**https://martinfowler.com/articles/newMethodology.html**

**La nueva metodología**

En los últimos años ha habido un florecimiento de un nuevo estilo de metodología de software, conocidos como métodos ágiles. Alternativamente, se caracteriza como un antídoto para la burocracia o una licencia para hackear que ha despertado el interés en todo el panorama del software. En este ensayo, exploro las razones de los métodos ágiles, centrándome no tanto en su peso sino en su naturaleza adaptativa y su orientación hacia las personas.

13 de diciembre de 2005 Foto de Martin Fowler Martin Fowler Traducciones: ruso · chino · indonesio · italiano · francés · bielorruso · danés · rumano · portugués Encuentre artículos similares a esto mirando estas etiquetas: popular · ágil · teoría del proceso

**Contenido**

- De nada, a monumental, a ágil

-Predictivo versus Adaptativo Separación de diseño y construcción

- La impredecibilidad de los requisitos ¿Es imposible predecir?

-Controlar un proceso impredecible - Iteraciones

-El cliente adaptable

-Poniendo a las personas primero

-Unidades de Programación Plug-Compatible

- Los programadores son profesionales responsables

-Gestionar un proceso orientado a las personas

- La dificultad de medición

-El papel del liderazgo empresarial

- El proceso de auto-adaptación

- Sabores del desarrollo ágil

-Manifiesto Ágil

- XP (programación extrema)

-Melé Cristal Prueba conducida por contexto

- Lean Development (Proceso racional unificado )

-¿Deberías ir ágil?

Probablemente, el cambio más notable en el proceso de software en los últimos años haya sido la aparición de la palabra "ágil". Hablamos de métodos de software ágiles, de cómo introducir la agilidad en un equipo de desarrollo, o de cómo resistir la inminente tormenta de agilistas decididos a cambiar prácticas bien establecidas.

Este nuevo movimiento surgió de los esfuerzos de varias personas que se ocuparon de los procesos de software en la década de 1990, los encontró deficientes y buscaron un nuevo enfoque para el proceso de software. La mayoría de las ideas no eran nuevas, de hecho mucha gente creía que mucho software exitoso se había construido de esa manera durante mucho tiempo. Hubo, sin embargo, una opinión de que estas ideas habían sido sofocadas y no habían sido tratadas con suficiente seriedad, particularmente por personas interesadas en el proceso del software.

Este ensayo fue originalmente parte de este movimiento. Originalmente lo publiqué en julio de 2000. Lo escribí, como la mayoría de mis ensayos, como parte de tratar de comprender el tema. En ese momento, utilicé Extreme Programming durante varios años después de tener la suerte de trabajar con Kent Beck, Ron Jeffries, Don Wells y, sobre todo, el resto del equipo Chrysler C3 en 1996. Desde entonces, tuve conversaciones y leí libros de otras personas que tenían ideas similares sobre el proceso de software, pero que no necesariamente habían querido tomar el mismo camino que Extreme Programming. Entonces, en el ensayo quise explorar cuáles eran las similitudes y diferencias entre estas metodologías

Si está interesado en curiosidades históricas, puede leer la versión original de este artículo. Aparte de los cambios de formato, el texto no se modifica.

Comentario:

//Mi conclusión entonces, que todavía creo ahora, es que había algunos principios fundamentales que unían estas metodologías, y estos principios contrastaban notablemente con los supuestos de las metodologías establecidas.//

Este ensayo ha seguido siendo uno de los ensayos más populares en mi sitio web, lo que significa que me siento un poco convencido de mantenerlo actualizado. En su forma original, el ensayo exploró estas diferencias en los principios y proporcionó una encuesta de métodos ágiles tal como yo los entendí. Demasiado ha sucedido con los métodos ágiles ya que me mantengo al día con la parte de la encuesta, aunque proporciono algunos enlaces para continuar con sus exploraciones. Las diferencias en los principios aún permanecen, y esta discusión la he mantenido.

**De nada, a monumental, a ágil**

La mayoría del desarrollo de software es una actividad caótica, a menudo caracterizada por la frase "código y corrección". El software está escrito sin mucho de un plan subyacente, y el diseño del sistema se improvisa a partir de muchas decisiones a corto plazo. En realidad, esto funciona bastante bien ya que el sistema es pequeño, pero a medida que el sistema crece, cada vez es más difícil agregar nuevas características al sistema. Además, los errores se vuelven cada vez más frecuentes y cada vez más difíciles de solucionar. Un signo típico de dicho sistema es una fase de prueba larga después de que el sistema se haya "completado". Una fase de prueba tan larga hace estragos en los horarios, ya que es imposible programar las pruebas y la depuración.

El movimiento original para tratar de cambiar esto introdujo la noción de metodología. Estas metodologías imponen un proceso disciplinado sobre el desarrollo de software con el objetivo de hacer que el desarrollo de software sea más predecible y más eficiente. Lo hacen mediante el desarrollo de un proceso detallado con un fuerte énfasis en la planificación inspirada en otras disciplinas de ingeniería, por lo que me gusta referirme a ellas como metodologías de ingeniería (otro término ampliamente utilizado para ellas son las metodologías basadas en planes).

Las metodologías de ingeniería han existido por mucho tiempo. No se han notado por ser terriblemente exitoso. Son menos conocidos por ser populares. La crítica más frecuente a estas metodologías es que son burocráticas. Hay tantas cosas que hacer para seguir la metodología que todo el ritmo de desarrollo se ralentiza.

Metodologías ágiles desarrolladas como una reacción a estas metodologías. Para muchas personas, el atractivo de estas metodologías ágiles es su reacción ante la burocracia de las metodologías de ingeniería. Estos nuevos métodos intentan un compromiso útil entre ningún proceso y demasiado proceso, proporcionando solo el proceso suficiente para obtener un pago razonable. El resultado de todo esto es que los métodos ágiles tienen algunos cambios significativos en el énfasis de los métodos de ingeniería. La diferencia más inmediata es que están menos orientadas a documentos, por lo general enfatizando una menor cantidad de documentación para una tarea determinada. En muchos sentidos, están más bien orientados al código: siguen una ruta que dice que la parte clave de la documentación es el código fuente.

Sin embargo, no creo que este sea el punto clave sobre los métodos ágiles. La falta de documentación es un síntoma de dos diferencias mucho más profundas:

- Los métodos ágiles son adaptativos en lugar de predictivos. Los métodos de ingeniería tienden a tratar de planificar una gran parte del proceso de software en gran detalle durante un largo período de tiempo, esto funciona bien hasta que las cosas cambien. Entonces su naturaleza es resistir el cambio. Los métodos ágiles, sin embargo, dan la bienvenida al cambio. Intentan ser procesos que se adaptan y prosperan en el cambio, incluso hasta el punto de cambiarse a sí mismos.

- Los métodos ágiles están más orientados a las personas que a los procesos. El objetivo de los métodos de ingeniería es definir un proceso que funcione bien para quien lo esté usando. Los métodos ágiles afirman que ningún proceso volverá a formar la habilidad del equipo de desarrollo, por lo que el papel de un proceso es apoyar al equipo de desarrollo en su trabajo.

En las siguientes secciones exploraré estas diferencias con más detalle, para que pueda entender cómo es un proceso adaptativo y centrado en las personas, sus ventajas y desventajas, y si es algo que debe usar: como desarrollador o cliente de software.

**Predictivo versus Adaptativo**

**Separación de diseño y construcción**

La inspiración habitual para las metodologías son las disciplinas de ingeniería, como la ingeniería civil o mecánica. Tales disciplinas ponen mucho énfasis en la planificación antes de construir. Dichos ingenieros trabajarán en una serie de dibujos que indican con precisión qué se debe construir y cómo deben reunirse estas cosas. Muchas decisiones de diseño, como la forma de manejar la carga en un puente, se realizan a medida que se producen los dibujos. Los dibujos se entregan a un grupo diferente, a menudo una compañía diferente, para ser construido. Se supone que el proceso de construcción seguirá los dibujos. En la práctica, los constructores se encontrarán con algunos problemas, pero estos generalmente son pequeños.

Dado que los dibujos especifican las piezas y cómo deben juntarse, sirven de base para un plan de construcción detallado. Tal plan puede descubrir las tareas que deben hacerse y qué dependencias existen entre estas tareas. Esto permite un cronograma y un presupuesto razonablemente predecibles para la construcción. También dice en detalle cómo las personas que hacen el trabajo de construcción deberían hacer su trabajo. Esto permite que la construcción sea menos hábil intelectualmente, aunque a menudo son muy hábiles a mano.

Entonces, lo que vemos aquí son dos actividades fundamentalmente diferentes. Diseño que es difícil de predecir y requiere personas costosas y creativas, y una construcción que es más fácil de predecir. Una vez que tenemos el diseño, podemos planificar la construcción. Una vez que tengamos el plan para la construcción, podremos tratar la construcción de una manera mucho más predecible. En la ingeniería civil, la construcción es mucho más costosa y temporal que el diseño y la planificación.

Entonces, el enfoque para las metodologías de ingeniería de software es el siguiente: queremos un cronograma predecible que pueda usar personas con habilidades más bajas. Para hacer esto debemos separar el diseño de la construcción. Por lo tanto, tenemos que descubrir cómo hacer el diseño del software para que la construcción pueda ser sencilla una vez que se haya completado la planificación.

Entonces, ¿qué forma toma este plan? Para muchos, este es el rol de las notaciones de diseño como el UML. Si podemos tomar todas las decisiones importantes usando UML, podemos construir un plan de construcción y luego entregar estos diseños a los programadores como una actividad de construcción.

Pero aquí yace la pregunta crucial. ¿Se puede obtener un diseño que sea capaz de convertir la codificación en una actividad de construcción predecible? Y si es así, ¿el costo de hacerlo es lo suficientemente pequeño para que este enfoque valga la pena?

Todo esto trae algunas preguntas a la mente. El primero es la cuestión de cuán difícil es llevar un diseño similar a UML a un estado que pueda ser entregado a los programadores. El problema con un diseño similar al UML es que puede verse muy bien en papel, pero tiene serias fallas cuando realmente tiene que programarlo. Los modelos que usan los ingenieros civiles se basan en muchos años de práctica que están consagrados en los códigos de ingeniería. Además, los problemas clave, como la forma en que las fuerzas juegan en el diseño, son susceptibles de análisis matemático. La única verificación que podemos hacer de los diagramas similares a UML es la revisión por pares. Si bien esto es útil, conduce a errores en el diseño que a menudo solo se descubren durante la codificación y la prueba. Incluso diseñadores expertos, como yo mismo me considero, a menudo se sorprenden cuando convertimos ese diseño en software.

Otro problema es el costo comparativo. Cuando construyes un puente, el costo del esfuerzo de diseño es de aproximadamente el 10% del trabajo, y el resto es de construcción. En el software, la cantidad de tiempo dedicado a la codificación es mucho, mucho menos. McConnell sugiere que para un proyecto grande, solo el 15% del proyecto es prueba de código y unidad, una reversión casi perfecta de las proporciones de construcción del puente. Incluso si acumula en todas las pruebas como parte de la construcción, el diseño sigue siendo el 50% del trabajo. Esto plantea una pregunta importante sobre la naturaleza del diseño en el software en comparación con su papel en otras ramas de la ingeniería.

Este tipo de preguntas llevaron a Jack Reeves a sugerir que, de hecho, el código fuente es un documento de diseño y que la fase de construcción es en realidad el uso del compilador y el enlazador. De hecho, cualquier cosa que pueda tratar como construcción puede y debe automatizarse.

Este pensamiento conduce a algunas conclusiones importantes:

- En software: la construcción es tan barata que es gratis

-En software todo el esfuerzo es diseño, y por lo tanto requiere personas creativas y talentosas

-Los procesos creativos no se planifican fácilmente, por lo que la predictibilidad puede ser un objetivo imposible.

- Deberíamos ser muy cuidadosos con la metáfora de la ingeniería tradicional para construir software. Es un tipo diferente de actividad y requiere un proceso diferente

**La impredecibilidad de los requisitos**

Hay un estribillo que he escuchado sobre cada proyecto problemático con el que me he encontrado. Los desarrolladores vienen y me dicen "el problema con este proyecto es que los requisitos siempre cambian". Lo que me sorprende de esta situación es que a cualquiera le sorprende. En la construcción de los requisitos del software de negocios, los cambios son la norma, la pregunta es qué hacemos al respecto.

Una ruta es tratar los requisitos cambiantes como resultado de una ingeniería de requisitos deficientes. La idea detrás de la ingeniería de requisitos es obtener una imagen completa de los requisitos antes de comenzar a construir el software, obtener la aprobación del cliente para estos requisitos y luego establecer procedimientos que limiten los cambios de requisitos después del cierre de sesión.

Un problema con esto es que solo tratar de comprender las opciones para los requisitos es difícil. Es aún más difícil porque la organización de desarrollo generalmente no proporciona información de costos sobre los requisitos. Al final, te encuentras en una situación en la que quizás desees tener un techo solar en tu auto, pero el vendedor no puede decirte si agrega $ 10 al costo del auto, o $ 10,000. Sin mucha idea del costo, ¿cómo puedes saber si quieres pagar por ese techo solar?

La estimación es difícil por muchas razones. Parte de esto es que el desarrollo de software es una actividad de diseño y, por lo tanto, difícil de planificar y de coste. Parte de esto es que los materiales básicos siguen cambiando rápidamente. Parte de esto es que tanto depende de qué personas individuales están involucradas, y los individuos son difíciles de predecir y cuantificar

La naturaleza intangible del software también interviene. Es muy difícil ver qué valor tiene una característica del software hasta que la utilices de verdad. Solo cuando usa una versión temprana de algún software realmente comienza a entender qué características son valiosas y qué partes no lo son.

Esto lleva al punto irónico de que las personas esperan que los requisitos se puedan cambiar. Después de todo el software se supone que es suave. Entonces, no solo son requisitos modificables, sino que deben ser cambiables. Se requiere mucha energía para lograr que los clientes del software arreglen los requisitos. Es incluso peor si alguna vez han incursionado en el desarrollo de software, porque entonces "saben" que el software es fácil de cambiar.

Pero incluso si pudieras resolver todo eso y realmente pudieras obtener un conjunto de requisitos preciso y estable, probablemente aún estés condenado. En la economía actual, las fuerzas empresariales fundamentales están cambiando el valor de las características del software con demasiada rapidez. Lo que podría ser un buen conjunto de requisitos ahora, no es un buen conjunto en seis meses. Incluso si los clientes pueden arreglar sus requisitos, el mundo de los negocios no va a detenerse por ellos. Y muchos cambios en el mundo de los negocios son completamente impredecibles: cualquiera que diga lo contrario está mintiendo o ya ha realizado un billón en el comercio bursátil.

Todo lo demás en el desarrollo de software depende de los requisitos. Si no puede obtener requisitos estables, no puede obtener un plan predecible.

¿Es imposible predecir?

En general, no. Hay algunos desarrollos de software donde la previsibilidad es posible. Organizaciones como el grupo de software de transbordadores espaciales de la NASA son un excelente ejemplo de donde el desarrollo de software puede ser predecible. Requiere mucha ceremonia, mucho tiempo, un gran equipo y requisitos estables. Hay proyectos por ahí que son transbordadores espaciales. Sin embargo, no creo que gran parte del software empresarial encaje en esa categoría. Para esto necesitas un tipo diferente de proceso.

Uno de los grandes peligros es pretender que puede seguir un proceso predecible cuando no puede. Las personas que trabajan en metodología no son muy buenas para identificar las condiciones de frontera: los lugares donde la metodología pasa de ser apropiada a inapropiada. La mayoría de los metodólogos quieren que sus metodologías sean utilizables por todos, por lo que no entienden ni publicitan sus condiciones de frontera. Esto lleva a que las personas utilicen una metodología en circunstancias incorrectas, como el uso de una metodología predecible en una situación impredecible.

Hay una gran tentación de hacer eso. La previsibilidad es una propiedad muy deseable. Sin embargo, si cree que puede ser predecible cuando no puede, esto lleva a situaciones en las que las personas elaboran un plan desde el principio, y luego no manejan adecuadamente la situación donde el plan se derrumba. Ves el plan y la realidad separándose lentamente. Durante mucho tiempo puede pretender que el plan sigue siendo válido. Pero en algún momento la deriva se vuelve demasiado y el plan se desmorona. Por lo general, la caída es dolorosa.

Entonces, si se encuentra en una situación que no es predecible, no puede usar una metodología predictiva. Eso es un duro golpe. Significa que muchos de los modelos para controlar proyectos, muchos de los modelos para toda la relación con el cliente, ya no son verdad. Los beneficios de la previsibilidad son tan grandes que es difícil dejarlos ir. Al igual que muchos problemas, la parte más difícil es simplemente darse cuenta de que el problema existe.

Sin embargo, dejar de lado la predictibilidad no significa que deba volver al caos incontrolable. En su lugar, necesita un proceso que pueda darle control sobre una imprevisibilidad. De eso se trata la adaptabilidad.

**Controlar un proceso impredecible – Iteraciones**

Entonces, ¿cómo nos controlamos en un mundo impredecible? La parte más importante y aún difícil es saber con precisión dónde estamos. Necesitamos un mecanismo de retroalimentación honesto que pueda decirnos con precisión cuál es la situación a intervalos frecuentes.

La clave de esta retroalimentación es el desarrollo iterativo. Esto no es una idea nueva. El desarrollo iterativo ha existido por un tiempo bajo muchos nombres: incremental, evolutivo, en etapas, espiral... muchos nombres. La clave del desarrollo iterativo es producir con frecuencia versiones de trabajo del sistema final que tengan un subconjunto de las características requeridas. Estos sistemas de trabajo tienen poca funcionalidad, pero deberían ser fieles a las demandas del sistema final. Deben estar completamente integrados y tan cuidadosamente probados como una entrega final.

El objetivo de esto es que no hay nada como un sistema probado e integrado para llevar una dosis contundente de realidad a cualquier proyecto. Los documentos pueden ocultar todo tipo de defectos. El código no probado puede ocultar muchos defectos. Pero cuando las personas realmente se sientan frente a un sistema y trabajan con él, entonces los defectos se hacen evidentes: tanto en términos de errores como en términos de requisitos incomprendidos.

El desarrollo iterativo también tiene sentido en procesos predecibles. Pero es esencial en los procesos adaptativos porque un proceso adaptativo debe ser capaz de lidiar con los cambios en las características requeridas. Esto conduce a un estilo de planificación donde los planes a largo plazo son muy fluidos, y los únicos planes estables son los planes a corto plazo que se crean para una única iteración. El desarrollo iterativo le proporciona una base sólida en cada iteración en la que puede basar sus planes posteriores.

Una pregunta clave para esto es cuánto tiempo debe ser una iteración. Diferentes personas dan diferentes respuestas. XP sugiere iteraciones de una o dos semanas. SCRUM sugiere una duración de un mes. Crystal puede estirar aún más. La tendencia, sin embargo, es hacer que cada iteración sea lo más corta posible. Esto proporciona comentarios más frecuentes, para que sepa dónde está más seguido.

**El cliente adaptable**

Este tipo de proceso adaptativo requiere un tipo diferente de relación con un cliente que los que a menudo se consideran, particularmente cuando el desarrollo lo realiza una empresa separada. Cuando contrata a una empresa por separado para desarrollar software, la mayoría de los clientes preferiría un contrato de precio fijo. Dígales a los desarrolladores lo que quieren, pida ofertas, acepte una oferta, y luego la responsabilidad recae sobre la organización de desarrollo para construir el software.

Un contrato de precio fijo requiere requisitos estables y, por lo tanto, un proceso predictivo. Los procesos adaptables y los requisitos inestables implican que no se puede trabajar con la noción usual de precio fijo. Intentar ajustar un modelo de precio fijo a un proceso adaptativo termina en una explosión muy dolorosa. La parte desagradable de esta explosión es que el cliente se lastima tanto como la compañía de desarrollo de software. Después de todo, el cliente no querría ningún software a menos que su negocio lo necesitara. Si no lo consiguen, su negocio sufrirá. Entonces, incluso si le pagan a la compañía de desarrollo nada, aún así pierden. De hecho, pierden más de lo que pagarían por el software (¿por qué pagarían por el software si el valor comercial de ese software fuera menor?)

Por lo tanto, existen peligros para ambas partes al firmar el contrato tradicional de precio fijo en condiciones en las que no se puede usar un proceso predictivo. Esto significa que el cliente tiene que trabajar de manera diferente.

Esto no significa que no pueda arreglar un presupuesto para el software por adelantado. Lo que significa es que no puede fijar el tiempo, el precio y el alcance. El enfoque ágil habitual es fijar el tiempo y el precio, y permitir que el alcance varíe de forma controlada.

En un proceso adaptativo, el cliente tiene un control mucho más detallado sobre el proceso de desarrollo de software. En cada iteración obtienen tanto para verificar el progreso como para alterar la dirección del desarrollo del software. Esto lleva a una relación mucho más cercana con los desarrolladores de software, una verdadera asociación comercial. Este nivel de participación no es para cada organización de cliente, ni para cada desarrollador de software; pero es esencial hacer que un proceso de adaptación funcione correctamente.

Todo esto proporciona una serie de ventajas para el cliente. Para empezar, obtienen un desarrollo de software mucho más receptivo. Un sistema utilizable, aunque mínimo, puede entrar en producción desde el principio. El cliente puede entonces cambiar sus capacidades de acuerdo con los cambios en el negocio, y también aprender de cómo se usa el sistema en realidad.

Tan importante como esto es una mayor visibilidad del verdadero estado del proyecto. El problema con los procesos predictivos es que la calidad del proyecto se mide por conformidad con el plan. Esto hace que sea difícil para las personas señalar cuándo la realidad y el plan divergen. El resultado común es un gran error en el programa al final del proyecto. En un proyecto ágil, hay una reelaboración constante del plan con cada iteración. Si hay malas noticias, tiende a llegar antes, cuando todavía hay tiempo para hacer algo al respecto. De hecho, este control de riesgos es una ventaja clave del desarrollo iterativo.

Los métodos ágiles llevan esto más lejos al mantener las longitudes de iteración pequeñas, pero también al ver estas variaciones de una manera diferente. Mary Poppendieck resumió esta diferencia en el punto de vista mejor para mí con su frase "Un cambio tardío en los requisitos es una ventaja competitiva". Creo que la mayoría de la gente ha notado que es muy difícil para las personas de negocios realmente entender lo que necesitan del software al principio. A menudo vemos que las personas aprenden durante el proceso qué elementos son valiosos y cuáles no. A menudo, las características más valiosas no son del todo obvias hasta que el cliente haya tenido la oportunidad de jugar con el software. Los métodos ágiles buscan aprovechar esto, alentar a los empresarios a conocer sus necesidades a medida que se construye el sistema, y ​​construir el sistema de tal forma que los cambios puedan incorporarse rápidamente.

Todo esto tiene una importancia importante lo que constituye un proyecto exitoso. Un proyecto predictivo a menudo se mide según cuán bien cumplió su plan. Un proyecto que está a tiempo y con costo se considera un éxito. Esta medida no tiene sentido para un entorno ágil. Para los agilistas, la cuestión es el valor comercial: ¿el cliente obtuvo un software que es más valioso para ellos que el costo que se le pone? Un buen proyecto predictivo funcionará según el plan, un buen proyecto ágil construirá algo diferente y mejor de lo que el plan original preveía.

**Poniendo a las personas primero**

Ejecutar un proceso de adaptación no es fácil. En particular, requiere un equipo de desarrolladores muy eficaz. El equipo debe ser eficaz tanto en la calidad de las personas como en la forma en que el equipo se combina. También hay una sinergia interesante: no solo la adaptabilidad requiere un equipo fuerte, la mayoría de los buenos desarrolladores prefieren un proceso adaptativo.

**Unidades de Programación Plug-Compatible**

Uno de los objetivos de las metodologías tradicionales es desarrollar un proceso donde las personas involucradas sean partes reemplazables. Con este proceso, puede tratar a las personas como recursos que están disponibles en varios tipos. Tienes un analista, algunos codificadores, algunos probadores, un gerente. Los individuos no son tan importantes, solo los roles son importantes. De esta manera, si planifica un proyecto, no importa qué analista y qué evaluadores obtenga, solo que sabe cuántos tiene para saber cómo afecta la cantidad de recursos a su plan.

Pero esto plantea una pregunta clave: ¿están las personas involucradas en el desarrollo de software partes reemplazables? Una de las características clave de los métodos ágiles es que rechazan esta suposición.

Quizás el rechazo más explícito de las personas como recursos es Alistair Cockburn. En su artículo Caracterización de las personas como componentes no lineales de primer orden en el desarrollo de software, señala que los procesos predecibles requieren componentes que se comporten de forma predecible. Sin embargo, las personas no son componentes predecibles. Además, sus estudios de proyectos de software le han llevado a la conclusión de que las personas son el factor más importante en el desarrollo de software.

En el título, [de su artículo] me refiero a las personas como "componentes". Así es como las personas son tratadas en la literatura de diseño de proceso / metodología. El error en este enfoque es que las "personas" son altamente variables y no lineales, con modos únicos de éxito y falla. Esos factores son de primer orden, no insignificantes. El hecho de que los diseñadores de procesos y metodologías no los tengan en cuenta contribuye al tipo de trayectorias de proyectos no planificados que tan a menudo vemos.

- [Cockburn no lineal]

Uno se pregunta si no la naturaleza del desarrollo de software funciona en contra de nosotros aquí. Cuando estamos programando una computadora, controlamos un dispositivo inherentemente predecible. Como estamos en este negocio porque somos buenos para hacer eso, somos idealmente idóneos para jugar cuando nos enfrentamos con seres humanos.

Aunque Cockburn es el más explícito en su visión centrada en las personas del desarrollo de software, la noción de personas primero es un tema común con muchos pensadores en software. El problema, con demasiada frecuencia, es que la metodología se ha opuesto a la noción de personas como factor de primer orden en el éxito del proyecto.

Esto crea un fuerte efecto de retroalimentación positiva. Si espera que todos sus desarrolladores sean unidades de programación compatibles con plug-ins, no intente tratarlos como individuos. Esto reduce la moral (y la productividad). La gente buena busca un lugar mejor para estar, y terminas con lo que deseas: unidades de programación compatibles con el enchufe.

Decidir que la gente es lo primero es una gran decisión, una que requiere mucha determinación para avanzar. La noción de personas como recursos está profundamente arraigada en el pensamiento empresarial, y sus raíces se remontan al impacto del enfoque de Gestión Científica de Frederick Taylor. Al dirigir una fábrica, este enfoque taylorista puede tener sentido. Pero para el trabajo altamente creativo y profesional, que creo que es el desarrollo de software, esto no es válido. (Y, de hecho, la fabricación moderna también se está alejando del modelo taylorista).

**Los programadores son profesionales responsables**

Una parte clave de la noción taylorista es que las personas que hacen el trabajo no son las personas que mejor pueden encontrar la mejor manera de hacer ese trabajo. En una fábrica esto puede ser cierto por varias razones. Parte de esto es que muchos trabajadores de fábrica no son las personas más inteligentes o creativas, en parte porque la tensión entre la gerencia y los trabajadores es que la gerencia gana más dinero cuando los trabajadores ganan menos.

La historia reciente nos muestra cada vez más que esto es falso para el desarrollo de software. Las personas cada vez más brillantes y capaces se sienten atraídas por el desarrollo de software, atraídas tanto por su brillo como por sus grandes recompensas. (Ambos me tentaron de la ingeniería electrónica). A pesar de la recesión de principios de los años 00, todavía hay una gran cantidad de talento y creatividad en el desarrollo de software.

(Puede haber un efecto generacional aquí. Algunas pruebas anecdóticas me hacen preguntarme si las personas más brillantes se han aventurado en la ingeniería del software en los últimos quince años más o menos. De ser así, esta sería una razón de por qué existe tal culto a la juventud en el negocio de la informática, como la mayoría de los cultos, tiene que haber un grano de verdad en ello).

Cuando desea contratar y retener a personas buenas, debe reconocer que son profesionales competentes. Como tales, son las mejores personas para decidir cómo llevar a cabo su trabajo técnico. La noción taylorista de un departamento de planificación separado que decide cómo hacer las cosas solo funciona si los planificadores entienden cómo hacer el trabajo mejor que los que lo hacen. Si tienes personas brillantes y motivadas que hacen el trabajo, entonces esto no se cumple.

**Gestionar un proceso orientado a las personas**

La orientación hacia las personas se manifiesta de diferentes maneras en los procesos ágiles. Conduce a diferentes efectos, no todos son consistentes.

Uno de los elementos clave es aceptar el proceso en lugar de imponer un proceso. A menudo, los procesos de software son impuestos por figuras administrativas. Como tales, a menudo son resistidos, particularmente cuando las figuras administrativas han tenido una cantidad significativa de tiempo fuera del desarrollo activo. Aceptar un proceso requiere compromiso, y como tal necesita la participación activa de todo el equipo

Esto termina con el resultado interesante de que solo los propios desarrolladores pueden elegir seguir un proceso de adaptación. Esto es particularmente cierto para XP, que requiere mucha disciplina para ejecutar. Crystal se considera a sí mismo como un enfoque menos disciplinado que es apropiado para un público más amplio.

Otro punto es que los desarrolladores deben poder tomar todas las decisiones técnicas. XP llega al corazón de esto, donde en su proceso de planificación establece que solo los desarrolladores pueden hacer estimaciones sobre cuánto tiempo les llevará hacer algo de trabajo.

Tal liderazgo técnico es un gran cambio para muchas personas en puestos directivos. Tal enfoque requiere compartir la responsabilidad donde los desarrolladores y la gerencia tienen un lugar igual en el liderazgo del proyecto. Tenga en cuenta que digo igual. La administración todavía juega un rol, pero reconoce la experiencia de los desarrolladores.

Una razón importante para esto es la tasa de cambio de tecnología en nuestra industria. Después de unos años, el conocimiento técnico se vuelve obsoleto. Esta vida media de habilidades técnicas no tiene paralelo en ninguna otra industria. Incluso los técnicos tienen que reconocer que ingresar a la gerencia significa que sus habilidades técnicas se marchitarán rápidamente. Los ex desarrolladores deben reconocer que sus habilidades técnicas desaparecerán rápidamente y deben confiar y confiar en los desarrolladores actuales.

**La dificultad de medición**

Si tiene un proceso donde las personas que dicen cómo debe hacerse el trabajo son diferentes de las personas que realmente lo hacen, los líderes necesitan alguna forma de medir qué tan efectivos son los hacedores. En Gestión científica hubo un fuerte impulso para desarrollar enfoques objetivos para medir el rendimiento de las personas.

Esto es particularmente relevante para el software debido a la dificultad de aplicar medidas al software. A pesar de nuestros mejores esfuerzos, no podemos medir las cosas más simples sobre el software, como la productividad. Sin buenas medidas para estas cosas, cualquier tipo de control externo está condenado.

Introducir una gestión mesurada sin buenas medidas conduce a sus propios problemas. Robert Austin hizo una excelente discusión de esto. Él señala que al medir el rendimiento, debe obtener todos los factores importantes bajo medición. Cualquier cosa que falta tiene el resultado inevitable de que los hacedores alterarán lo que hacen para producir las mejores medidas, incluso si eso reduce claramente la verdadera efectividad de lo que hacen. Esta disfunción de medición es el talón de Aquiles de la gestión basada en la medición.

La conclusión de Austin es que debe elegir entre la administración basada en la medición y la administración del delegatorio (donde los hacedores deciden cómo hacer el trabajo). La gestión basada en la medición es más adecuada para el trabajo simple repetitivo, con bajos requisitos de conocimiento y resultados fácilmente medibles, exactamente lo opuesto al desarrollo de software.

El objetivo de todo esto es que los métodos tradicionales han funcionado bajo el supuesto de que la gestión basada en la medición es la forma más eficiente de gestión. La comunidad ágil reconoce que las características del desarrollo de software son tales que la gestión basada en la medición conduce a niveles muy altos de disfunción de medición. En realidad, es más eficiente usar un estilo de administración delegatorio, que es el tipo de enfoque que está en el centro del punto de vista agilista.

**El papel del liderazgo empresarial**

Pero los técnicos no pueden hacer todo el proceso ellos mismos. Necesitan orientación sobre las necesidades comerciales. Esto lleva a otro aspecto importante de los procesos adaptativos: necesitan un contacto muy estrecho con la experiencia empresarial.

Esto va más allá de la participación de la mayoría de los proyectos en el rol del negocio. Los equipos ágiles no pueden existir con comunicación ocasional. Necesitan un acceso continuo a la experiencia empresarial. Además, este acceso no es algo que se maneje a nivel de gestión, es algo que está presente para cada desarrollador. Como los desarrolladores son profesionales capaces en su propia disciplina, necesitan poder trabajar en igualdad de condiciones con otros profesionales de otras disciplinas.

Una gran parte de esto, por supuesto, se debe a la naturaleza del desarrollo adaptativo. Como toda la premisa del desarrollo adaptativo es que las cosas cambian rápidamente, necesita un contacto constante para avisar a todos de los cambios.

No hay nada más frustrante para un desarrollador que ver que su arduo trabajo se desperdicia. Por lo tanto, es importante asegurarse de que exista una experiencia comercial de buena calidad que esté disponible tanto para el desarrollador como de la calidad suficiente para que el desarrollador pueda confiar en ellos.

**El proceso de auto-adaptación**

Hasta ahora he hablado sobre la adaptabilidad en el contexto de un proyecto que adapta su software con frecuencia para cumplir con los requisitos cambiantes de sus clientes. Sin embargo, hay otro ángulo de adaptabilidad: el del proceso que cambia con el tiempo. Un proyecto que comienza utilizando un proceso adaptativo no tendrá el mismo proceso un año después. Con el tiempo, el equipo encontrará lo que funciona para ellos y modificará el proceso para que encaje.

La primera parte de la auto-adaptabilidad es la revisión periódica del proceso. Usualmente haces esto con cada iteración. Al final de cada iteración, tenga una reunión corta y hágase las siguientes preguntas (extraídas de Norm Kerth)

¿Qué hicimos bien?

¿Qué hemos aprendido?

¿Qué podemos hacer mejor?

¿Qué nos desconcierta?

Estas preguntas te llevarán a ideas para cambiar el proceso para la próxima iteración. De esta forma, un proceso que comienza con problemas puede mejorar a medida que el proyecto avanza, adaptándose mejor al equipo que lo utiliza.

Si la auto-adaptabilidad ocurre dentro de un proyecto, es aún más marcada en toda la organización. Una consecuencia de la auto-adaptabilidad es que nunca se debe esperar encontrar una sola metodología corporativa. En cambio, cada equipo no solo debe elegir su propio proceso, sino que también debe afinar activamente su proceso a medida que avanza con el proyecto. Si bien los procesos publicados y la experiencia de otros proyectos pueden actuar como una inspiración y una línea de base, la responsabilidad profesional de los desarrolladores es adaptar el proceso a la tarea en cuestión.

**Sabores del desarrollo ágil**

El término 'ágil' se refiere a una filosofía de desarrollo de software. Bajo este amplio paraguas se encuentran muchos más enfoques específicos, como Extreme Programming(programación extrema), Scrum(melé), Lean Development(desarrollo delgado), etc. Cada uno de estos enfoques más específicos tiene sus propias ideas, comunidades y líderes. Cada comunidad es un grupo distinto, pero para llamarse correctamente ágil debe seguir los mismos principios generales. Cada comunidad también toma prestadas ideas y técnicas entre sí. Muchos profesionales se mueven entre diferentes comunidades y difunden diferentes ideas; en general, es un ecosistema complicado pero vibrante.

Hasta ahora he dado mi opinión sobre la imagen general de mi definición de ágil. Ahora quiero presentar algunas de las diferentes comunidades ágiles. Solo puedo ofrecer una descripción general rápida aquí, pero sí incluyo referencias para que pueda profundizar aún más si lo desea.

Como estoy por comenzar a dar más referencias, este es un buen punto para señalar algunas fuentes de información general sobre métodos ágiles. El centro web es Agile Alliance, una organización sin ánimo de lucro creada para fomentar e investigar el desarrollo de software ágil. Para libros, sugeriría descripciones de Alistair Cockburn y Jim Highsmith. El libro de Craig Larman sobre el desarrollo ágil contiene una historia muy útil de desarrollo iterativo. Para ver más de mis puntos de vista sobre métodos ágiles, mire las secciones apropiadas de mis artículos y blog.

La siguiente lista de ninguna manera está completa. Refleja una selección personal de los sabores de ágil que más me han interesado e influenciado durante la última década más o menos.

**Manifiesto Ágil**

El término "ágil" fue secuestrado para esta actividad a principios de 2001, cuando un grupo de personas que habían participado activamente en este trabajo se unieron para intercambiar ideas y crearon el Manifiesto para el Desarrollo Agile de Software.

Antes de este taller, varios grupos diferentes habían estado desarrollando ideas similares sobre el desarrollo de software. La mayoría, pero no todas, de este trabajo provenía de la comunidad de software orientada a objetos que durante mucho tiempo había defendido los enfoques de desarrollo iterativo. Este ensayo fue escrito originalmente en 2000 para tratar de unir estos diversos hilos. En ese momento no había un nombre común para estos enfoques, pero el apodo 'peso ligero' había crecido a su alrededor. Muchas de las personas involucradas no sentían que este fuera un buen término, ya que no transmitía con exactitud la esencia de lo que trataban estos enfoques.

Hubo algunos que hablaron sobre cuestiones más amplias en estos enfoques en 2000 en un taller organizado por Kent Beck en Oregon. Aunque este taller se centró en la Programación Extrema (la comunidad que en ese momento había ganado la mayor atención) varios no XPers habían asistido. Una de las discusiones que surgieron fue si era mejor para XP ser un movimiento amplio o concreto. Kent prefería una comunidad cohesiva más centrada.

El taller fue organizado, si no recuerdo mal, principalmente por Jim Highsmith y Bob Martin. Se contactaron con personas que sentían que estaban activas en comunidades con estas ideas similares y reunieron a diecisiete de ellas para el taller de Snowbird(cocainómanos). La idea inicial era simplemente reunirnos y construir una mejor comprensión de los enfoques de los demás. Robert Martin estaba ansioso por obtener una declaración, un manifiesto que se podría utilizar para unir a la industria detrás de este tipo de técnicas. También decidimos que queríamos elegir un nombre para que actuara como un nombre paraguas para los diversos enfoques.

Durante el curso del taller, decidimos usar 'ágil' como el nombre del paraguas, y se obtuvieron valores que forman parte del manifiesto. La sección de principios se inició en el taller, pero en su mayoría se desarrolló en una wiki posterior.

El esfuerzo claramente golpeó un nervio, creo que todos estábamos muy sorprendidos por el grado de atención y aprecio que recibió el manifiesto. Aunque el manifiesto no es una definición rigurosa de ágil, proporciona una declaración de enfoque que ayuda a concentrar las ideas. Poco después de que terminamos el manifiesto Jim Highsmith y yo escribimos un artículo para la revista SD que proporcionó algunos comentarios al manifiesto.

Más tarde ese año, la mayoría de los diecisiete que escribieron el manifiesto volvieron a reunirse, junto con algunos otros, en OOPSLA 2001. Hubo una sugerencia de que los autores del manifiesto deberían comenzar algún movimiento ágil en curso, pero los autores acordaron que fueron solo las personas que asistieron a ese taller y produjeron ese manifiesto. No había forma de que ese grupo pudiera reclamar el liderazgo de toda la comunidad ágil. Habíamos ayudado a lanzar el barco y debería dejarlo ir para que quien quisiera navegar en ella lo hiciera. Así que ese fue el final de los diecisiete autores manifiestos como un cuerpo organizado.

Un siguiente paso que siguió, con la participación activa de muchos de estos autores, fue la formación de la alianza ágil. Este grupo es un grupo sin fines de lucro destinado a promover e investigar métodos ágiles. Entre otras cosas, patrocina una conferencia anual en los Estados Unidos.

**XP (programación extrema)**

Durante la temprana popularidad de los métodos ágiles a fines de la década de 1990, Extreme Programming fue la que recibió la mayor atención. En muchos sentidos todavía lo hace.

Las raíces de XP se encuentran en la comunidad de Smalltalk, y en particular la estrecha colaboración de Kent Beck y Ward Cunningham a finales de los años ochenta. Ambos refinaron sus prácticas en numerosos proyectos durante la década de los 90, extendiendo sus ideas de un enfoque de desarrollo de software adaptable y orientado a las personas.

Kent continuó desarrollando sus ideas durante los trabajos de consultoría, en particular el proyecto Chrysler C3, que desde entonces se conoce como el proyecto de creación de programación extrema. Comenzó a usar el término "programación extrema" alrededor de 1997. (C3 también marcó mi contacto inicial con Extreme Programming y el comienzo de mi amistad con Kent).

A finales de la década de 1990, la palabra Extreme Programming se extendió, inicialmente a través de descripciones en grupos de noticias y el wiki de Ward Cunningham, donde Kent y Ron Jeffries (un colega en C3) dedicaron mucho tiempo a explicar y debatir las diversas ideas. Finalmente, se publicaron varios libros hacia fines de los 90 y comienzos de los 00 que detallaron algunos aspectos del enfoque. La mayoría de estos libros tomaron el libro blanco de Kent Beck como su base. Kent produjo una segunda edición del libro blanco en 2004, que fue una rearticulación significativa del enfoque.

XP comienza con cinco valores (Comunicación, Retroalimentación, Simplicidad, Coraje y Respeto). Luego los elabora en catorce principios y nuevamente en veinticuatro prácticas. La idea es que las prácticas son cosas concretas que un equipo puede hacer día a día, mientras que los valores son el conocimiento y la comprensión fundamentales que sustentan el enfoque. Los valores sin prácticas son difíciles de aplicar y se pueden aplicar de muchas maneras, por lo que es difícil saber por dónde empezar. Las prácticas sin valores son actividades rutinarias sin un propósito. Se necesitan tanto valores como prácticas, pero existe una gran brecha entre ellos: los principios ayudan a cerrar esa brecha. Muchas de las prácticas de XP son técnicas antiguas, probadas y probadas, pero muchas veces son olvidadas por muchos, incluida la mayoría de los procesos planificados. Además de resucitar estas técnicas, XP las entreteje en un todo sinérgico donde cada una se refuerza con las otras y se da un propósito por los valores.

Uno de los más llamativos, e inicialmente atractivo para mí, es su fuerte énfasis en las pruebas. Si bien todos los procesos mencionan las pruebas, la mayoría lo hace con un énfasis bastante bajo. Sin embargo, XP pone las pruebas en la base del desarrollo, con cada programador escribiendo pruebas mientras escribe su código de producción. Las pruebas se integran en un proceso continuo de integración y construcción que proporciona una plataforma altamente estable para el desarrollo futuro. El enfoque de XP aquí, a menudo descrito bajo el título de Test Driven Development (TDD)(desarrollo impulsado por prueba), ha sido influyente incluso en lugares que no han adoptado mucho más de XP.

Hay una gran cantidad de publicaciones sobre programación extrema. Un área de confusión, sin embargo, es el cambio entre la primera y la segunda edición del libro blanco. Dije anteriormente que la segunda edición es una "rearticulación" de la programación extrema, ya que el enfoque sigue siendo el mismo, pero se describe con un estilo diferente. La primera edición (con cuatro valores, doce prácticas y algunos principios importantes pero ignorados en su mayoría) tuvo una gran influencia en la industria del software y la mayoría de las descripciones de programación extrema se escribieron en base a la descripción de la primera edición. Tenga esto en cuenta al leer material sobre XP, especialmente si fue preparado antes de 2005. De hecho, la mayoría de las descripciones web comunes de XP se basan en la primera edición.

El lugar natural para descubrir más es la segunda edición del libro blanco. Este libro explica los antecedentes y las prácticas de XP en un paquete corto (160 páginas). Kent Beck editó una serie de libros multicolores sobre programación extrema en el cambio de siglo, si se ve obligado a elegir uno para sugerir que vaya por el morado, recuerde que al igual que la mayoría del material, está basado en la primera edición.

Hay un montón de material en la web sobre XP, pero la mayor parte se basa en la primera edición. Una de las pocas descripciones que conozco que tiene en cuenta la segunda edición es un documento sobre The New XP (PDF) de Michele Marchesi, que fue el anfitrión de las conferencias originales de XP en Cerdeña. Para la discusión sobre XP hay una lista de correo yahoo.

Mi participación en los primeros días y las amistades dentro de la comunidad de XP significa que tengo una familiaridad, afición y predisposición distintas hacia XP. Creo que su influencia se debe a unir los principios del desarrollo ágil con un conjunto sólido de técnicas para llevarlos a cabo. Gran parte de las primeras obras sobre ágil descuidaron a la última, lo que suscitó dudas sobre si las ideas ágiles eran realmente posibles. XP proporcionó las herramientas mediante las cuales las esperanzas de agilidad se podían realizar.

**scrum**

Scrum también se desarrolló en los años 80 y 90 principalmente con los círculos de desarrollo de OO como una metodología de desarrollo altamente iterativa. Sus desarrolladores más conocidos fueron Ken Schwaber, Jeff Sutherland y Mike Beedle.

Scrum se concentra en los aspectos administrativos del desarrollo de software, dividiendo el desarrollo en iteraciones de treinta días (llamadas 'sprints') y aplicando un monitoreo y control más estrecho con las reuniones diarias de scrum. Pone mucho menos énfasis en las prácticas de ingeniería y muchas personas combinan su enfoque de gestión de proyectos con prácticas de ingeniería de programación extrema. (Las prácticas de administración de XP no son muy diferentes). Ken Schwaber es uno de los defensores más activos de Scrum, su sitio web es un buen lugar para comenzar a buscar más información y su libro es probablemente la mejor referencia.

**Cristal**

Alistair Cockburn ha sido durante mucho tiempo una de las principales voces en la comunidad ágil. Desarrolló la familia Crystal de métodos de desarrollo de software como un grupo de enfoques adaptados a equipos de diferentes tamaños. Crystal es visto como una familia porque Alistair cree que se requieren diferentes enfoques ya que los equipos varían en tamaño y la criticidad de los errores cambia.

A pesar de sus variaciones, todos los enfoques de cristal comparten características comunes. Todos los métodos de cristal tienen tres prioridades: seguridad (en el resultado del proyecto), eficiencia, habitabilidad (los desarrolladores pueden vivir con el cristal). También comparten propiedades comunes, de las cuales las tres más importantes son: Entrega Frecuente, Mejoramiento Reflexivo y Comunicación Estrecha.

La prioridad de habitabilidad es una parte importante de la mentalidad cristalina. La búsqueda de Alistair (como yo lo veo) busca cuál es la menor cantidad de proceso que puede hacer y aún tiene éxito con una suposición subyacente de baja disciplina que es inevitable con los humanos. Como resultado, Alistair ve a Crystal requiriendo menos disciplina que la programación extrema, intercambiando menos eficiencia por una mayor habitabilidad y menores posibilidades de fracaso.

A pesar del perfil de Crystal, no hay una descripción exhaustiva de todas sus manifestaciones. El más bien descrito es Crystal Clear, que tiene una descripción de libro moderna. También hay una wiki para más material y discusión de Crystal.

**Prueba conducida por contexto**

Desde el principio, los desarrolladores de software han sido los impulsores de la comunidad ágil. Sin embargo, muchas otras personas están involucradas en el desarrollo de software y se ven afectadas por este nuevo movimiento. Un grupo obvio de este tipo son los probadores, que a menudo viven en un mundo muy contenido por el pensamiento de cascada. Con directrices comunes que establecen que la función de las pruebas es garantizar la conformidad del software con las especificaciones escritas por adelantado, el papel de los evaluadores en un mundo ágil no está nada claro.

Como resultado, varias personas en la comunidad de prueba han estado cuestionando gran parte del pensamiento de pruebas convencionales durante bastante tiempo. Esto ha llevado a un grupo conocido como prueba basada en contexto. La mejor descripción de esto es el libro Lessons Learned in Software Testing. Esta comunidad también es muy activa en la web, eche un vistazo a los sitios alojados por Brian Marick (uno de los autores del manifiesto ágil), Brett Pettichord, James Bach y Cem Kaner.

**Lean Development (desarrollo de apoyo)**

Recuerdo que hace unos años daba una charla sobre métodos ágiles en la conferencia de Desarrollo de Software y hablaba con una mujer entusiasta sobre los paralelos entre las ideas ágiles y el movimiento esbelto en la fabricación. Mary Poppendieck (y su esposo Tom) han llegado a ser partidarios activos de la comunidad ágil, en particular mirando las superposiciones e inspiraciones entre la producción ajustada y el desarrollo de software.

El movimiento magro en la fabricación fue iniciado por Taiichi Ohno en Toyota y a menudo se conoce como el sistema de producción de Toyota. La producción Lean fue una inspiración para muchos de los primeros agilistas. Los Poppendiecks son los más destacados al describir cómo interactúan estas ideas. En general, soy muy cauteloso con este tipo de razonamiento por analogía, de hecho, la separación de ingeniería entre diseño y construcción nos metió en este lío en primer lugar. Sin embargo, las analogías pueden conducir a buenas ideas y creo que las ideas lean han introducido muchas ideas útiles y herramientas en el movimiento ágil.

El libro y el sitio web de Poppendiecks son los puntos de partida obvios para obtener más información.

**Proceso racional unificado**

Otro proceso bien conocido que ha salido de la comunidad orientada a objetos es el Proceso Unificado Racional (a veces simplemente denominado Proceso Unificado). La idea original era que, al igual que los lenguajes de modelado unificado UML, la UP podría unificar los procesos de software. Dado que RUP apareció casi al mismo tiempo que los métodos ágiles, hay mucha discusión sobre si los dos son compatibles.

RUP es una gran colección de prácticas y es realmente un marco de procesos en lugar de un proceso. En lugar de dar un solo proceso para el desarrollo de software, busca proporcionar un conjunto común de prácticas para que los equipos elijan para un proyecto individual. Como resultado, el primer paso de un equipo usando RUP debe ser definir su proceso individual, o como lo llama RUP, un caso de desarrollo.

Los aspectos comunes clave de RUP es que se usa en casos de uso (el desarrollo se basa en características visibles para el usuario), iterativo y centrado en la arquitectura (hay una prioridad para construir una arquitectura desde el principio que durará hasta el final del proyecto).

Mi experiencia con RUP es que su problema es su variabilidad infinita. He visto descripciones del uso de RUP que van desde la cascada rígida con 'iteraciones de análisis' hasta una imagen ágil perfecta. Me llamó la atención que el deseo de las personas de comercializar el RUP como el proceso único llevó a un resultado donde las personas pueden hacer prácticamente cualquier cosa y llamarlo RUP, lo que hace que RUP sea una frase sin sentido.

A pesar de todo esto, hay algunas personas muy fuertes en la comunidad RUP que están muy alineadas con el pensamiento ágil. Me ha impresionado en toda mi reunión con Phillippe Kruchten y su libro es el mejor punto de partida para RUP. Craig Larman también ha desarrollado descripciones de cómo trabajar con RUP en un estilo ágil en su popular libro introductorio sobre diseño OO.

**¿Deberías ir ágil?**

Usar un método ágil no es para todos. Hay varias cosas a tener en cuenta si decide seguir este camino. Sin embargo, creo que estas metodologías son ampliamente aplicables y debería ser utilizadas por más personas de las que actualmente las consideran.

En el entorno actual, la metodología más común es el código y la solución. Aplicar más disciplina que caos es casi seguro que ayudará, y el enfoque ágil tiene la ventaja de que es mucho menos que usar un método de peso pesado. Aquí el peso ligero de los métodos ágiles es una ventaja. Es más probable que se sigan procesos más simples cuando no está acostumbrado a ningún proceso.

Para alguien nuevo en métodos ágiles, la pregunta es por dónde empezar. Al igual que con cualquier tecnología o proceso nuevo, debe hacer su propia evaluación. Esto le permite ver cómo encaja en su entorno. Como resultado, gran parte de mis consejos siguen que he dado para otros enfoques nuevos, trayendo recuerdos de la primera vez que hablé sobre técnicas orientadas a objetos.

El primer paso es encontrar proyectos adecuados para probar métodos ágiles. Dado que los métodos ágiles son fundamentalmente orientados a las personas, es esencial que comiences con un equipo que quiera probar y trabajar de manera ágil. No solo es más difícil trabajar con un equipo reacio, la imposición de métodos ágiles a las personas renuentes está fundamentalmente en desacuerdo con toda la noción de desarrollo ágil.

También es valioso tener clientes (aquellos que necesitan el software) que quieran trabajar de esta manera colaborativa. Si los clientes no colaboran, entonces no verá todas las ventajas de un proceso adaptativo. Habiendo dicho eso, hemos encontrado en varias ocasiones que hemos trabajado con clientes que no quisieron colaborar, pero cambiaron de opinión en los primeros meses a medida que comenzaron a comprender el enfoque ágil.

Mucha gente afirma que los métodos ágiles no se pueden usar en proyectos grandes. Nosotros (ThoughtWorks) hemos tenido un buen éxito con proyectos ágiles con alrededor de 100 personas y varios continentes. A pesar de esto, sugiero elegir algo más pequeño para empezar. Los proyectos grandes son intrínsecamente más difíciles de todos modos, por lo que es mejor comenzar a aprender en un proyecto de un tamaño más manejable.

Algunas personas aconsejan elegir un proyecto con poco impacto comercial para empezar, de esa manera si algo sale mal entonces hay menos daño. Sin embargo, un proyecto sin importancia a menudo es una prueba pobre ya que a nadie le importa mucho el resultado. Prefiero aconsejar a la gente que tome un proyecto que es un poco más crítico de lo que te sientes cómodo.

Quizás lo más importante que puede hacer es buscar a alguien con más experiencia en métodos ágiles para ayudarlo a aprender. Cada vez que alguien hace algo nuevo inevitablemente comete errores. Encuentre a alguien que ya haya cometido muchos errores para evitar hacerlo usted mismo. De nuevo, esto es algo cierto para cualquier nueva tecnología o técnica, un buen mentor vale su peso en oro. Por supuesto, este consejo es útil ya que ThoughtWorks (trabajo mental)y muchos de mis amigos en la industria hacen mentores en métodos ágiles. Eso no altera el hecho de que creo firmemente en la importancia de encontrar un buen mentor.

Y una vez que hayas encontrado un buen mentor, sigue sus consejos. Es muy fácil adivinar gran parte de esto y he aprendido por experiencia que muchas técnicas realmente no se pueden entender hasta que hayas hecho un intento razonable de probarlas. Uno de los mejores ejemplos que escuché fue un cliente nuestro que decidió probar programación extrema durante un par de meses. Durante ese período, dejaron en claro que harían lo que dijera el mentor, incluso si pensaban que era una mala idea. Al final de ese período de prueba, se detendrían y decidirían si querían continuar con alguna de las ideas o volver a la forma de trabajo anterior. (En caso de que se lo pregunte, decidieron continuar con XP).

Una de las preguntas abiertas sobre métodos ágiles es dónde se encuentran las condiciones de contorno. Uno de los problemas con cualquier técnica nueva es que no está realmente al tanto de dónde están las condiciones de frontera hasta que las cruza y falla. Los métodos ágiles todavía son demasiado jóvenes para ver acciones suficientes como para tener una idea de dónde están los límites. Esto se complica aún más por el hecho de que es tan difícil decidir qué significan el éxito y el fracaso en el desarrollo del software, así como demasiados factores variables para identificar fácilmente el origen de los problemas.

Entonces, ¿dónde no deberías usar un método ágil? Creo que principalmente se trata de la gente. Si las personas involucradas no están interesadas en el tipo de colaboración intensa que requiere el trabajo ágil, entonces va a ser una gran lucha conseguir que trabajen con ella. En particular, creo que esto significa que nunca debes tratar de imponer un trabajo ágil a un equipo que no quiere probarlo.

Ha habido mucha experiencia con métodos ágiles en los últimos diez años. En ThoughtWorks(trabajo mental) siempre utilizamos un enfoque ágil si nuestros clientes están dispuestos, la mayoría del tiempo lo son. Yo (y nosotros) seguimos siendo grandes admiradores de esta forma de trabajar.