

# ¿Serán la Teoría del Caos y un Enfoque Despreocupado

## Las Claves para un Desarrollo de Proyectos Mejor y Más Rápido?

### El escenario

Seguramente usted se ha enfrentado a esta situación antes: las especificaciones del proyecto han cambiado con tanta frecuencia que el actualizar el cronograma se ha convertido en un ejercicio inútil. Hasta hace poco, existía una sola explicación aceptada por todos para esta situación una mala metodología es decir no se debería haber comenzado un proyecto sin un conjunto bien definido de especificaciones.

Pero, la vida real nos ha demostrado que el compilar un conjunto de este tipo de especificaciones es una aventura llena de peligros. Tendrá que hacer estimaciones sin nada concreto o invertir grandes cantidades de tiempo y dinero para obtener una guía definitiva de especificaciones. Y por supuesto, hablar de "especificaciones definitivas" parece una contradicción de términos en desarrollo de software.

Hablando ahora sobre la fase de diseño, no debe sorprendernos que los desarrolladores del equipo digan algo como "ahora que entiendo el problema pienso que puedo crear el código correcto". Pero de esto se dan cuenta cuando el módulo ya ha sido escrito. Otra contradicción de términos en nuestra industria podría ser "diseño de software correcto al primer intento".

Finalmente, hablemos sobre pruebas, cuando se establecen cronogramas esta es un área sobre lo que no hay nada escrito. Dependiendo de cuán bien se hayan definido y comprendido las especificaciones y el diseño y también de la calidad del equipo de desarrollo, las pruebas podrían representar del 20 al 80% del tiempo total. Gerentes de proyecto sin experiencia con frecuencia asignan un máximo de 15% a esta fase. Esto ocurre, porque la alta gerencia podría pensar que un mayor porcentaje significaría que el equipo de desarrollo ha estado creando software de mala calidad.

Le parece conocido este escenario? Si es que no, usted pertenece a una minoría con suerte. Tanto estudios formales como informales han demostrado que la mayoría de proyectos de software no cumplen los cronogramas ni los presupuestos, a veces con consecuencias desastrosas.

### La causa del problema

Claro que este escenario no es nada nuevo. Hace ya cerca de veinte años se crearon diferentes metodologías para disciplinar los proyectos de software utilizando prácticas estándares de ingeniería. A pesar de esto, la situación ha permanecido igual y los creadores de metodologías dicen que la culpa ha sido por la lenta adopción de sus propuestas. Después de todos estos años, uno se pregunta por qué el mercado ha sido tan reticente en adoptarlas, si en realidad estas metodologías pueden resolver problemas tan costosos.

De hecho, voceros respetados de la comunidad del desarrollo sugieren que las así llamadas metodologías monumentales pueden haber causado más daños que beneficios y que la situación parece que se ha tornado más grave desde el inicio del Internet y el increíble aumento de la velocidad en los cambios en los modelos de negocios. Estos voceros piensan que las metodologías monumentales surgieron de paradigmas mal concebidos, como por ejemplo, que la ingeniería de software se asemeja a la eléctrica, química y especialmente a la civil. Según un análisis posterior (que se ha demorado quince años en llevar a cabo), existen algunas diferencias fundamentales

## el escenario

...no debe sorprendernos que los desarrolladores del equipo digan algo como "ahora que entiendo el problema pienso que puedo crear el código correcto..."

## la causa del problema

...las así llamadas metodologías monumentales pueden haber causado más daños que beneficios ...

1. En la ingeniería civil, usted hace un análisis detenido y un diseño del producto a fabricar y luego ejecuta un proceso de construcción bastante caro, pero en términos generales rutinario y sin mayores cambios. Esto es posible solamente porque el producto fue comprendido muy bien, es relativamente simple y (especialmente) las necesidades y leyes físicas no cambiaban.

La mayoría de proyectos de software trata de capturar relaciones complejas y siempre cambiantes y flujos de trabajo, además de una teoría unificada de cómo funcionan las organizaciones humanas. No existe todavía una teoría que pueda convertir tales sistemas en algo tan predecible como, por ejemplo, la resistencia de los materiales.

2. Algunas investigaciones sugieren que la creación de software es un ejercicio continuo de diseño (es decir: hallar una solución, construirla, ver si funciona, si funciona, ir al próximo reto, sino comenzar de nuevo.) Esto hace que el centro de gravedad económico del proyecto se aleje de la construcción repetitiva y se acerque al lado creativo. En este sentido, los proyectos de software se asemejan más a los departamentos de investigación y desarrollo que a las fábricas.

3. El análisis y diseño de la ingeniería civil se llevan a cabo por gente muy inteligente y preparada, en cambio la fabricación en sí puede ser realizada por gente menos preparada. En ingeniería de software, profesionales igual de inteligentes y preparados se dedican a la construcción. Si uno contrata programadores no muy listos que escriben código sin pensar mucho al respecto para cumplir especificaciones, es poco probable que se logre un proyecto exitoso. Una consecuencia muy importante es la siguiente: en la ingeniería civil, los métodos de trabajo pueden ser impuestos por una autoridad, en cambio en ingeniería de software, el líder debe convencer a sus compañeros.

Tenemos que darnos cuenta de que sí existen proyectos de software con especificaciones fijas y que se pueden comprender de manera razonable. En estos casos la mentalidad de la ingeniería civil funciona muy bien. Lamentablemente, tengo el presentimiento de que la mayoría de proyectos de software no cumple estos parámetros y en estos casos, la escuela de administración basada en "análisis/diseño/definición de tareas y costos/creación de calendarios/construcción siguiendo planes" no es adecuada.

## El inicio de la solución

### el inicio de la solución

Martín Fowler afirma de manera convincente que probablemente la diferencia fundamental entre los proyectos de software y de otras ingenierías es que los primeros no son predecibles. Como hemos indicado, esto es el resultado de especificaciones cambiantes y de la falta de comprensión de cómo deber funcionar el sistema que se está tratando de automatizar. Entonces, pregunta Martín, cómo podemos controlar un mundo no predecible? La respuesta, o más bien dicho, el fundamento teórico de la respuesta viene de un campo no esperado: la teoría del caos, conocida más formalmente (sospecho que deseaban evitar los chistes) como teoría de Sistemas Complejos Adaptables (CAS, según sus siglas en el inglés). El enfoque de la teoría de CAS es explicar cómo sistemas complejos tales como el cuerpo humano, la economía nacional, o la evolución de las especies, pueden ser estables e incluso mejoradas, sin una entidad de control que indique los cursos de acción a tomarse (algunos dirían: aún a pesar de que existen entidades que tratan de normar cuáles son las acciones a tomar.)

## Todos los CAS tienen algunas propiedades principales

### todos los CAS tienen algunas propiedades principales

1. Está compuesto de agentes independientes (por ejemplo, empresas en una economía o células nerviosas en un cerebro). Estos agentes interactúan entre sí de manera no lineal, es decir, el entero es más que la suma de sus partes. El control tiende a ser muy disperso y el orden no se impone de arriba hacia abajo, pero de alguna manera surge de las interacciones de sus agentes, a pesar de que cada uno tiene sus propios objetivos.

2. Existen varios niveles de organización dentro de un CAS y cada nivel provee los bloques para construir el siguiente nivel. Por ejemplo, las personas forman departamentos, los departamentos empresas y las empresas la economía. Además, las agregaciones actúan como agentes también.

3. Los agentes anticipan el futuro, tomando acciones basadas en un conjunto de reglas internas. Estas pueden ser modelos internos simples o complejos que han sido autoimpuestos por el agente (por ejemplo, una regla relativa al estado de mercado que hace que se abran o cierren las contrataciones de personal en una empresa).

4. Cada CAS tiene muchos nichos y cada uno es capaz de ser explotado por unos pocos agentes particulares. Esto diversifica los agentes, ya que se adaptarán para sobrevivir y prosperar, lo cual a su vez hace que los nichos sean cada vez más diversos. Un ejemplo de algo que se mueve rápidamente es el mercado del Internet

Un concepto muy revelante de nuestro análisis es aquel del orden emergente, el orden que aparece por la cooperación / competencia (conocido como "cooperencia") de los agentes del CAS. El siguiente relato demuestra este concepto de manera clara (esta historia, al igual que la mayor a de esta sección ha sido extra do de un libro que consi- dero como lectura obligatoria: "El Desarrollo de Software Adaptable" de Jim Highsmith.)

A mediados de los años 1980, Craig Reynolds estaba interesado en simular el compor- tamiento de una bandada de pájaros. En vez de intentar descubrir y programar reglas para toda la bandada, Reynolds decidió programar el comportamiento de cada "boid" (que fue el nombre con el que denominó a sus pájaros simulados). Las reglas que si- guieron sus "boids" fueron:

1. Intentan mantener una separación mínima con otros "boids".
2. Intentan lograr que su velocidad se asemeje a la de los "boids" cercanos.
3. Intentan moverse hacia el centro de la masa de "boids" cercanos para alcanzar la coherencia con éstos.

Por sorprendente que parezca, las bandadas de "boids" simulaban bastante bien a una bandada real de pájaros. El orden surgía del comportamiento individual sin que haya ninguna regla sobre el grupo.

Es interesante tomar en cuenta que en la mayoría de CAS, la complejidad en sí nos impide comprender cómo funciona todo el sistema. Pero, al comprender cómo los agentes relevantes se comportan, podemos generalmente simular el todo, y así volver- lo predecible.

Entonces, para regresar a la pregunta que se planteó al inicio de esta sección, cómo podemos controlar un sistema no predecible (en nuestro caso un proyecto de softwa- re)? Una respuesta simplificada se aproxima a lo siguiente:

1. Identificar los agentes relevantes: clientes, programadores, usuarios, etc.
2. Identificar patrones de comportamiento de estos agentes en proyectos exitosos.
3. Sumergir a los agentes en este comportamiento y luego soltarlos. Se espera que se auto-organizarán, haciendo que el proyecto no solo sobreviva sino que prospere con energía al filo del caos.

Le parece esto algo intrigante pero poco profundo para intentarlo? Espere porque va- mos a analizar no sólo uno sino dos enfoques desarrollados de manera independiente (pero que en todo caso tienen algunas semejanzas sorprendentes) que se basan en estas ideas y que han sido usados ya en proyectos exitosos de desarrollo de software (aquellos en que todo el mundo termina feliz al final del día).

## Problemas de diseño extremos, soluciones de programación extremas

### problemas de diseño extremos, soluciones de programación extremas

El sentido común del desarrollo de software nos dice que mientras más tarde se intro- duzcan cambios en un sistema, éste se volverá más caro y complicado. Eventualmente ser más atractiva la idea de escribir un nuevo sistema en vez de añadir nueva funcio- nalidad al ya existente. Por lo tanto, deberá llevar a cabo un análisis completo para averiguar las necesidades posibles (presentes y futuras) y crear un diseño que sea lo suficientemente flexible como para poder incorporarlas. Debido a la velocidad actual de cambios tecnológicos y de mercado, tal tarea parece más apta para un adivino an- tes que para un ingeniero de software.

Kent Beck nos invita a un mundo distinto: un mundo en el cual añadir funcionalidad en la mitad o al final del proceso de desarrollo es solo ligeramente más caro que hacerlo al principio. Antes de iniciar un debate complicado sobre este proceso de soñar des- pierto, consideremos por un momento cuáles serían las consecuencias de tal utopía.

Inicialmente, pondría atención en solamente los requerimientos actuales y luego di- señaría una solución que satisfaga únicamente dichas necesidades. De esta manera, se detendría por fin el proceso de adivinanzas. La solución producida sería tan simple como sea posible y como usted sabe, la simplicidad en el diseño es la clave del éxito. Entonces, escribiría el código que cumpla con los requerimientos actuales y no incluiría ni una línea adicional. Los programas cortos son más fáciles de comprender, contie- nen menos errores y en general mejoran la productividad del programador. Finalmente,

tal tarea parece más  
apta para un adivino  
antes que para un in-  
geniero de software..

el cliente recibiría en el menor tiempo posible exactamente lo requerido, que es generalmente exactamente aquello por lo que había pagado. Suena grandioso, no?

Los beneficios son tan increíbles que Jim Highsmith dice que, aun cuando la propuesta de Ken Beck no fuera cierta, deberíamos desarrollar sistemas de tal manera que sea verdadera. Pero esto aparentemente es imposible de lograr, ya que hemos notado cómo un sistema inicialmente simple se vuelve cada vez más complicado mientras se añaden características, a la vez que el diseño se vuelve malo y aumentar la funcionalidad se complica paulatinamente, se vuelve riesgosa, y realmente poco agradable. Cómo podemos evitar este proceso? Martín Fowler define el "refactoring" como el proceso de cambiar un sistema de software de tal manera que no se altere el comportamiento externo del código pero se mejore su estructura interna. Mejor aún, Fowler no se queda en la propuesta sino que provee consejos detallados en un ejemplo en "Refactoring", que es otro libro de lectura obligada de desarrollo de software. Recuerde la idea consiste en cambiar código que ya funciona para convertirlo a más adaptable a cambios que se presenten en el futuro, y no para que corra más rápidamente ni para ahorrar en su consumo de memoria.

Pero tal vez ahora se sienta desconfiado, ya que todos conocemos lo que ocurre cuando se cambia código que funciona: el sistema comienza a fallar en sitios poco esperados y de manera poco esperada. Falla a menos que haya estado programando mediante el método de programación con pruebas iniciales. Esta técnica propone que cuando a un programador se le asigna un proyecto, deber desarrollar una serie de pruebas para comprobar si su solución resuelve el problema. Luego escribe un pequeño programa o rutina para correr las pruebas de manera automatizada. Solamente después de haber hecho todo esto, el programador comenzará el trabajo de escribir el código verdadero de la solución. Ciertamente esto requiere más disciplina y trabajo que solamente codificar, pero los beneficios son muchos: cuando se crean las pruebas, uno puede pensar también en la solución, verificar si se han satisfechos los requerimientos y finalmente, cuando se hace "refactoring" uno puede estar seguro de que no se ha "roto" el funcionamiento. El "refactoring" y la programación con pruebas iniciales son ejemplos claros de técnicas de desarrollo complementarias y son parte de las doce prácticas de la Programación Extrema, o XP según sus siglas, una metodología de desarrollo que promueve un enfoque de regresar a los conceptos básicos.

El XP confronta directamente las metodologías monumentales, también conocidas como metodologías Gran M, ya que propone un conjunto mínimo de prácticas dirigidas a impactar directamente en la productividad de los programadores y la calidad de su trabajo en vez de controlar todo el proyecto por medio de documentación e hitos. En el capítulo 10 de "Extreme Programming Explained" (Programación Extrema Explicada), Kent Beck resume las doce prácticas:

1. El Juego de la Planificación: determine rápidamente el alcance de la próxima versión al combinar las prioridades de negocio con los estimados técnicos. Cuando la realidad supere al plan, actualícelo.
2. Pequeñas versiones: ponga en producción a un sistema simple rápidamente, luego lance al mercado nuevas versiones durante un ciclo corto.
3. Metáfora. Guíe todo el desarrollo con una historia simple de cómo funciona todo el sistema.
4. Diseño simple: el sistema deber diseñarse de tal manera que sea lo más simple posible en cualquier momento dado. La complejidad adicional se retira tan pronto sea descubierta.
5. Pruebas: los programadores continuamente escribirán pruebas de unidad, que deben correr sin problemas para que el proceso de desarrollo pueda continuar. Los clientes escribirán pruebas demostrando que se ha completado la implementación de las características.
6. "Refactoring": los programadores cambiarán la estructura del sistema sin cambiar su comportamiento para eliminar la duplicación, mejorar la comunicación, simplificar, y añadir flexibilidad.
7. Programación por pares: todo el código de producción será escrito por dos programadores que compartirán una sola máquina.
8. Propiedad colectiva: cualquiera puede cambiar código de cualquier parte del sistema en cualquier momento.
9. Integración continua: se integrará y creará las nuevas versiones del sistema muchas veces por día, siempre que se haya completado una tarea.
10. Semanas de 40 horas: solamente se trabajará 40 horas por semana, como regla. Nunca se trabajará horas extras dos semanas seguidas.

## problemas de diseño extremos, soluciones de programación extremas

...el "refactoring" y la programación con pruebas iniciales son ejemplos claros de técnicas de desarrollo complementarias y son parte de las doce prácticas de la Programación Extrema...

11. Cliente en el sitio: incluya un verdadero usuario como parte del equipo, que esté disponible a tiempo completo para contestar las preguntas.
12. Estándares de codificación: todos los programadores deberán escribir código que cumplan las reglas y se enfatizar la comunicación durante su creación.

Algunas de las prácticas indicadas son de sentido común: cliente disponible en el lugar, estándares de codificación. En cambio, otras son técnicas redescubiertas: ejecución de pruebas y simplificación en el diseño, "refactoring" y el juego de la planificación. Otras prácticas pueden ser controversiales: programación por pares, y propiedad colectiva. Beck presenta razones para alentar la aplicación de estas prácticas e incluye detalles explícitos sobre cómo emplearlas. El resultado general de la metodología: haga lo mínimo que puede llegar a funcionar, obtenga toda la retroalimentación que pueda, cambie calendarios y haga "refactoring" libremente. No hay un plan a largo plazo o un horario fijo y se prefiere la comunicación personal a la escrita. Desde el punto de vista de las metodologías monumentales, este comportamiento es perezoso (sino se lo puede considerar "malo"). De todas maneras, lo importante es: ¿Se consiguen los resultados esperados de esta manera?

Hace cerca de diez años, Alistair Cockburn fue contratado por IBM para que cree una metodología para el desarrollo orientado a objetos. Parte de su enfoque fue entrevistar al mayor número posible de miembros de equipos de proyectos, anotando todo lo que decían contribuía a su éxito o fracaso. Según sus palabras:

"En el estudio de IBM, todo equipo exitoso "se disculpaba" por no seguir un proceso formal, por no usar una herramienta tipo CASE de alta tecnología, por simplemente mantenerse cerca y analizar cualquier cosa que surgía. Mientras tanto, una buena cantidad de equipos con problemas no lograban encontrar el motivo de sus fallas a pesar de seguir un proceso formal, tal vez se olvidaron de algún paso? Finalmente comencé a conocer equipos que se daban cuenta que su éxito se había dado justamente por no seguir procesos elegantes con entregas formales, pero que más bien discutían con libertad y entregaban software, luego de realizar varias pruebas."

Estos resultados han sido consistentes, desde 1991 a 1999, desde Hong Kong al continente Americano, en Noruega, y en Sudáfrica, tanto para sistemas en COBOL como Smalltalk, Java, Visual Basic, Sapiens y Synon. Para resumir los resultados podemos decir:

Siempre que pueda reemplazar la documentación escrita con comunicación oral, hágalo. Así reducir la dependencia en productos escritos y mejorará la probabilidad de llegar a entregar el sistema.

Mientras más frecuentemente pueda entregar pedazos del sistema que corran y hayan sido probados, depender menos de "promesas" y podrá entregar el sistema.

Las personas son entes que se comunican. Incluso los programadores más introvertidos, funcionan mejor en un ambiente en donde haya comunicación informal llevada a cabo cara a cara que en aquel donde exista solamente comunicaciones escritas. Desde un punto de vista de costos y horarios, el escribir algo toma más tiempo y comunica menos que una discusión con ejemplos en la pizarra blanca.

Aparentemente, comportamientos similares a aquellos propuestos por XP han resultado exitosos en muchos lugares y en muchas pocas. ¿Se acuerda de lo que discutimos sobre agentes en un ambiente no predecible? Mencionamos que los agentes definen sus reglas de comportamiento y los van refinando para poder llevar a cabo su trabajo adecuadamente en su nicho. Aquí podemos observar cómo cada equipo de desarrollo ha descubierto de manera independiente el mismo conjunto básico de reglas que les permite auto-organizarse y del cual surge un orden, en vez de que se imponga desde arriba autoritariamente. Si seguimos la teoría de CAS, deberíamos tratar de incluir estas reglas en nuestros equipos de desarrollo, para poder mejorar sus probabilidades de éxito.

# La Programación Extrema y los Centros de Desarrollo Virtual

## programación extrema y centros de desarrollo virtual

aplicar XP a los CDV(Centros de Desarrollo Virtual) que son aquellos equipos que están geográficamente separados y que pueden llegar a tener varias docenas de desarrolladores.

Para nosotros, es especialmente relevante analizar cómo se puede aplicar XP a los CDV(Centros de Desarrollo Virtual) que son aquellos equipos que están geográficamente separados y que pueden llegar a tener varias docenas de desarrolladores. Kent Beck señala que XP asume que los grupos son pequeños (de 20 personas máximo) que trabajan en el mismo lugar, por lo que resulta interesante preguntarnos qué ocurre con grandes proyectos y con grupos que trabajan en diferentes lugares.

Analicemos primero el problema del tamaño de grupo. A pesar de que XP recomienda usar todas sus prácticas, si es posible usar solamente un subconjunto de éstas. Además se puede modificarlas un poco para obtener todos los beneficios del XP puro. En la Conferencia "JavaOne 2001", Michael Lauer presentó una conferencia sobre el uso de XP para grandes proyectos de J2EE. Describió los detalles de un proyecto exitoso que excedía considerablemente el tamaño recomendado por Beck. En resumen, Lauer propuso lo siguiente:

1. Contar con un diseño serio desde el inicio en vez de concentrarse en la metáfora propuesta por XP.
2. Contar con un grupo principal de arquitectos senior para desarrollar una serie de patrones de arquitectura de componentes, por ejemplo patrones de persistencia de objetos o patrones en cuanto a la interfaz de usuario MVC. Se entregarán estos patrones a los desarrolladores.
3. Usar un subconjunto de prácticas de XP. Algunas de las excluidas fueron: programación por pares obligatoria, la semana de 40 horas (qué pena !) y, lo más sorprendente, el cliente disponible en el mismo local.

Lauer afirma que de esta manera logró :

1. Mantener contentos a los gerentes, ya que un arquitecto administraba de 20 a 40 desarrolladores.
2. Disminuir los costos de entrenamiento (aun cuando solo 1 o 2 de cada 10 solicitantes tenían el perfil correcto para el proyecto).
3. Acortar los ciclos de desarrollo.
4. Crear software más robusto.
5. Capacitación continua
6. Eliminación del mito del mes-hombre, logrando un gran grado de paralelismo en el desarrollo.

Y por supuesto, Lauer y sus clientes estaban contentos y sentían que habían logrado algo importante. Entonces, parece que el factor del tamaño de equipo podría eliminarse si se varía en cierto grado a XP.

Cuando se analiza la dispersión geográfica del equipo, tenemos que recordar que lo que requiere XP es que exista comunicación fluida de persona a persona. Debido a las mejoras continuas de las telecomunicaciones y las herramientas de Internet que van desde las más simples ("chat") a las más sofisticadas (reuniones virtuales), estar presentes electrónicamente en el mismo ambiente es casi igual a estar físicamente en el mismo cuarto. La experiencia se está mejorando cada vez más. Sin embargo, el hecho de estar en diferentes husos horarios sí podrá afectar la comunicación, sin importar que tan buena sea la infraestructura de comunicaciones.

Kent Beck trata de delimitar claramente dónde funciona y en qué condiciones no funciona XP. Por ejemplo, funciona muy bien cuando se trata de equipos localizados en un sitio y el proyecto a desarrollarse no es un sistema de misión crítica. Y que pasa si su proyecto cumple con todas las especificaciones? Sería interesante poder iniciar el trabajo con el modelo XP (o algo equivalente) y luego adaptarlo durante la ejecución para que satisfaga las demandas del proyecto. Pero parece difícil cambiar la metodología de desarrollo en medio del proyecto. Usando terminología de la metodología CAS, lo que necesitamos es no solo descubrir y emular el comportamiento de los agentes exitosos pero también conocer los procesos que usan los agentes para intentar encontrar y adoptar nuevas reglas. Si logramos esto y podemos emular dichos procesos, podremos crear equipos que no solo creen buen software sino que también adapten los procedimientos a su ambiente específico. De esta manera su rendimiento mejorará cada vez más y ésta es justo la idea detrás de la metodología de Desarrollo de Software Adaptable (o ASD según sus siglas en inglés) que es nuestro siguiente tema

# Un enfoque colaborativo al manejo de sistemas complejos

## un enfoque colaborativo al manejo de sistemas complejos

...un nuevo vocabulario para el manejo de proyectos, por ejemplo, el ciclo de desarrollo debe contar con tres fases: especular, colaborar y aprender. Suena extraño ? ...

En su libro, Jim Highsmith, trata de comprender cómo las organizaciones y los equipos en general trabajan juntos y logran completar sus proyectos. Al concluir este análisis, trata recién de estudiar el desarrollo de proyectos de software como un caso especial. Highsmith se dedicó a la enseñanza y uso de metodologías monumentales por muchos años. Luego inició el uso de un enfoque más liviano, un poco menos extremo que el de Kent Beck y sus colaboradores. De todas maneras, nos alienta a cambiar de paradigmas, para usar una de las palabras favoritas de Microsoft.

Highsmith introdujo un nuevo vocabulario para el manejo de proyectos, por ejemplo, el ciclo de desarrollo debe contar con tres fases: especular, colaborar y aprender. Suena extraño? Prefirió usar especular en vez de planificar ya que piensa que la palabra "planificación" se usa cuando se sabe exactamente hacia dónde nos encaminamos, pero en realidad en CAS tenemos un sueño, una visión apasionada pero poco clara de lo que queremos. Según él, descubrimos lo que necesitamos mientras se desarrolla el trabajo. Además, si durante la creación de un plan, uno se desvía, se piensa que es un defecto. Los gerentes opinan, en ese momento, que se debe regresar al plan original, mientras que en el CAS se considera que los desvíos son los esfuerzos del equipo por encontrar la verdadera solución, por lo que deberían seguirse cuidadosamente si pensamos que nos dirigirán a ésta. Use la palabra "colaborar" en vez de "construir" ya que opina que la actividad más importante de un equipo es trabajar juntos, y no contar con una lista de tareas a ejecutarse. Sostiene que el poder del equipo no consiste en las fortalezas individuales de sus miembros, sino más bien en la cooperación abierta y generosa para lograr el objetivo más importante: el cumplimiento de la misión del proyecto. Finalmente, escoge la palabra "aprender" en vez de "retroalimentar" debido a que desde un punto de vista de ingeniería a la idea de retroalimentar es obtener información sobre el rendimiento específico. Pero en un sistema complejo, no se busca lo óptimo sino la adaptación a condiciones siempre cambiantes: la idea de algo ptimo hace sentido solamente cuando existen condiciones estables, en donde también hay límites preestablecidos a alcanzarse. En los sistemas complejos éstos también existen, pero siempre cambian (a veces se transforman en algo peor, o algo mejor), entonces el objetivo es aprender cuáles son los límites actuales, cuáles partes de su comportamiento le ayudarán en dichas circunstancias y qué partes se deberán cambiar. Entonces, no hace falta solamente la retroalimentación, sino el aprendizaje.

Tal vez la transición desde planificar-construir-recibir retroalimentación hacia especular-colaborar-aprender suene como mero juego de palabras, especialmente cuando averigüe que el ASD propone ciclos cortos de desarrollo orientados a la entrega de componentes, como la XP, sin ningún vocabulario complicado detrás de esto. Pero me parece que la situación es semejante a cuando fui de programación de procedimientos a aquella de orientación a objetos. Inicialmente, pensaba que la idea de que "los objetos se mandan mensajes que reciben respuestas" era algo rara, sobre todo porque me parecía que simplemente estaban llamando a funciones como siempre. Pero las palabras tienen significados poderosos y el visualizar el sistema como una telaraña de actores que llevan a cabo su trabajo solicitándose entre sí ayuda específica en vez de que sea un conjunto, organizado jerárquicamente, de menús, ventanas, funciones y bases de datos, cambia en verdad la manera en que se crean los sistemas. No se trata del idioma o de las herramientas usadas, sino de la manera en que piensa sobre los problemas a resolver y sus soluciones. Tengo que reconocer que la terminología de ASD y su manera de ver el mundo ayudan a organizar los proyectos de software de un modo más natural.

Una cosa que XP no tiene es un nivel administrativo. Kent Beck señala que XP fue creado para equipos localizados en el mismo lugar, compuestos de 10 a 12 personas, en donde la comunicación constante y los ciclos cortos de construcción y retroalimentación casi reemplazan por completo la necesidad de contar con administradores especializados. Pero en los Centros de Desarrollo Virtual existen equipos distribuidos y probablemente más grandes que los de XP, entonces se requiere de algún tipo de coordinación. Como comente anteriormente, Michael Lauer resolvió el problema al incorporar algunas prácticas de las metodologías monumentales: contar con un documento de especificaciones detallado, presionar al equipo para que trabaje horas extras, y todo esto funcionó en general. El ASD también reconoce la necesidad de que existan administradores pero, basándose de nuevo en el comportamiento de organizaciones exitosas que viven en ambientes que cambian rápidamente, Jim Highsmith propone un modelo diferente para el manejo de proyectos de software. (Dentro de los ambientes cambiantes consideramos la biotecnología, la consultoría, y el software entre otros).

Según su descripción:

Sentido común, lo cual significa la manera de percibir el mundo que nos rodea, es algo que es indispensable en la administración. Si notamos que el mundo es estable y predecible, nuestro enfoque a la administración será muy diferente que cuando es turbulento y no predecible. Tener la idea de que el mundo es relativamente estable, un punto de vista como el de Newton, hizo que se creen prácticas de administración conocidas como "Control de Comando". En cambio, el percibir al mundo como algo cambiante y no predecible ha generado un nuevo conjunto de prácticas administrativas, conocidas a veces como participativas, modernas, o centradas en las personas. En un mundo tumultuoso, en el cual la idea de "sentido común" se mejora con la comprensión de sistemas complejos adaptables, creo que términos más aplicables a estas prácticas administrativas modificadas serían Liderazgo-Colaboración, en donde "liderazgo" reemplaza a "comando" y "colaboración" a "control".

Los líderes de un proyecto de CAS deben tener ideas, un equipo que trabaje en conjunto, y el deseo de arriesgarse. La colaboración se encargará de cómo se organizan los componentes de software y cómo los miembros del equipo se coordinan. Un principio de CAS es que no se monitorea a las personas basándose en las tareas cumplidas sino según un estado cada vez mejorado del proyecto. Esto significa se administra el estado del trabajo y no su flujo. Y esto nos lleva a otro aspecto de la administración: responsabilidad. En un proyecto CAS cada miembro se hace responsable del estado actual y del éxito (o fracaso) final del proyecto, entonces se detiene el juego de culpar a otro, así de plano. Se puede preguntar a cada uno la situación actual, ya que la comunicación fluye libremente, y nadie puede decir que no sabía. Además, se llevan horarios con información de cada ciclo de desarrollo para forzar al equipo a tomar decisiones. Esto es importante porque caso contrario los equipos tienen la tendencia a seguir todas las opciones nuevas, sin decidirse por un camino específico.

Los calendarios son ejemplos de una técnica ASD para evitar que un equipo, que se auto-organiza, llegue al caos. A pesar de que pueda pensar que el ASD es muy teórico y poco práctico, este no es el caso. De hecho, Jim Highsmith demuestra con ejemplos específicos y prácticos cómo ha funcionado en proyectos de software complejos. Pone énfasis en la palabra "ejemplos" ya que piensa que los proyectos complejos deben adaptarse, por lo que presenta patrones cuya utilidad se ha demostrado en vez de reglas fijas. Usted deberá decidir si su proyecto se asemeja a los incluidos en los ejemplos. Externamente, un proyecto ASD se parece mucho a uno XP, pero las bases teóricas más fuertes y la administración para evitar el caso hacen que sean intrigantes. En resumen, es divertido leer el libro: "Desarrollo de Software Adaptable" ya que está lleno de ejemplos y analogías. Por este motivo, le aliento a hacerlo y conocer sus detalles que no estén dentro del alcance de este informe.

## Conclusiones

### conclusiones

...equipos que se auto-organizan y se adaptan durante ciclos cortos de desarrollo en espiral y mediante el aprendizaje para alcanzar objetivos en movimientos...

La velocidad actual de cambios del Internet y adelantos del comercio electrónico no nos permiten el uso de metodologías inmensas que asumen que existe un ambiente bastante estable, para el manejo de proyectos de desarrollo de software. Debido a que la programación libre y sin dirección no es una alternativa, especialmente cuando se trata de proyectos grandes y distribuidos, se ha creado toda una familia de metodologías conocidas como livianas o ágiles. Una de éstas que es especialmente atractiva, por basarse fuertemente en los fundamentos de la teoría de Sistemas Adaptables Complejos CAS (también conocidos como teoría del caos), es el Desarrollo de Software Adaptable (ASD). Este propone equipos que se auto-organizan y se adaptan durante ciclos cortos de desarrollo en espiral y mediante el aprendizaje para alcanzar objetivos en movimiento. La metodología ágil más popular es la Programación Extrema (XP) que da importancia a los equipos pequeños y presenta prácticas específicas que mejoran la productividad del programador y la calidad y resiliencia del código.

Ambas metodologías se asemejan mucho a pesar del hecho de que evolucionaron de manera independiente. Ya que ASD propone más bien un conjunto de patrones y no uno de reglas fijas y, además, alienta procesos de adaptación permanentes, pienso que el uso de prácticas de XP con una administración ASD y con una adaptación rápida por medio del aprendizaje, constituyen una combinación muy efectiva para el manejo del desarrollo de software con equipos virtuales.

# R e c u r s o s

A pesar de que he escrito usando la primera persona, prácticamente todo lo que he dicho ha sido extraído de las fuentes indicadas más adelante. Solo tengo la esperanza de no haber entendido mal o recalcado las partes incorrectas de lo escrito por los autores originales. Fue mi idea el combinar ASP con XP a pesar de que es muy probable que alguien ya lo haya pensado antes. Trataé de mantenerlos informados acerca de lo que pase en este campo.

Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 2000

Alistair Cockburn, *Growth of Human Factors in Application Development*, <http://members.aol.com/acockburn/papers/adchange.htm>, 1995

Martin Fowler, *The New Methodology*, <http://www.martinfowler.com/articles/newMethodology.html>, 2001

Martin Fowler et al., *Refactoring*, Addison-Wesley, 2000

James A. Highsmith III, *Adaptive Software Development*, Dorset House, 2000

James A. Highsmith III, *Extreme Programming*, <http://www.cutter.com/ead/ead0002.html>, 2000

Michael Lauer, *Scaling XP to Large Projects for J2EE*, <http://java.sun.com/javaone/javaone2001/pdfs/2218.pdf>