se han ganado no se han quedado atrás en sus réplicas. En esta contienda hay algunas observaciones de carácter técnico, pero la ironía prevalece. Lo que sigue es apenas una muestra.

Uno de los ataques más duros y tempranos contra los MAs proviene de una carta de Steven Rakitin a la revista *Computer*, bajo la rúbrica "El Manifiesto genera cinismo" [Rak01]. Refiriéndose a la estructura opositiva del Manifiesto, Rakitin expresa que en su experiencia, los elementos de la derecha (vinculados a la mentalidad planificadora) son esenciales, mientras que los de la izquierda sólo sirven como excusas para que los *hackers* escupan código irresponsablemente sin ningún cuidado por la disciplina de ingeniería. Así como hay una lectura literal y una estrecha del canon de CMM, Rakitin practica una "interpretación *hacker*" de propuestas de valor tales como "responder al cambio en vez de seguir un plan" y encuentra que es un generador de caos. La interpretación *hacker* de ese mandamiento sería algo así como: "¡Genial! Ahora tenemos una razón para evitar la planificación y codificar como nos dé la gana".

Más tarde, un par de libros de buena venta, *Questioning Extreme Programming* de Pete McBreen [McB02] y *Extreme Programming Refactored: The case against XP* de Matt Stephens y Doug Rosenberg [SR03] promovieron algunas dudas sobre XP pero no llegaron a constituirse como impugnaciones convincentes por su uso de evidencia circunstancial, su estilo redundante, su tono socarrón y su falta de método argumentativo. Que a último momento se redima a XP sugiriendo mejoras y refactorizaciones no ayuda a la causa de un dictamen anunciado desde sus títulos. Mucho más fundamentado es el estudio puramente crítico de Edward Berard, quien señala un buen número de "falacias" y medias verdades en el discurso ágil pero no logra, por alguna razón que se nos escapa, que su crítica levante vuelo [Ber03].

Gerold Keefer, de AVOCA GmbH, ha publicado ya dos versiones de un estudio titulado (con guiño a los expertos que reconozcan la alusión a Dijkstra) "Extreme Programming considerado dañino para el desarrollo confiable de software" [Kee03]. Algunos de los hechos tenidos en cuenta son que no sólo el primero, sino también el segundo proyecto de referencia de XP fueron cancelados; que las cifras que proporciona Beck sobre el costo del cambio metodológico no son tomadas en serio ni siquiera en el seno de la comunidad XP; que resultados de investigaciones en curso cuestionan la viabilidad de la programación orientada por pruebas[4], cuyo proceso puede consumir hasta el 30% o 40% de los recursos de un proyecto; y que reportes de proyectos de gran envergadura demuestran su insuperable problema de escalabilidad, que se manifiesta antes de lo que se cree.

Otros argumentos de Keefer apuntan a los altos costos y magros resultados de la programación en pares; a las malas prácticas resultantes de la negación a documentar; al retorno a la "programación de garage" y a la idea de que "el código se documenta a sí mismo" cuestionadas por Brooks [Bro75] hace treinta años; a la escasa verosimilitud de las historias de éxito de C3 y VCAPS, etcétera. Por añadidura, XP no se expide sobre proyectos con alcances, precios y fechas fijas, ni sobre requerimientos no funcionales como performance y seguridad, ni sobre ambientes de desarrollo físicamente distribuidos, ni (a pesar de su insistencia en reutilización y patrones) sobre integración de componentes listos para usar (COTS). A Keefer, por último, no le persuade la premisa de "hacer la cosa más simple que pueda funcionar", sino que prefiere la postura de Einstein: "Que sea lo más simple posible, pero no más simple que eso".

Inesperadamente, el promotor de RAD SteveMcConnell se ha opuesto con vehemencia a las ideas más radicales del movimiento ágil. Se ha manifestado contrario tanto a las estrategias técnicas como a las tácticas promocionales; dice McConnell: "Esta industria tiene una larga historia en cuanto a pegarse a cuanta moda sale", pero "luego se encuentra, cuando los bombos y platillos se enfrían, que estas tendencias no logran el éxito que sus evangelistas prometieron". Los cargos de McConnell contra XP son numerosos. En primer lugar, diferentes proyectos requieren distintos procesos; no se puede escribir software de aviónica para un contrato de defensa de la misma manera en que se escribe un sistema de inventario para un negocio de alquiler de videos.

Además, las reglas de XP son excesivamente rígidas; casi nadie aplica las 12 en su totalidad, y pocas de ellas son nuevas: Tom Gilb proponía lineamientos similares muchos años atrás. Dadas las premisas desaliñadas y las falsas publicidades, McConnell expresa que "el fenómeno parece un caso de hipnosis colectiva" [Baer03]. En diversas presentaciones públicas y en particular en SD West 2004, McConnell ha considerado la programación sin diseño previo, el uso atropellado de XP, la programación automática y la costumbre de llamar "ágil" a cualquier práctica como algunas de las peores ideas en construcción de software del año 2000.

---

[4] La programación orientada por pruebas (*test-driven development*) en su modalidad convencional es en efecto resistida en ciertos ambientes académicos. Dijkstra afirmaba que "la prueba de programas puede ser una manera muy efectiva para mostrar la presencia de *bugs*, pero es irremediablemente inadecuada para mostrar su ausencia" [Dij72]. En la academia se prefieren los métodos formales de demostración, a los que en el movimiento ágil rara vez se hace referencia.

Un miembro de Rational, John Smith [SmiS/f], opina que la terminología de XP encubre una complejidad no reconocida; mientras las palabras "artefacto" y "producto de trabajo" no figuran en los índices de sus libros canónicos, Smith cuenta más de 30 artefactos encubiertos: Historias, Restricciones, Tareas, Tareas técnicas, Pruebas de aceptación, Código de software, Entregas, Metáforas, Diseños, Documentos de diseño, Estándares de codificación, Unidades de prueba, Espacio de trabajo, Plan de entrega, Plan de iteración, Reportes y notas, Plan general y presupuesto, Reportes de progreso, Estimaciones de historias, Estimaciones de tareas, Defectos, Documentación adicional, Datos de prueba, Herramientas de prueba, Herramientas de gestión de código, Resultados de pruebas, *Spikes* (soluciones), Registros de tiempo de trabajo, Datos métricos, Resultados de seguimiento. La lista, dice Smith, es indicativa, no exhaustiva. En un proyecto pequeño, RUP demanda menos que eso. Al no tratar sus artefactos como tales, XP hace difícil operarlos de una manera disciplinada y pasa por ser más ligero de lo que en realidad es.

Hay otras dificultades también; a diferencia de RUP, en XP las actividades no están ni identificadas ni descriptas. Las "cosas que se hacen" están tratadas anecdóticamente, con diversos grados de prescripción y detalle. Finalmente hay prácticas en XP que decididamente escalan muy mal, como la refactorización, la propiedad colectiva del código, las metáforas en lugar de una arquitectura tangible y las entregas rápidas, que no tienen en cuenta siquiera cuestiones elementales de la logística de despliegue. Smith afirma no estar formulando una crítica, pero pone en duda incluso, por su falta de escalabilidad, que XP esté haciendo honor a su nombre. Por otra parte, en toda la industria se sabe que la refactorización *en general* no escala muy bien. Eso se manifiesta sobre todo en proyectos grandes en los que hay exigencias no funcionales que son clásicas "quebradoras de arquitectura", como ser procedimientos ad hoc para resolver problemas de performance, seguridad o tolerancia a fallas. En estos casos, como dice Barry Boehm [Boe02a], ninguna dosis de refactorización será capaz de armar a Humpty Dumpty nuevamente.

Mientras Ivar Jacobson [Jac02] y Grady Booch (ahora miembro de la Agile Alliance) han saludado la aparición de los métodos innovadores (siempre que se los combine con RUP), un gran patriarca del análisis orientado a objetos, Stephen Mellor [Mel03], ha cuestionado con acrimonia sus puntos oscuros demostrando, como al pasar, que UML ejecutable sería mejor. Mellor impugna la definición de "cliente" que hacen los MAs: no existe, dice, semejante cosa. En las empresas hay expertos en negocios y en tecnología, en mercadeo y en productos; los MAs no especifican quién de ellos debe estar en el sitio, cuándo y con qué frecuencia. Mellor también deplora las consignas estridentes de los agilistas y escenifica una parodia de ellas: "¡Programadores del mundo, uníos!", "¡Aplastemos la ortodoxia globalizadora!", "¡Basta de exportar puestos de trabajo en programación!", "Un espectro recorre Europa ¡El espectro es Extreme!"… Además se pregunta: en un sistema muy distribuido ¿quién es el usuario de un determinado dominio? ¿Qué cosa constituye una "historia simple"? ¿Cómo se mantiene una pieza para que sea reutilizable?

Como este no es un estudio evaluativo, dejamos las preguntas de Mellor en suspenso. El lector que participa en esta comunidad tal vez pueda ofrecer sus propias respuestas, o pensar en otros dilemas todavía más acuciantes.

Conclusiones

La tabla siguiente permite apreciar las convergencias y divergencias en la definición de los MAs de primera línea, así como resumir sus características claves, los nombres de los promotores iniciales y sus fechas de aparición. En la comunidad de los MAs, ninguno de los autores se identifica solamente con su propia criatura. Muchos teóricos y practicantes se mueven con relativa frecuencia de un método ágil a otro, y a veces se los encuentra defendiendo y perfeccionando modelos que no necesariamente son los suyos.

Metodología	Acrónimo	Creación	Tipo de modelo	Característica
Adaptive Software Development	ASD	Highsmith 2000	Prácticas + Ciclo de vida	Inspirado en sistemas adaptativos complejos
Agile Modeling	AM	Ambler 2002	"Metodología basada en la práctica"	Suministra modelado ágil a otros métodos
Crystal Methods	CM	Cockburn 1998	"Familia de metodologías"	MA con énfasis en modelo de ciclos
Agile RUP	dX	Booch, Martin, Newkirk 1998	Framework / Disciplina	XP dado vuelta con artefactos RUP
Dynamic Solutions Delivery Model	DSDM	Stapleton 1997	Framework / Modelo de ciclo de vida	Creado por 16 expertos en RAD
Evolutionary Project Management	Evo	Gilb 1976	Framework adaptativo	Primer método ágil existente
Extreme Programming	XP	Beck 1999	"Disciplina en prácticas de ingeniería"	Método ágil radical
Feature-driven development	FDD	De Luca & Coad 1998 Palmer & Felsing 2002	"Metodología"	Método ágil de diseño y construcción
Lean Development	LD	Charette 2001, Mary y Tom Poppendieck	"Forma de pensar" – Modelo logístico	Metodología basada en procesos productivos
Microsoft Solutions Framework	MSF	Microsoft 1994	Lineamientos, Disciplinas, Prácticas	Framework de desarrollo de soluciones
Rapid Development	RAD	McConnell 1996	Survey de técnicas y modelos	Selección de *best practices*, no método
Rational Unified Process	RUP	Kruchten 1996	Proceso unificado	Método (¿ágil?) con modelado
Scrum	Scrum	Sutherland 1994 - Schwaber 1995	"Proceso" (framework de management)	Complemento de otros métodos, ágiles o no

Se han omitido algunas corrientes cuyas prácticas reflejan parecidos ocasionales, así como la mayor parte de los modelos híbidos (XBreed, XP@Scrum, EUP, EnterpriseXP, DSDM+Prince2+MSF, IndustrialXP), aunque se ha dejado la referencia a Agile RUP (o dX) debido al valor connotativo que tiene la importancia de sus promotores. Los métodos ágiles híbridos y los dialectos industriales merecen un estudio separado. Aparte de los mencionados quedan por mencionar Grizzly de Ron Crocker [Cro04] y Dispersed eXtreme Programming (DXP), creado por Michael Kircher en Siemens.

Más allá de la búsqueda de coincidencias triviales (por ejemplo, la predilección por el número 12 en los principios asociados al Manifiesto, en Lean Development, en las prácticas del primer XP), queda por hacer un trabajo comparativo más sistemático, que ponga en claro cuáles son los elementos de juicio que definen la complementariedad, cuál es el grado de acatamiento de los principios ágiles en cada corriente, cuáles son los escenarios que claman por uno u otro método, o cómo engrana cada uno (por ejemplo) con CMMI y los métodos de escala industrial.

Tras la revisión que hemos hecho, surge de inmediato que entre los diversos modelos se presenta una regularidad inherente a la lógica de las fases, por distintas que sean las premisas, los ciclos, los detalles y los nombres. Dado que el ciclo de vida es de diferente longitud en cada caso, las fases de la tabla no son estrictamente proporcionales.

Los métodos que hemos examinado no son fáciles de comparar entre sí conforme a un pequeño conjunto de criterios. Algunos, como XP, han definido claramente sus procesos, mientras que otros, como Scrum, son bastante más difusos en ese sentido, limitándose a un conjunto de principios y valores. Lean Development también presenta más principios que prácticas; unos cuantos que ellos, como la satisfacción del cliente, están menos articulados que en Scrum, por ejemplo. Ciertos MAs, como FDD, no cubren todos los pasos del ciclo de vida, sino unos pocos de ellos. Varios métodos dejan librada toda la ingeniería concreta a algún otro método sin especificar.

Modelo	→ Fases →				
Modelo en Cascada	Especificar requerimientos	Especificar funciones	Especificar diseño	Codificación y Prueba	Prueba y validación
ASD	Especular	Colaborar	Aprender		
DSDM	Estudio de Viabilidad	Estudio del Negocio	Iteración del Modelo Funcional	Iteración Diseño & Versión	Implementación
Evo	Concepto	Análisis prelim. Req.	Diseño Arquitect.	Desarrollo iterativo	Entrega
Extreme Programming	Exploración	Planeamiento	Iteraciones hasta entrega	Productización	
FDD	Modelo general	Lista de Rasgos	Planear por Rasgo	Diseñar por Rasgo	Construir por Rasgo
Microsoft Sync & Stab.	Planeamiento	Desarrollo	Estabilización		
Proceso Unificado	Incepción	Elaboración	Construcción	Transición	
Scrum	Pre-juego: Planeamiento	Pre-juego: Montaje	Juego	Liberación	

Por más que exista una homología estructural en su tratamiento del proceso, se diría que en un primer análisis hay dos rangos o conjuntos distintos de MAs en la escala de complejidad. Por un lado están los MAs declarativos y programáticos como XP, Scrum, LD, AM y RAD; por el otro, las piezas mayores finamente elaboradas como Evo, DSDM y Crystal. En una posición intermedia estaría ASD, cuya naturaleza es muy peculiar. No calificaríamos a RUP como metodología ágil en plenitud, sino más bien como un conjunto enorme de herramientas y artefactos, al lado de unos (pocos) lineamientos de uso, que acaso están mejor articulados en AM que en la propia documentación nativa de RUP. Los recursos de RUP se pueden utilizar con mayor o menor conflicto en cualquier otra estrategia, sin necesidad de crear productos de trabajo nuevos una y otra vez. Como sea, habría que indagar en qué casos los métodos admiten hibridación por ser semejantes y complementarios en un mismo plano, y en qué otros escenarios lo hacen porque se refieren a distintos niveles de abstracción.

La intención de este trabajo ha sido describir la naturaleza de los MAs, establecer su estado actual y poner en claro sus relaciones con otras metodologías, en particular Microsoft Solutions Framework. Esperamos que la escala de tratamiento haya permitido

apreciar que el conjunto de los MAs es más heterogéneo de lo que sostiene la prensa de ambos bandos, que los estereotipos sobre su espíritu transgresor no condicen mucho con los rigores de algunas de sus formulaciones, y que la magnitud de sus novedades se atempera un poco cuando se comprueba que algunas de sus mejores ideas son también las más antiguas y seguras.

No ha sido parte del programa de este estudio la promoción o la crítica de los métodos; si se han documentado las protestas en su contra ha sido solamente para ilustrar el contexto. No se ha buscado tampoco marcar su contraste o su afinidad con los métodos pesados; sólo se ha señalado su complementariedad allí donde se la ha hecho explícita. Un ejercicio pendiente sería tomar el modelo de ciclos de cada método particular, o sus artefactos, o su definición de los equipos de trabajo, y mapear cada elemento de juicio contra los de MSF, porque hasta hoy esto se ha hecho solamente a propósito de DSDM [PT03] o, a nivel de código, con XP [Jef04].

Esta operación podría establecer correspondencias encubiertas por diferencias meramente terminológicas y señalar los puntos de inflexión para el uso de los MAs en las disciplinas de MSF. También habría que considerar combinaciones múltiples. No sería insensato proponer que MSF se utilice como marco general, Planguage como lenguaje de especificación de requerimiento, Scrum (con sus patrones organizacionales) como método de management, XP (con patrones de diseño, programación guiada por pruebas y refactorización) como metodología de desarrollo, RUP como abastecedor de artefactos, ASD como cultura empresarial y (¿por qué no?) CMM como metodología de evaluación de madurez.

Hay mucho por hacer en la comparación de los métodos; los *surveys* disponibles en general han tratado sólo pequeños conjuntos de ellos, sin que pueda visualizarse todavía una correlación sistemática sobre una muestra más amplia y menos sesgada. También hay tareas por hacer en la coordinación de los MAs con un conjunto de técnicas, prácticas y frameworks que guardan con ellos relaciones diferenciales: metodologías basadas en arquitectura, programación guiada por rasgos y por pruebas, métodos formales de verificación, lenguajes de descripción arquitectónica, modelos de línea de productos, patrones de diseño y organizacionales, refactorización.

Como lo testimonian los acercamientos desde los cuarteles de UML/RUP, o la reciente aceptación de muchos de sus principios por parte de los promotores de CMMI o ISO9000, o la prisa por integrarlos a otros métodos, o por encontrar términos equidistantes, los MAs, al lado de los patrones, han llegado para quedarse. Han democratizado también la discusión metodológica, antes restringida a unos pocos especialistas corporativos y a clubes organizacionales de acceso circunscripto. Se puede disentir con muchos de ellos, pero cabe reconocer su obra de transformación: de ahora en adelante, la metodología no volverá a ser la misma.

© Carlos Reynoso, 2004

Vínculos ágiles

Existen innumerables sitios dedicados a MAs, más o menos permanentes. La disponibilidad de documentos y discusiones sobre esos métodos es abrumadora. Aquí sólo hemos consignado los sitios esenciales, activos en Abril de 2004.

http://agilemanifesto.org/ – El Manifiesto Ágil.

http://alistair.cockburn.us/crystal/aboutmethodologies.html – Página inicial de Crystal Methodologies, de Alistair Cockburn.

http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap – Materiales de XP en el sitio de Ward Cunnimgham, creador del Wiki Wiki Web.

http://comunit.sourceforge.net – Página de COMUnit en Source Forge. Herramienta para prueba de unidad y prueba funcional requeridas por XP para programas en Visual Basic .NET.

http://dotnetjunkies.com/WebLog/darrell.norton/articles/4306.aspx – Weblog consagrado a Lean Software Development (LSD) con abundantes referencias a tecnología .NET y vínculos adicionales.

http://dotnetrefactoring.com – Herramientas de refactorización para .NET.

http://industrialxp.org – Páginas de IndustrialXP (IXP), de Industrial Logic.

http://jeffsutherland.com – Sitio de Jeff Sutherland, con vínculos a Scrum y artículos sobre tecnologías ligadas a XML y SOAP.

http://members.aol.com/humansandt/crystal/clear – Descripción de Crystal Clear.

http://name.case.unibz.it – Sitio de NAME (Network for Agile Methodologies Experience) de Italia. Contiene vínculo a csUnit.

http://www.nunit.org – Herramienta de desarrollo orientado a prueba y refactorización para .NET.

http://sourceforge.net/projects/nunit – Herramienta de prueba de unidad para lenguajes del framework .NET.

http://sourceforge.net/projects/opnieuw – Proyecto de código abierto para refactorización en lenguaje C#.

http://www.adaptivesd.com – Página de Jim Highsmith y Adaptive Software Development.

http://www.agilealliance.com – Numerosos artículos específicos de MAs y vínculos.

http://www.agilemodeling.com – Sitio oficial de Agile Modeling de Scott Ambler.

http://www.controlchaos.com – Sitio de Ken Schwaber. Referencias a Scrum y otros métodos.

http://www.csunit.org/index.php – Sitio de csUnit, una herramienta de control para VisualStudio .NET conforme a las prácticas de prueba automatizada de unidades en XP.

http://www.cutter.com – Página de Cutter Consortium – Incluye publicaciones especializadas en MAs.

http://www.dotnetrefactoring.com – Vínculo a C# Refactoring Tool, herramienta comercial de refactorización para .NET.

http://www.dsdm.org – Sitio de DSDM. Hay documentos específicos de comparación e integración con Microsoft Solutions Framework.

http://www.enterprisexp.org – Página de Enterprise XP en Quadrus Developments.

http://www.extremeprogramming.org – Página semi-oficial de XP.

http://www.gilb.com – Es el sitio de Tom Gilb, creador de Evo. Dispone de una buena cantidad de documentos, gráficas y glosarios, incluyendo el manuscrito de 600 páginas de Competitive Engineering [Gilb03a].

http://www.iturls.com/English/SoftwareEngineering/SE_Agile.asp – Sitio de IT Source sobre ingeniería de software ágil. Innumerables artículos y vínculos.

http://www.iturls.com/English/SoftwareEngineering/SE_fop.asp – Sitio de IT Source sobre Feature Oriented Programming y FDD, con nutrido repertorio de vínculos relevantes.

http://www.jimhighsmith.com – Páginas de Jim Highsmith y Adaptive Software Development.

http://www.mapnp.org/library/quality/tqm/tqm.htm – Vínculos a información sobre Total Quality Management (TQM).

http://www.nebulon.com/fdd/index.html – Sitio de FDD en la empresa Nebulon de Jeff DeLuca, "consultor, autor e inventor" que creó el método. Abundante información del modelo, diagramas y plantillas.

http://www.objectmentor.com – Sitio de la compañía de Robert C. Martin, signatario del Manifiesto, con vínculos a sitios de XP, C#, Visual Basic .NET, .NET Enterprise Solution Patterns.

http://www.poppendieck.com/ld.htm – Página principal de los creadores de Lean Development.

http://www.refactoring.com – Página de Martin Fowler consagrada a refactorización. En http://www.refactoring.com/catalog/index.html se encuentran las refactorizaciones más comunes.

http://www.vbunit.org – vbUnit 3, herramienta de prueba de unidad para Visual Basic.

http://www.xbreed.net/ – Página de XP+Scrum+Patrones por Mike Beedle

http://www.xprogramming.com/index.htm – Recursos comunitarios para eXtreme Programming, mantenido por Ron Jeffries. Hay materiales relacionados con .NET y C#.

http://www.xprogramming.com/software.htm – Software para desarrollo en XP sobre diversas plataformas, con sección de herramientas para .NET.

http://www.xtreme-simplicity.net – Herramientas para refactorización conformes a XP para Visual Basic.NET y C#.

Referencias bibliográficas

[Abr02] Pekka Abrahamsson. "Agile Software development methods: A minitutorial". VTT Technical Research Centre of Finland, http://www.vtt.fi/virtual/agile/seminar2002/Abrahamsson_agile_methods_minitutorial.pdf, 2002.

[ADM96] Advanced Development Methods. "Controlled chaos: Living on the Edge", http://www.controlchaos.com/ap.htm, 1996.

[Ale77] Christopher Alexander. *A pattern language*. Oxford University Press, 1977.

[Amb02a] Scott Ambler. *Agile Modeling: Effective practices for Extreme Programming and the Unified Process*. John Wiley & Sons, 2002.

[Amb02b] Scott Ambler. "Agile Modeling and the Unified Process". http://www.agilemodeling.com/essays/agileModelingRUP.htm, 2002.

[Amb02c] Scott Ambler. "The Enterprise Unified Process (EUP): Extending the RUP for real-world organizations". http://szakkoli.sch.bme.hu/ooffk/oookea/Scott_Ambler_EUPOverview.pdf, 2003.

[Amb03] Scott Ambler. "Artifacts for Agile Modeling: The UML and beyond", http://www.agilemodeling.com/essays/modelingTechniques.htm, 2003.

[Amb04] Scott Ambler. "Agile Modeling Essays", http://www.agilemodeling.com/essays.htm, 2004.

[ASR+02] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen y Juhani Warsta. "Agile Software Development Methods". *VTT Publications* 478, Universidad de Oulu, Suecia, 2002.

[Baer03] Martha Baer. "The new X-Men". *Wired 11.09*, Setiembre de 2003.

[Bat03] Don Batory. "A tutorial on Feature Oriented Programming and Product Lines". *Proceedings of the 25th International Conference on Software Engineering, ICSE'03*, 2003.

[BBB+01a] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunnimgham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas. "Agile Manifesto". http://agilemanifesto.org/, 2001.

[BBB+01b] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunnimgham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas. "Principles behind the Agile Manifesto". http://agilemanifesto.org/principles.html, 2001.

[BC89] Kent Beck y Ward Cunningham. "A laboratory for teaching Object-Oriented thinking". *OOPSLA'89 Conference Proceedings, SIGPLAN Notices*, 24(10), Octubre de 1989.

[BDS+98] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber y Jeff Sutherland. "SCRUM: A pattern language for hyperproductive software development". En N. Harrison, B. Foote, and H. Rohnert, eds., *Pattern Languages of Program Design*, vol. 4, pp. 637-651. Reading, Addison-Wesley, 1998.

[Beck99a] Kent Beck. *Extreme Programming Explained: Embrace Change*. Reading, Addison Wesley, 1999.

[Beck99b] Kent Beck, "RE: (OTUG) XP and Documentation". *Rational's Object Technology User Group Mailing List*, 23 de Marzo de 1999, 18:26:04 +0100.

[Beck01] Kent Beck. "Interview with Kent Beck and Martin Fowler". Addison-Wesley, http://www.awprofessional.com/articles/article.asp?p=20972&redir=1, 23 de Marzo de 2001.

[Ber03] Edward Berard. "Misconceptions of the Agile zealots". The Object Agency, http://www.svspin.org/Events/Presentations/MisconceptionsArticle20030827.pdf, 2003.

[BF00] Kent Beck y Martin Fowler. *Planning Extreme Programming*. Reading, Addison Wesley, 2000.

[BMP98] Grady Booch, Robert Martin y James Newkirk. *Object Oriented Analysis and Design With Applications*, 2ª edición, Addison-Wesley, 1998

[Boe02a] Barry Boehm. "Get ready for Agile Methods, with care". *Computer*, pp. 64-69, Enero de 2002.

[Boe02b] Barry Boehm. "The future of Software Engineering". USC-CSE, Boeing Presentation, 10 de Mayo de 1992.

[Bro75] Frederick Brooks. *The Mythical Man-Month*. Reading, Addison-Wesley.

[Bro87] Frederick Brooks. "No silver bullets – Essence and accidents of software engineering". *Computer*, pp. 10-19, Abril de 1987.

[Bro95] Frederick Brooks. *The Mythical Man-Month*. Boston, Addison-Wesley.

[BT75] Vic Basili y Joe Turner. "Iterative Enhancement: A Practical Technique for Software Development", *IEEE Transactions on Software Engineering*, 1(4), pp. 390-396, 1975.

[Cha01] Robert Charette, *The Foundation Series on Risk Management, Volume II: Foundations of Lean Development*, Cutter Consortium, Arlington, 2001.

[Cha04] Robert Charette. "The decision is in: Agile versus Heavy Methodologies". Cutter Consortium, *Executive Update*, 2(19), http://www.cutter.com/freestuff/apmupdate.html, 2004.

[CLC03] David Cohen, Mikael Lindvall y Patricia Costa. "Agile Software Development. A DACS State-of-the-Art Report", *DACS Report*, The University of Maryland, College Park, 2003.

[CLD00] Peter Coad, Eric Lefebvre y Jeff DeLuca. *Java modeling in color with UML: Enterprise components and process*. Prentice Hall, 2000.

[Coc97a] Alistair Cockburn. “Software development as Community Poetry Writing. Cognitive, cultural, sociological, human aspects of software development”. *Annual Meeting of the Central Ohio Chapter of the ACM*, http://alistair.cockburn.us/crystal/articles/sdacpw/softwaredevelopmentascommunitypoetrywriting.html, 1997.

[Coc97b] Alistair Cockburn. *Surviving Object-Oriented Projects*. Addison-Wesley.

[Coc00] Alistair Cockburn. “Balancing Lightness with Sufficiency”. http://alistair.cockburn.us/crystal/articles/blws/balancinglightnesswithsufficiency.html, Setiembre de 2000.

[Coc01] Alistair Cockburn. *Writing effective Use Cases*. Reading, Addison-Wesley.

[Coc02] Alistair Cockburn. “Crystal Clear. A human-powered methodology for small teams, including The Seven Properties of Effective Software Projects”. Borrador. *Humans and Technology*, versión del 27 de febrero de 2002.

[Cop01] Lee Copeland. “Developers approach Extreme Programming with caution”. *Computerworld*, p. 7, 22 de Octubre de 2001.

[Cro04] Ron Crocker. *Large-scale Agile Software Development*. Addison-Wesley, 2004.

[CS95a] Michael Cusumano y Richard Selby. *Microsoft secrets: How the world’s most powerful software company creates technology, shapes markets, and manages people*. The Free Press, 1995.

[CS95b] James O. Coplien y Douglas C. Schmidt, *Pattern Languages of Program Design* (*A Generative Development-Process Pattern Language)*, Reading, Addison Wesley, 1995.

[Dij72] Edsger Dijkstra. “The Humble Programmer,” 1972 ACM Turing Award Lecture, ACM Annual Conference, Boston, 14 de Agosto - *Communications of the ACM*, 15(10), pp. 859-866, Octubre de 1972.

[DS90] Peter DeGrace y Leslie Hulet Stahl. *Wicked problems, righteous solutions*. Englewood Cliffs, Yourdon Press, 1990.

[DS03] DSDM Consortium y Jennifer Stapleton. *DSDM: Business Focused Development*. 2ª edición, Addison-Wesley, 2003.

[ESPI96] ESPI Exchange. “Productivity claims for ISO 9000 ruled untrue”. Londres, European Software Process Improvement Foundation, p. 1, 1996.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke y Don Roberts. *Refactoring: Improving the design of existing code*. Addison Wesley, 1999.

[Fow01] Martin Fowler. “Is design dead?”. *XP2000 Proceedings*, http://www.martinfowler.com/articles/designDead.html, 2001.

[Gilb76] Tom Gilb. *Software metrics*. Chartwell-Bratt, 1976.

[Gilb88] Tom Gilb. *Principles of Software Engineering Management*. Reading, Addison-Wesley, 1988.

[Gilb97] Tom Gilb. *The Evolutionary Project Manager Handbook*. Evo Mini-Manuscript, http://www.ida.liu.se/~TDDB02/pkval01vt/EvoBook.pdf.

[Gilb99] Tom Gilb. "Evolutionary Project Management". Manuscrito inédito, material para el curso DC SPIN, junio de 1999.

[Gilb02] Tom Gilb. "10 Evolutionary Project Management (Evo) Principles". http://www.xs4all.nl/~nrm/EvoPrinc/EvoPrinciples.pdf, Abril de 2002.

[Gilb03a] Tom Gilb. *Competitive Engineering*. [Borrador] www.gilb.com, 2003.

[Gilb03b] Kai Gilb. *Evolutionary Project Management & Product Development (or The Whirlwind Manuscript)*, http://www.gilb.com/Download/EvoProjectMan.pdf, 2003.

[Gla01] Robert Glass. "Agile versus Traditional: Make love, not war". *Cutter IT Journal*, 14(12), diciembre de 2001.

[HesS/f] Wolfgang Hesse. "Dinosaur meets Archaeopteryx? Seven Theses on Rational's Unified Process (RUP)", http://citeseer.ist.psu.edu/571547.html, sin fecha.

[HHR+96] Pat Hall, Fiona Hovenden, Janet Rachel y Hugh Robinson. "Postmodern Software Development". *MCS Technical Reports*, 1996.

[Hig99] Jim Highsmith. "Adaptive Management Patterns". *Cutter IT Journal*, 12(9), Setiembre de 1999.

[Hig00a] Jim Highsmith. *Adaptive software development: A collaborative approach to managing complex systems*. Nueva York, Dorset House, 2000.

[Hig00b] Jim Highsmith. "Extreme Programming". *EBusiness Application Development*, Cutter Consortium, Febrero de 2000.

[Hig01] Jim Highsmith. "The Great Methodologies Debate. Part 1". *Cutter IT Journal*, 14(12), diciembre de 2001.

[Hig02a] Jim Highsmith. "What is Agile Software Development". *Crosstalk*, http://www.stsc.hill.af.mil/crosstalk/2002/10/highsmith.html, Octubre de 2002.

[Hig02b] Jim Highsmith. *Agile Software Development Ecosystems*. Boston, Addison Wesley, 2002.

[Hock00] Dee Hock. *Birth of the chaordic age*. San Francisco, Berrett-Koehler, 2000.

[Hol95] John Holland. *Hidden Order: How Adaptation builds Complexity*. Addison Wesley, 1995.

[HT00] Andrew Hunt y David Thomas. *The Pragmatic Programmer*. Reading, Addison Wesley, 2000.

[Jac02] Ivar Jacobson. "A resounding Yes to Agile Processes – But also to more". *The Rational Edge*, Marzo de 2002.

[JAH01] Ron Jeffries, Ann Anderson, Chet Hendrikson. *Extreme Programming Installed*. Upper Saddle River, Addison-Wesley, 2001.

[Jef04] Ron Jeffries. *Extreme Programming adventures in C#*. Redmond, Microsoft Press, 2004.

[Kee03] Gerold Keefer. "Extreme Programming considered harmful for reliable software development 2.0". AVOCA GmbH, Documento público, 2001.

[Kru95] Philippe Kruchten. "The 4+1 View Model of Architecture." *IEEE Software* 12(6), pp. 42-50, Noviembre de 1995.

[Kru00] Philippe Kruchten. *The Rational Unified Process: An introduction*. Addison-Wesley, 2000.

[Kru01] Philippe Kruchten. "Agility with the RUP". *Cutter IT Journal*, 14(12), diciembre de 2001.

[Lar03] Craig Larman. *UML y patrones*. Madrid, Prentice Hall.

[Lar04] Craig Larman. *Agile & Iterative Development. A Manager's Guide*. Reading, Addison-Wesley 2004.

[McB02] Pete McBreen. *Questioning Extreme Programming*. Addison Wesley, 2002.

[McC93] Steve McConnell. *Code complete*. Redmond, Microsoft Press, 1993.

[McC96] Steve McConnell. *Rapid Development. Taming wild software schedules*. Redmond, Microsoft Press, 1996.

[McC02] Craig McCormick. "Programming Extremism". *Upgrade*, 3(29, http://www.upgrade-cepis.org/issues/2002/2/up3-2McCormick.pdf, Abril de 2002. (Original en *Communications of the ACM*, vol. 44, Junio de 2001).

[MDL87] Harlan Mills, Michael Dyer y Richard Linger. "Cleanroom software engineering", *IEEE Software*, Setiembre de 1987.

[Mel03] Stephen Mellor. "What's wrong with Agile?", Project Technology, http://www.projtech.com/pdfs/shows/2003/wwwa.pdf, 2003.

[Mil56] George Miller. "The magical number seven, plus or minus two: Some limits on our capacity for processing information". *Psychology Review*, 63, pp. 81-97, 1956.

[MS02a] Microsoft Solutions Framework Process Model, v. 3.1. http://www.microsoft.com/technet/itsolutions/techguide/msf/msfpm31.mspx, 2002.

[MS02b] Microsoft Solutions Framework – MSF Project Management Discipline. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/tandp/innsol/default.asp, 2002.

[MS03] Microsoft Solutions Framework Version 3.0 Overview, http://www.microsoft.com/technet/itsolutions/techguide/msf/msfovrvw.mspx, 2003.

[MS04] Microsoft Patterns & Practices. "Organizaing patterns version 1.0.0", http://msdn.microsoft.com/architecture/patterns/02/default.aspx, 2004.

[MVD03] Tom Mens y Arie Van Deursen. "Refactoring: Emerging trends and open problems". http://homepages.cwi.nl/~arie/papers/refactoring/reface03.pdf, Octubre de 2003.

[NA99] Joe Nandhakumar y David Avison. "The fiction of methodological development: a field study of information systems development". *Information Technology & People*, 12(2), pp. 176-191, 1999.

[Nor04] Darrell Norton. "Lean Software Development Overview", http://dotnetjunkies.com/WebLog/darrell.norton/articles/4306.aspx, 2004.

[NV04] James Newkirk y Alexei Vorontsov. *Test-driven development in Microsoft .Net*. Microsoft Press, 2004.

[Op92] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. Tesis de Doctorado, University of Illinois at Urbana-Champaign, 1992.

[Orr03] Ken Orr. "CMM versus Agile Development: Religious wars and software development". Cutter Consortium, Executive Reports, 3(7), 2003.

[Pau01] Mark Paulk. "Extreme Programming from a CMM Perspective". *IEEE Software*, 2001.

[Pau02] Mark Paulk. "Agile Methodologies from a CMM Perspective". SEI-CMU, Stiembre de 2002.

[Platt02] Michael Platt. "Microsoft Architecture Overview: Executive summary", http://msdn.microsoft.com/architecture/default.aspx?pull=/library/en-us/dnea/html/eaarchover.asp, 2002.

[PMB04] PMBOK® Guide – 2000 Edition. Project Management Institute, http://www.pmi.org/prod/groups/public/documents/info/pp_pmbok2000welcome.asp, 2004.

[Pop01] Mary Poppendieck. "Lean Programming". http://www.agilealliance.org/articles/articles/LeanProgramming.htm, 2001.

[PP03] Mary Poppendieck y Tom Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, 2003.

[PT03] David Preedy y Paul Turner. "DSDM & MSF: A Joint Paper". http://www.dsdm.org, 2003.

[PW92] Dewayne E. Perry y Alexander L. Wolf. "Foundations for the study of software architecture". *ACM SIGSOFT Software Engineering Notes*, 17(4), pp. 40–52, Octubre de 1992.

[Rac95] L. B. S. Racoon. "The Chaos Model and the Chaos Life Cycle". *Software Engineering Notes*, 20(1), Enero de 1995.

[Rak01] Steven Rakitin. "Manifesto elicits cynicism". *Computer*, pp. 4-7, Diciembre de 2001.

[Rie00] Dirk Riehle. "A comparison of the value systems of Adaptive Software Development and Extreme Programming: How methodologies may learn from

each other". http://www.riehle.org/computer-science/research/2000/xp-2000.pdf, 2000.

[Roy70] Winston Royce. "Managing the development of large software systems: Concepts and techniques". *Proceedings of WESCON*, Agosto de 1970.

[RS01] Bernhard Rumpe y Astrid Schröder. "Quantitative survey on Extreme Programming Projects". Informe, Universidad de Tecnología de Munich, 2001.

[SB02] Ken Schwaber y Mike Beedle. *Agile software development with Scrum*. Prentice-Hall, 2002.

[Sch95] Ken Schwaber. "The Scrum development process". *OOPSLA '95 Workshop on Business Object Design and Implementation*, Austin, 1995.

[SEI03] Capability Maturity Model® for Software (SW-CMM®). http://www.sei.cmu.edu/cmm/cmm.html, 2003.

[She97] Sarah Sheard. "The frameworks quagmire, a brief look". Software Productivy Consortium, NFP, http://www.software.org/quagmire/frampapr/FRAMPAPR.HTML, 1997.

[Shi03] Shine Technologies. "Agile Methodologies Survey Result". http://www.shinetech.com/ShineTechAgileSurvey2003-01-17.pdf, Enero de 2003.

[SmiS/f] John Smith. "A comparison of RUP® and XP". *Rational Software White Paper,* sin fecha.

[SN04] Will Stott y James Newkirk. "Test-driven C#: Improve the design and flexibility of your project with Extreme Programming techniques". *MSDN Magazine*, http://msdn.microsoft.com/msdnmag/issues/04/04/ExtremeProgramming/default.aspx. Abril de 2004.

[SR03] Matt Stephens y Doug Rosenberg. *Extreme Programming Refactored: The case against XP*. Apress, 2003.

[Sta97] Jennifer Stapleton. *Dynamic Systems Development Method – The method in practice*. Addison Wesley, 1997.

[Sut01] Jeff Sutherland. "Agile *can* scale: Inventing and reinventing SCRUM in five companies". *Cutter IT Journal*, Diciembre de 2001.

[TBT00] Duane Truex, Richard Baskerville y Julie Travis. "Amethodical Systems Development: The deferred meaning of systems development methods". *Accounting, Management and Information Technology*, 10, pp. 53-79, 2000.

[TJ02] Richard Turner y Apurva Jain. "Agile meets CMMI: Culture clash or common cause". En Don Wells y Laurie A. Williams (Eds.), *Proceedings of Extreme Programming and Agile Methods - XP/Agile Universe 2002*, *Lecture Notes in Computer Science 2418*, Springer Verlag, 2002.

[TN86] Hirotaka Takeuchi e Ikujiro Nonaka. "The new product development game". *Harvard Business Review*, pp. 137-146, Enero-Febrero de 1986.

[Tra02] Aron Trauring. “Software methodologies: The battle of the Gurus”. *Info-Tech White Papers*, 2002.

[Tur03] Paul Turner. “DSDM, Prince2 and Microsoft Solutions Framework”. DSDM Consortium, http://www.dsdm.org/kss/details.asp?fileid=274, Noviembre de 2003.

[Wie98] Karl Wiegers. “Read my lips: No new models!”. *IEEE Software*. Setiembre-Octubre de 1998.

[Wie99b] Karl Wiegers. “Molding the CMM to your organization”. *Software Development*, Mayo de 1998.

[WS95] Jane Wood y Denise Silver. *Joint Application Development*. 2ª edición, Nueva York, John Wiley & Sons, 1995.

[WTR91] James Womack, Daniel Jones y Daniel Roos. *The machine that changed the world: The story of Lean Production*. HarperBusiness, 1991.