



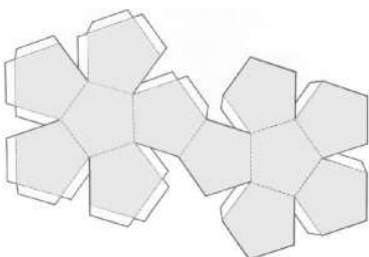
Code & fix

por Fco. Javier Mtz. de Ibarreta León

Dice Steve McConnell en su libro “Desarrollo y gestión de proyectos informáticos” [SMC96] que codificar y corregir (code-and-fix) es un modelo poco útil pero bastante común. Y así es, a punto de concluir el año 2003 sorprende ver la forma de trabajar en ciertas empresas dedicadas al desarrollo de software donde se consideran las fases de análisis y diseño como etapas prescindibles en un proyecto, algo que no es trabajo sino más bien una pérdida de tiempo y que por tanto ha de omitirse. Sí, a estas alturas aún hay quien piensa que la forma más rápida y eficaz de desarrollar aplicaciones informáticas consiste en situarse frente al teclado y comenzar a programar directamente, sea cual sea la complejidad del proyecto (consecuentemente estas personas, para quienes el desarrollo de software conlleva exclusivamente tareas de programación, sólo conocen la figura del programador). Al fin y al cabo algo como un gráfico con el modelo conceptual de una base de datos le dice poco o nada a un cliente; sí, unos dibujos muy estéticos de rectángulos unidos por líneas, pero lo que pide es una aplicación, un programa que funcione y resuelva determinadas necesidades de su empresa, de manera que es fácil caer en la tentación de pensar que todo trabajo que no sea teclear el código de ese programa es inútil.

Si en cambio dejamos a un lado la informática parece que las cosas se ven de distinta manera; al observar un edificio a muy poca gente se le pasará por la cabeza la idea de que no hayan existido unos planos para su construcción y que el resultado sea fruto de un proceso en que de principio a fin sólo se hayan utilizado ladrillos y cemento. Quizá el hecho de que la naturaleza del producto final es muy diferente, cuando se trata de edificios de cuando lo es de aplicaciones informáticas, haga creer que el proceso de desarrollo de algo tan etéreo como es el software no necesita de ninguna actividad análoga a aquellas efectuadas antes de comenzar con la construcción de un edificio. Pero no es necesario recurrir a ejemplos de tal complejidad; siempre se pueden encontrar situaciones en que se da por hecho la existencia de un trabajo previo y otras donde los productos finales resultan aparentemente tan sencillos que llega a sorprender donde comienza su proceso de elaboración real.

Recordemos por ejemplo las clases de trabajos manuales de cuando teníamos doce años; planteémonos el reto de tomar papel, tijeras y pegamento para construir un dodecaedro... no, no, sin regla, ni lápiz ni compás (“éso es para los niños”, “trazar unas líneas guía no es trabajar”...), comencemos directamente a cortar el papel... hummmm, ¿y si lo dejamos en recortar un simple pentágono?... ¡vaya!, sin la ayuda de unos trazos a base de regla, lápiz y compás no es tan sencillo, ¿verdad?, pero lo que se pretende construir parece sencillo de obtener a simple vista... Pues bien, en el desarrollo de software ocurre algo similar, pero la necesidad de esos trazos previos, que en otros contextos cualquiera puede entender, aquí no parece ser tan evidente para algunos.



Para McConnell [SMC96] code-and-fix es un modelo no formal que se utiliza normalmente porque es simple, pero no porque funcione bien. Puede ser efectivo para proyectos de pequeño tamaño (literalmente para “proyectos pequeños que se intentan liquidar poco después de ser construidos”, de hecho “resulta peligroso para otro tipo de proyectos que no sean pequeños”), pero nada más. Sin embargo el concepto de proyecto grande o proyecto pequeño se convierte en ocasiones en algo subjetivo; en palabras de Booch, Jacobson y Rumbaugh [BJR99], “un montón de empresas de desarrollo de software comienzan queriendo construir rascacielos, pero enfocan el problema como si estuvieran enfrentándose a la caseta de un perro”, y claro, considerando los proyectos como pequeños y con plazo de entrega para ayer, ¿no es mejor comenzar a codificar cuanto antes?. McConnell presenta este revelador ejemplo de que no es así:

Larry Constantine cuenta una historia sobre el concurso de Software de la Australian Computer Society (Constantine, 1995b). El concurso consistió en llamar a equipos formados por tres personas para desarrollar y entregar una aplicación de 200 puntos de función en 6 horas.

El equipo de Ernst y Young decidió seguir una metodología de desarrollo formal, una versión reducida de la metodología que acostumbraban a seguir; completa con actividades por etapas y entregas intermedias. Su propuesta incluyó un cuidadoso análisis y diseño, parte de lo que se describe en este capítulo como bases técnicas. Muchos de sus competidores se metieron de lleno en la codificación, y en las primeras horas, el equipo de Ernst y Young se quedó atrás.

Sin embargo, al mediodía el equipo de Ernst y Young era el equipo dominante. Al terminar el día, este equipo perdió, pero no fue debido a su metodología formal. Perdieron porque sobreescribieron accidentalmente alguno de los archivos, entregando menos funciones al final del día de las que habían demostrado que tenían a la hora del almuerzo.

Reaparecieron unos meses más tarde en otro concurso de desarrollo rápido (esta vez con control de versiones y haciendo copias de seguridad) y ganaron (Constantine, 1996).

La cuestión es bien simple: ¿pensar en qué es lo que hay que hacer y en cómo hacerlo antes de empezar?, ¿o bien empezar a hacer no se sabe muy bien qué ni de que manera?. Como dice el proverbio chino, no siempre el mejor camino es el más corto. □

Referencias bibliográficas.

- [SMC96] Desarrollo y gestión de proyectos informáticos. (lectura muy recomendable)
Steve McConnell
Ed.: McGraw-Hill
ISBN: 84-481-1229-6
- [BJR99] El Lenguaje Unificado de Modelado.
Grady Booch, James Rumbaugh, Ivar Jacobson
Ed.: Addison Wesley
ISBN: 84-7829-028-1