¿Aún en crisis?

Algunos mitos y desafíos de la Ingeniería de Software

Rubby Casallas

Hace ya casi 40 años que se trató por primera vez el tema de la "crisis del *software*" y se acuñó la expresión *Ingeniería de Software*. En ese momento fue identificado un gran número de problemas derivados de la utilización de métodos Ad Hoc (el ciclo programar-corregir). Hablar entonces de Ingeniería de *Software*, era expresar el deseo de contar con prácticas más disciplinadas que remplazaran la manera artesanal como se construían las aplicaciones.

Desde entonces ha habido evolución y progresos en varios aspectos de las prácticas de la construcción de *software*. Pero también ha habido muchos cambios en las tecnologías y un aumento en las expectativas de los clientes y de los usuarios. La aparición de Internet complicó, para los desarrolladores, la implementación de requerimientos no funcionales como la escalabilidad, la seguridad, la tolerancia a fallas o el manejo transaccional en esquemas Negocio-Negocio. Las necesidades crecientes de *software*, el cambio en la manera de hacer negocios, el tamaño cada vez mayor de las aplicaciones y la complejidad de los requerimientos no funcionales hacen de la construcción de *software* una disciplina retadora. Significa que la crisis, aunque distinta, sigue ahí.

Esta crisis se evidencia porque una gran cantidad de los proyectos fallan, sobrepasan los costos y los tiempos estimados. Hay muchos clientes y usuarios insatisfechos porque la aplicación que se entregó no satisface

los requerimientos o porque sencillamente no funciona bien. Es usual encontrarse con proyectos que nunca se terminan, nunca se utilizan o el costo de mantenerlos es muy alto, dada la dificultad de modificarlos ya sea para corregir, extender o adaptar.

El gran mito

Dentro del deseo de acercarse a la ingeniería para poder contar con prácticas más disciplinadas, estaba el hecho de querer definir los proyectos de construcción de *software* de manera similar a como se definen muchos proyectos de otras ingenierías.

Consideremos la construcción de un puente como un ejemplo, que aunque descrito de manera muy simplificada, permite ilustrar la idea. Cuando se decide construirlo hay claridad sobre la necesidad y el propósito. Seguramente se conocen el tráfico esperado, las condiciones del terreno, el peso de la superestructura, las cargas a las cuales estará sometido y muchos detalles más. Conocidas las restricciones y el contexto, le sigue la actividad de diseño, en la cual los ingenieros especializados realizan cálculos, analizan alternativas técnicas y económicas, hacen simulaciones y verificaciones, además de elaborar los planos de construcción.

Cuando el diseño está terminado, se planifican en detalle las tareas de construcción en diagramas de Gantt, se establece el orden de las mismas (localización topográfica, adecuación del sitio, excavación, cimentaciones, estribos, vigas, pavimento, protección lateral, entre otras). Una vez definidas y conocidos los rendimientos en la ejecución de cada una de las actividades se estima la duración de las tareas, el tiempo total de la construcción y se asignan los recursos de acuerdo con los roles. Todo esto pasa a ser parte contractual del proyecto, de estricto cumplimiento por parte del contratista que ejecuta la obra.

En ejemplos como este hay que resaltar al menos cuatro supuestos básicos: (1) los requerimientos son claros desde el principio; (2) los diseños se pueden elaborar de manera completa (y validar) antes de la construcción; (3) el proceso de construcción es predecible; y, por último, (4) hay gente bien entrenada y especializada para la realización de las distintas labores.

El mito es falso

Infortunadamente, en los proyectos de construcción de *software* los supuestos mencionados no son válidos por distintas razones. (1) Los requerimientos no son claros ni completos ni al principio del proyecto y, en mu-

chas ocasiones, ni siquiera al final. (2) Los diseños no son completos, no se sabe como expresarlos ni cómo validarlos. Se quedan sin respuesta las preguntas: ¿este diseño permitirá la construcción de una solución para los requerimientos (incompletos y ambiguos) dados? ¿Este es un buen diseño (viable, eficiente, evaluable, adaptable)? (3) En este contexto, el proceso de programación no puede ser predecible. ¿Cómo prever cuántas veces el cliente va a cambiar de opinión? ¿Cómo prever en qué momento nos vamos a dar cuenta que al diseño le faltaba considerar la escalabilidad? (4) Además, es difícil conseguir personas bien entrenadas y especializadas en las distintas tecnologías, metodologías y procesos. Tampoco es clara la separación de roles: ¿Quien analiza es capaz de programar? ¿Quien dirige el proyecto sabe de dirección de proyectos? ¿Quien dirige el proyecto sabe de las tecnologías que estamos usando? ¿Quién diseña prueba? ¿Quien programa habla con el cliente?

Entonces, la pregunta obvia es: ¿por qué razón habría de funcionar, en el caso del *software*, un esquema de construcción en ingeniería donde ninguno de los supuestos sobre los que se basa es cierto? La respuesta es que no hay razón y eso lo comprobamos en sobre costos, frustración e insatisfacción. Entonces, ¿cuál es la alternativa? Vamos a analizar una posible opción muy prometedora, pero también con varios desafíos que deben ser superados.

Juntar cosas conocidas y desmitificar otras

Hoy por hoy es popular, y en parte gracias al proceso unificado (UP) y en particular al RUP[1], que el desarrollo por ciclos es una buena estrategia de construcción de *software*. El proyecto global se desarrolla por ciclos consecutivos y al finalizar uno de estos, siempre hay un producto funcional. El conjunto de requerimientos que se adicionan en un ciclo se denomina un incremento. El proceso de *software* por equipos (*TSP: Team software Process*) [2], también basa su estrategia en ciclos incrementales. Pero es en las metodologías ágiles en las que la idea de construcción por incrementos toma más fuerza. El ejemplo más destacado es eXtreme Programming (XP) [3], en donde en un ciclo (aquí se llaman iteraciones) se desarrolla un incremento (aquí se llaman historias de usuario), en tan sólo una o dos semanas.

Es importante resaltar que en un ciclo de desarrollo debe producirse un *software* que puede ser utilizado por un usuario final. No se trata de definir

un ciclo para hacer la persistencia del sistema, otro para incluir un módulo de manejo de sonido ni de ir construyendo capas hasta tener el resultado final con todos los requerimientos.

Si volvemos a nuestro ejemplo de la construcción del puente con dos calzadas vehiculares y dos peatonales, es como si dividiéramos la construcción en las siguientes etapas: primero, construir un puente peatonal; luego, incrementarlo con una calzada para vehículos; posteriormente, construir los separadores viales; luego un incremento para adicionar el alumbrado y las protecciones. Este proceso se repetiría, añadiendo dos incrementos, para completar la construcción del puente con la otra calzada vehicular y peatonal.

Por supuesto que no se necesita saber de construcción de puentes para entender que en este caso el desarrollo con incrementos así definidos es absurdo. En el caso del *software* no lo es, y en lo que resta, vamos a analizar en detalle cómo este desarrollo incremental permite que propongamos soluciones concretas para los problemas derivados de los cuatro supuestos mencionados antes, los cuales se han convertido en mitos en la ingeniería de *software*.

Los requerimientos

Mito: "se deben tener todos los requerimientos del sistema claramente definidos antes de empezar el proyecto para que éste sea exitoso".

El mito es falso y son varias las razones. Watts Humphrey en su libro "Managing the *software* Process" [4] explica muy bien varias de ellas. Puede que se conozcan de manera global los objetivos que pretende el *software*, pero nunca se podrá tener el detalle de los requerimientos al inicio del proyecto. Es normal que el cliente y los usuarios no sepan los detalles de lo que quieren. También es normal que una vez la aplicación esté en uso se vean nuevas posibilidades o que se cuestionen requerimientos que ya se implementaron. Los requerimientos son cambiantes porque las condiciones cambian, las expectativas aumentan, los detalles solo se aclaran con la utilización. La documentación de los detalles es difícil y en aplicaciones grandes es imposible. Hay un problema de comunicación entre quien necesita y va a utilizar el *software* y quien lo va a desarrollar. Hay una gran frustración cuando después de seis o más meses de levantamiento de información, en donde se produjeron páginas y páginas de texto escritas en lenguaje natural, difíciles de validar, todo empiece a cambiar.

La estrategia de desarrollo incremental parte de la base de que los requerimientos son incompletos y que es normal no saber lo que queremos y que no podamos definir todos los detalles. La estrategia es manejar pocos requerimientos a la vez para lograr una mayor focalización. Comenzar centrándose en lo que le da valor al cliente; en lo que es importante.

Lo más valioso es cuando el incremento está terminado y los usuarios reales pueden interactuar con el *software* y validar si lo implementado era lo que se necesitaba. Si no es así, es menos costoso echar para atrás lo realizado durante el ciclo, que en el caso en el que todo el sistema ya está construido. El esquema tiene un efecto de borde sobre la responsabilidad del cliente hacia el producto.

El diseño

Mito: "diseñar completamente antes de empezar a programar para que sea más efectiva la labor de programación y más independiente".

El mito es falso porque es imposible hacer un diseño completo y detallado para un conjunto de requerimientos incompletos y ambiguos. Ante los cambios, lo que suele suceder es que el diseño inicial se empieza a deteriorar hasta quedar completamente inservible. Cuando esto sucede es fácil caer en el ciclo programar-corregir. Las decisiones de diseño se toman directamente sobre el código sin tener una visión global de la aplicación ni una abstracción que permita comunicar, discutir y evaluar alternativas.

Diseñar es ante todo un proceso de tomar decisiones sobre cómo romper en partes (componentes, módulos, objetos, aspectos, servicios) y sobre cómo pegar lo que se separó. El proceso de cómo "pegar" puede ser tan simple como llamar un método sobre el mismo espacio de memoria, o tan complejo como utilizar un *middleware* de distribución.

Infortunadamente, la escogencia de una tecnología particular nos impone unas limitaciones sobre como descomponer. Esto dificulta el proceso de abstracción y por consiguiente, mantener las aplicaciones cuando es necesario cambiarlas y adaptarlas sin deteriorarlas.

La estrategia del proyecto debe incluir el diseño de la arquitectura global de la aplicación. Pero luego, a medida que se avanza, se debe diseñar solamente para los requerimientos que se seleccionaron. Lo ideal es que lo que se va a construir en un ciclo debe ser pequeño. Debe añadir cosas al diseño inicial o, eventualmente, puede

implicar rediseñar algunas cosas ya existentes.

Mantener el diseño simple o no sobre diseñar es una habilidad que debemos aprender. Para exagerar un poco con un ejemplo, supongamos que nos piden implementar un editor de cajas y terminamos construyendo un *framework* que genera editores de cajas, círculos y triángulos, todos ellos tridimensionales. El problema no es el *framework*, es que lo que hicimos costó mucho y tomó mucho más tiempo y a lo mejor no estaba tan claro si el editor de cajas era necesario.

Las metodologías ágiles plantean una práctica nueva para reconciliarnos con la tarea de diseño: la reestructuración disciplinada del código o *refactoring*. Tenemos un nuevo ciclo que no separa el diseño de la programación sino que se intercalan de la siguiente manera: (1) Diseñar – (2) Programar – (3) Probar – (4) Reconstruir el diseño – (5) Evaluar el nuevo diseño – (6) Re-Diseñar – (7) *Refactoring*.

La secuencia de actividades tiene varias implicaciones. Primero, aceptamos la posibilidad de que durante la tarea de programación el diseño original que queríamos implementar cambie. Eso no es un problema, si realizamos la tarea 4 y hacemos explícito, a un nivel más abstracto, qué fue lo que implementamos. Sobre esa abstracción podemos evaluar (tarea 5), eventualmente ajustar (tarea 6) y aquí viene la nueva práctica, reconstruir en forma disciplinada el código para que refleje nuestras mejoras (tarea 7 *refactoring*).

Este ciclo funciona y se puede llevar a cabo, si y solo si, percibimos el diseño como el conocimiento del proyecto y lo cuidamos como un activo valioso. Martin Fowler en su artículo "*Is Design Dead?* [5], utiliza la expresión "*The Will to Design*" y la define como la determinación de asegurarse de que el diseño mantendrá una altísima calidad. De lo contrario, caemos en el doloroso programar-corregir.

La planeación y el seguimiento

Mito: "para administrar en forma adecuada el proyecto, se debe tener un plan detallado de todo el proyecto desde el principio."

El mito es falso porque cuando no hay claridad en los requerimientos y no se puede tener completos los diseños, es una falacia pretender planificar en detalle el proyecto. Hacer una planeación con información incompleta y, en muchas ocasiones contradictoria y ambigua, es imposible. No se puede prescribir con tanta anticipa-

ción teniendo tantas incógnitas. La consecuencia mayor de planear en detalle desde el principio es que rápidamente el plan queda obsoleto y se guarda dentro del escritorio. Trabajar sin plan no solo deja las cosas al azar, también hace imposible saber cuál es el avance del proyecto.

El plan inicial debe contener los grandes hitos y la estrategia, pero por cada ciclo es necesario disponer de un plan con más detalle y luego por cada semana el detalle necesario como para que se pueda hacer seguimiento y ver el avance. Sobre los planes cortos no debería haber replaneación. Realizarla en forma permanente, conduce a que se pierda la oportunidad de analizar el estimado contra lo real y, lo más importante, la oportunidad de aprender.

La gente

Mito: "personas entrenadas y especializadas lograrán el éxito del proyecto".

Contrario a los demás, este mito es verdadero. El asunto aquí es que es difícil de alcanzar. Por un lado, se ha vuelto muy complicado encontrar gente calificada que pueda construir aplicaciones utilizando las últimas tecnologías. Y por otro, los desarrolladores que manejan las últimas tecnologías típicamente son los recién egresados que no tienen aún experiencia para enfrentar problemas.

Pero no estamos hablando solamente de tecnologías. También estamos hablando de entrenamiento en procesos. Los procesos y las personas son importantes. Ninguno es más importante que el otro. Se logra un efecto extraordinario cuando existe la combinación de buenos desarrolladores y un buen proceso como herramienta de trabajo. Un buen proceso es el que da valor a lo que se hace. No se limita a obligar a la gente a llenar formatos sin saber para qué. Un buen proceso ayuda a aumentar la motivación y a despertar el compromiso por la calidad. Procesos que ayudan a que los equipos de desarrollos sustenten su estrategia en la colaboración basada en la asignación de roles y guiada por la planeación y seguimiento permanentes.

Estos procesos se pueden construir si se aprovecha el esquema de ciclos incrementales para entrar en una espiral de mejoramiento continuo. Así como lo plantea Humphrey en TSP, cada ciclo inicia con una definición de objetivos que no se quedan en los requerimientos sino que incluyen los objetivos del equipo de trabajo y de los individuos, y se termina con una evaluación que nos permite aprender de lo que pasó y proponer cómo mejorar para el siguiente ciclo.

Algunos desafíos

Quizás uno de los desafíos más grandes para que esta forma de construir *software* se lleve a la práctica con éxito es la contratación y la relación con el cliente. Mientras los esquemas de contratación no sean más flexibles, el desarrollo por incrementos no podrá hacerse realidad. No podremos tener mejores proyectos hasta tanto no se planteen en relación gana-gana, de responsabilidad compartida y participación activa de los clientes, usuarios y desarrolladores.

Otro desafío está en entender los proyectos de construcción de *software* como proyectos de aprendizaje donde es fundamental la administración del conocimiento. Deberíamos poder explicar, ¿por qué se tomaron las decisiones?, ¿cómo se enfrentaron los problemas?, ¿cómo se manejaron los riegos?, ¿cómo se corrigieron los errores?, ¿cómo se interactuó con el cliente?, ¿cuánto invertimos?, ¿cuántos defectos se encontraron durante cada ciclo?

Tenemos también como desafío apoyarse en los ciclos de desarrollo para hacer la construcción de equipo de trabajo en una espiral de mejoramiento continuo. En los proyectos de conocimiento, la gente y los procesos son importantes. La gente se puede entrenar y los procesos se pueden construir y mejorar. Tenemos que propender por lograr esquemas de compromisos visibles, responsabilidades compartidas, roles establecidos y, ante todo, un alto compromiso con la calidad de lo que se hace.

Notas de pie de página

[1]*Rational Software Process [1]*

Referencias

[1] Ivar Jacobson, Grady Booch, James Rumbaugh, "El proceso unificado de desarrollo de software". Pearson Educación. 2000.

[2] Watts Humphrey. "Introduction to the Team Software Process". Addison Wesley.2000.

[3] Kent Beck, Cynthia Andres."Extreme Programming Explained: Embrace Change". Addison Wesley. 2004.

[4] Watts Humphrey. "Managing the Software Process". Addison Wesley.1989.

[5] Martin Fowler. "Is Design Dead? Disponible en www.martinfowler.com/articles/designDead.html. Última consulta: Agosto 2007.

Rubby Casallas. *Ingeniera de sistemas y computación, Universidad de los Andes. Especialista en Sistemas de información en la organización, Uniandes. Doctora en informática (Área ingeniería de software). Universidad de Grenoble, Francia. En la actualidad es profesora asociada del departamento de sistemas y computación de la Universidad de los Andes y coordinadora de la Especialización en Construcción de Software (ECoS). Interés principal en construcción de fábricas de software basadas en modelos.*