

Niveles de calidad: el agujero en las metodologías de software

Abraham Otero Quintana (abraham.otero@gmail.com)
Francisco Morero Peyrona (peyrona@gmail.com)

1. Introducción

Hace aproximadamente seis años uno de los autores de este documento comenzó de modo prácticamente simultáneo dos proyectos de software diferentes. Uno consistía en desarrollar una aplicación de escritorio orientada al procesamiento de señal cuya funcionalidad debía poderse extender mediante plugins. La herramienta debía presentar un API a los plugins, para facilitar su construcción. En este proyecto iban a participar varios desarrolladores. El segundo proyecto consistía en realizar un análisis de sensibilidad de un modelo de reconstrucción tridimensional de imágenes. Para ello era necesario desarrollar un software que invocase al modelo (un programa) con múltiples variaciones de sus parámetros sobre una amplia colección de bases de datos de imágenes, y realizar un análisis de estos resultados empleando varias herramientas (otros programas ya existentes).

Ambos proyectos se llevaron a cabo con éxito. El primero se desarrolló en Java y sigue siendo mantenido y extendido en la actualidad. Ha comenzado a usarse fuera de la organización donde se creó, y previsiblemente dentro de diez años todavía se seguirá usando. El segundo tomó la forma de un script de Python de aproximadamente trescientas líneas. Se ejecutó una vez con éxito (en ello tardó unos seis días en una máquina fuera de serie en aquellos momentos) y proporcionó los resultados que se esperaban de él. En estos momentos, el autor no conserva una copia de ese programa.

La pregunta que pretende contestar este documento es: ¿Tiene sentido emplear el mismo proceso de desarrollo de software en ambos proyectos? Obviamente, no. No es lo mismo un proyecto diseñado para ser extendido, que se prevé que se usará (y por tanto mantendrá) durante bastantes años y en el cual van a participar varios desarrolladores, que un "proyecto" que consiste en desarrollar un script que sólo se va a ejecutar una vez, va a ser desarrollado por una única persona, no va a ser extendido y no va a ser necesario mantener. Desde el punto de vista del retorno de inversión *¿qué sentido tiene invertir tiempo en hacer ese script más legible y mantenible si cuando se ejecute una vez con éxito ha terminado su vida útil?*

Ese script fue el primer programa en Python que yo (Abraham) y escribía. Tomé muchos atajos y el código era horrible. Cuando conseguí hacerlo funcionar, automáticamente entré en modo "refactoring", y comencé a usar mi editor de texto para dar mejores nombres a las variables e intentar estructurar el código en funciones. Al cabo de un rato una pregunta comenzó a atormentarme: ¿Para qué estaba haciendo todo eso? Mi instinto de desarrollador me decía que debía escribir software legible y fácil de mantener. Sin embargo, en realidad estaba perdiendo el tiempo. No tenía sentido invertir más esfuerzo en esa pieza de software. Lo único que tenía que hacer ahora era lanzarlo y esperar seis días para obtener el resultado.

1.1. El elefante blanco en la habitación de las metodologías de desarrollo software

No todo el software que desarrollamos requiere la misma calidad. No es lo mismo desarrollar un software que sólo se va a ejecutar una vez, que un software que probablemente nunca va a ser necesario modificar salvo para corregir bugs, que un software que esperamos que se emplee durante muchos años y se continúe extendiendo. No es lo mismo desarrollar un software que se va a emplear de modo aislado, que un software que se va a tener que integrar con otras aplicaciones, que un software que debe exponer una API sobre la cual se va a construir más software. La cantidad de esfuerzo y recursos que hay que emplear para desarrollar un framework que pensamos hacer público no es la misma cantidad de esfuerzos y recursos que se necesitan para desarrollar una aplicación web de Intranet. Incluso aunque el framework y la aplicación web tengan el mismo número de líneas de código. El primer desarrollo debe tener una calidad más alta, y por tanto requerirá más recursos y tiempo.

El contenido del párrafo anterior es una auténtica trivialidad. Cualquier desarrollador con experiencia sabe lo que se cuenta en él. Pero siendo esto así ¿Por qué ninguna metodología de desarrollo de software tiene esto en cuenta? Todas las metodologías de desarrollo de software, tanto las ágiles como las más tradicionales, tienden a definir un proceso que está orientado a conseguir un software de la máxima calidad posible. Pero no siempre se desea obtener la máxima calidad posible. Tiempo y recursos, siempre limitados, dictan que un framework público y una aplicación web simple de Intranet no requieren la misma calidad y, por tanto, no deberían usar el mismo proceso de desarrollo.

Nosotros argumentamos que desde el principio de un proceso de desarrollo de software, debería tratar de definirse de un modo explícito el nivel de calidad que tiene sentido que se alcance y que éste influya en el proceso de desarrollo. Queremos dejar perfectamente claro que con esto no estamos buscando excusas para desarrollar software de mala calidad. Quizás en un mundo ideal todo el software desarrollado tendría una calidad máxima (a menudo, éste parece ser el objetivo que tratan de alcanzar las metodologías de desarrollo de software). Pero en el mundo real esto no es así; y las metodologías de desarrollo de software deberían asumir esta realidad **definiendo distintos procesos para alcanzar distintos niveles de calidad.**

Podría parecer una reflexión elemental, pero desde que tuvimos esta percepción, comenzamos a abordar los proyectos de software de un modo diferente. Definir explícitamente desde el principio qué nivel de calidad se desea alcanzar, y en base a eso qué técnicas y herramientas se van a emplear para intentar llegar allí ha cambiado nuestra forma de afrontar proyectos software. En realidad, los desarrolladores, antes de emprender un proyecto de software realizan este análisis en su cabeza, y en base a eso toman una serie de decisiones que van a guiar el desarrollo. Sin embargo, **este proceso en la actualidad es bastante poco formal, implícito y no se suele documentar**, del mismo modo que hace veinte años los buenos desarrolladores hacían refactoring, pero no era un proceso formal. Nosotros defendemos que este proceso debería ser algo formal, debería realizarse de modo explícito, debería ser documentado e incluso quizás debería formar parte de un contrato empresarial para el desarrollo de una aplicación.

Formalizar estas tareas que todos realizamos de modo informal permitirá compartir de un modo más eficaz conocimiento sobre cómo tratar de alcanzar distintos niveles de calidad dentro de un proyecto. Del mismo modo que formalizar el concepto de refactoring permitió crear un vocabulario común para hablar sobre refactoring, lo cual facilitó el compartir conocimiento en torno a esta práctica y la creación de un conjunto de herramientas para apoyar en estas tareas.

1.2. Las buenas prácticas

La idea central de este documento es que la definición del nivel global de calidad a alcanzar en un proyecto de software debe ser uno de los pasos explícitos en la etapa de análisis del proyecto. Y que el nivel de calidad que se desea alcanzar debe influir en el proceso de desarrollo de software.

Éste es el mensaje más importante que queremos transmitir. A continuación reflexionaremos sobre los distintos factores que impactan en el nivel de calidad que requiere un proyecto. Después expondremos una serie de recomendaciones (que a nosotros nos sirven, pero que cada cual tendrá que adaptar a su realidad) para alcanzar distintos niveles de calidad correspondientes con distintas categorías de proyectos. Como casi cualquier buena práctica en el mundo del desarrollo de software, estas recomendaciones deben ser tomadas con prudencia y adaptadas al contexto en el cual se van a aplicar. Estas recomendaciones son sólo "buenas prácticas" que a nosotros nos han funcionado. Posiblemente otros en el futuro compartan sus propias buenas prácticas. Y en caso de producirse esto, estamos seguros que no estarán de acuerdo en parte o totalmente con las muestras. Nunca habrá un consenso en este sentido, del mismo modo que en la actualidad no hay un consenso sobre cuál es la metodología de desarrollo de software a usar. Creemos que los detalles no son tan importantes como el concepto general en sí.

2. Factores que impactan el nivel

Distinguiremos dos tipos de factores diferentes: intrínsecos y externos. Los primeros tienen que ver sólo con la naturaleza del propio proyecto, mientras los segundos están relacionados con el entorno particular donde se va a desarrollar.

2.1. Factores intrínsecos

2.1.1. ¿Cuánto código va a depender de nuestro código?

El primer factor intrínseco que influye en el nivel de calidad requerido es **¿Cuánto código va a depender de este proyecto?**. Como regla general, cuanto más código vaya a emplear (dependa de) nuestro proyecto, más calidad deberá tener este último.

Supongamos, por ejemplo, que la pieza de código para la que deseamos establecer un nivel de calidad es una librería que esperamos reutilizar internamente dentro de nuestra empresa en varios proyectos. Si la librería tiene fallos, impactará a todos los proyectos que la usan. Si alguno de los proyectos que la usa tiene fallos, no provenientes de la librería, sólo afectarán a ese proyecto y el impacto para la organización va a ser menor. La librería es un componente más crítico, por lo que tiene sentido hacer un esfuerzo superior para conseguir un nivel de calidad más elevado. Como se ha dicho, cuanto más código vaya a depender de nuestro proyecto, más crítico será nuestro proyecto y más calidad se le debe exigir.

Podemos crear una jerarquía de estereotipos de proyecto que van a ser usados por más código, y que por tanto requerirán un mayor nivel de calidad (de mayor a menor):

- 1 Librería/framework del núcleo de un lenguaje de programación
- 2 Librería/framework que se va a publicar a instituciones diferentes de la que la ha desarrollado
- 3 Librería/framework que se va a emplear internamente dentro de una institución

- 4 Proyecto a medida cuyo código no será reutilizado de ningún modo, sino que será ejecutado directamente a partir de interacciones en la interfaz de usuario

Son dos escenarios muy diferentes aquél en el que el código que se va a desarrollar sólo se ejecuta como resultado de interacciones con la interfaz de usuario, y aquél en el que se ejecuta como resultado de llamadas desde otro código. En el primer caso, el número de caminos de ejecución posibles en ese código va a ser mucho menor. Es un número acotado e incluso puede ser posible testarlo de un modo completamente manual, o mediante herramientas como FEST y Swinger, en algunos escenarios.

Por ejemplo, podemos tener una clase XMLParser a la cual hay que especificarle a través de métodos cuál es la ruta del archivo que va a cargar, después debemos especificar cuál es el encoding del archivo XML, y finalmente se invoca a un método que lo parsea. Dada la actual implementación de la clase XMLParser, si primero especificamos el encoding del archivo y después su ruta, el encoding especificado es ignorado al parsear y se emplea uno por defecto. Esto no ha sido documentado por el desarrollador de la clase XMLParser; es más, podemos suponer que el desarrollador no es consciente de este comportamiento. Sin embargo, en un proyecto a medida donde esta clase sólo se emplea para cargar un archivo de propiedades, y donde ambas acciones se realizan en el "orden correcto" (aquél en el que funciona adecuadamente) podría dar como resultado un producto perfectamente funcional con el cual nuestro cliente está completamente contento. Un tema diferente es que este comportamiento no documentado (o desconocido) puede ser una bomba de relojería en el futuro, cuando haya que hacer cambios en el proyecto. Pero si tuviésemos la certeza de que no va a ser necesario hacer cambios en el proyecto ¿merece la pena corregir/documentar el comportamiento de la clase XMLParser? Un purista diría que sí, que la solución actual no es la óptima. Un pragmático, como los autores, diría que la solución actual es aceptable y mejorar un código que ya hace lo que tiene que hacer y que nunca va a ser necesario modificar por razones de negocio, es perder el tiempo. Pero bueno, del "tiempo" ya hablaremos más adelante.

Supongamos ahora que la clase XMLParser forma parte de una librería que nuestra empresa vende a otras empresas, las cuales construyen sus productos sobre ella. Los programadores de estas empresas en sus programas podrían primero especificar el encoding, después la ruta del archivo y finalmente invocar al método que parsea. Sus programas no funcionarían adecuadamente y no tienen forma de saber por qué, a no ser que lean el código fuente de la librería. En este caso el comportamiento de la clase XMLParser no es admisible. Debería haber sido identificado mediante tests, y debería haber sido corregido o documentado.

Cuando una librería es llamada desde otro código, que puede estar escrito por un desarrollador diferente al que escribió la librería, puede emplear las clases y métodos de la librería de un modo diferente a lo que los autores de la librería habían imaginado. Cambiar el orden en el que se invocan los métodos es sólo un pequeño ejemplo. Cuando un código sólo se ejecuta como resultado de interacciones en la interfaz de usuario, podemos considerar que un código es "aceptable" si realizamos todas las interacciones posibles en la interfaz de usuario y el código realiza correctamente las acciones que se esperan de él (como sucedía con nuestra clase XMLParser en el proyecto a medida). Pero cuando el código es una librería que va a ser llamada por otro código, el conjunto potencial de caminos de ejecución del código de la librería no se ve limitado a lo que un interfaz de usuario permite, sino que es potencialmente infinito. Es necesario, por tanto, pensar más detalladamente en qué API está exponiendo nuestra librería, tratar de impedir que ese API se emplee de modo incorrecto usando los mecanismos que los lenguajes de programación proporcionan con este fin (por ejemplo, empleando los modificadores de visibilidad adecuados para métodos y clases, lanzando excepciones...) y, en aquellos casos en los cuales las limitaciones de nuestro lenguaje de programación no nos permitan impedir que el usuario la librería cometa errores, documentar cómo debe emplearse correctamente.

En general, desarrollar un proyecto a medida requiere implementar la funcionalidad especificada por los requerimientos del negocio para resolver el problema, nada más. Desarrollar una librería requiere hacer esto también, pero además, en este caso es necesario hacer el ejercicio mental de preguntarse

una y otra vez "cómo un desarrollador haciendo llamadas a mi API puede hacer que mi código cometa un fallo y cómo lo evito".

Cuanto más desarrolladores vayan a emplear la librería, "más cosas raras" se le pueden ocurrir a alguno de ellos, y más cuidado debemos de tener. Además, cuanto más código esté empleando nuestra librería, más complicado será realizar algún cambio que afecte al API público. Si en un proyecto a medida se cambia un método público de una clase pública, sólo afecta al proyecto. Si cambiamos una librería interna de la empresa, puede que tengamos que avisar a todos nuestros compañeros de la empresa para que modifiquen adecuadamente sus proyectos. No es ideal, pero seguramente es admisible. Si hacemos un cambio similar en una librería pública, tendremos que avisar a nuestros usuarios y éstos probablemente se levantarán en armas, nos criticarán en los foros de discusión de nuestra librería y en sus blogs, y habrán perdido un poco de confianza en nosotros. Si hacemos ese tipo de cosas a menudo, dejarán de usar nuestra librería. Y en el caso de un API como las API core de Java, probablemente nunca podamos cambiarlo. El costo para los usuarios sería demasiado elevado. Si metemos la pata (por ejemplo, `java.util.Date`) el error es "para siempre", al menos en lo que a los programas Java se refiere. Millones de desarrolladores van a sufrir el error, pero no resulta viable modificar el API y pedir a millones de desarrolladores en todo el mundo que cambien centenares, o miles de millones de líneas de código. Por ello tiene sentido dedicar más recursos a su desarrollo y hacer lo posible para "acertar" la primera vez.

2.1.2. Longevidad del código

¿Cuánto tiempo vamos a usar este código? En este caso, un ejemplo extremo sería el del script que queremos ejecutar una sola vez para conseguir un determinado objetivo, como el script en el cálculo en el problema de la reconstrucción tridimensional, o un script para mover un conjunto de archivos de un directorio a otro. Una vez ese código se haya ejecutado con éxito una sola vez, podemos tirarlo. Lo único que necesitamos conseguir al escribir el script es saber que "ahora y aquí" funciona adecuadamente.

Al extremo contrario se encontraría algo como una librería del core de Java. Como ya hemos argumentado, la longevidad en este caso es muy extensa, para lo que a los programas Java se refiere. Si cometemos un fallo, tendremos que cargar para siempre con él. Tenemos que asegurarnos de que la primera vez acertamos.

La longevidad afecta a la calidad del código a través de varios mecanismos diferentes. Uno es el **cambio de funcionalidad**, ya bien sea añadiendo, cambiando o eliminando. Cuanto más tiempo viva un código, más probable es que haya que hacer cambios en su funcionalidad. Si sabemos que un código va a cambiar debemos tratar de que ese código tenga un nivel de calidad que facilite esos cambios en el futuro. Cuantos más cambios preveamos que vaya a tener, más nos debemos esforzar en tener un código de mayor calidad, es decir, un código más fácil de modificar. Y, por lo general, el número de cambios incrementa proporcionalmente con la vida útil del proyecto.

Otro mecanismo a través del cual la longevidad afecta a la calidad del código son los **cambios en el equipo de desarrollo**. Si un proyecto va a ser desarrollado y mantenido a lo largo de toda su vida útil por un mismo equipo de desarrollo en el cual no prevemos que vaya a haber cambios (bien porque el proyecto va a tener una vida muy corta, o bien porque la composición del equipo de desarrollo es muy estable) podemos permitirnos licencias que no nos podemos permitir en el caso de que preveamos que a lo largo de la vida útil del proyecto vaya a haber cambios significativos en el equipo que lo mantiene. Si siempre va a ser el mismo equipo el que trabaja en el proyecto, hay algunas cosas que se pueden confiar a la memoria colectiva del equipo, como por ejemplo, por qué se han tomado ciertas decisiones de diseño, o cómo se deben de realizar las llamadas a un determinado API interno. Sin embargo si ese equipo va a cambiar, todo este tipo de conocimiento debe ser capturado de algún modo en la documentación. Y contar con tests que verifiquen si se están violando los supuestos realizados en el diseño del proyecto será muy útil para evitar que los nuevos desarrolladores cometan errores.

Un tercer mecanismo son **cambios en el entorno de ejecución** del código. Cuanto más tiempo viva un código, más probable es que ese código se ejecute en diferentes sistemas operativos o versiones del mismo, de la máquina virtual, del servidor de aplicaciones, con versiones diferentes de la base de datos, que se instalen algún tipo de parches en el sistema... Como todo desarrollador sabe, es mucho más fácil escribir un código que se ejecuta sin problemas "en una máquina" (especialmente si esa máquina es la del equipo de desarrollo) que un código que podamos garantizar que se ejecuta sin problemas en cualquier máquina.

Si debido a la longevidad del proyecto sabemos que todo el entorno de ejecución va a mantenerse constante, puede ser aceptable (nuevamente, hacemos énfasis en que no es lo ideal) permitirse licencias que no van a ser aceptables si sabemos que va a haber cambios en el entorno de ejecución. Si nuestra aplicación se va a ejecutar siempre en el servidor de la empresa, que corre bajo Linux, y la aplicación sólo va a utilizarse durante un mes por el departamento de marketing para una campaña: ¿Tiene sentido que en los test hagamos pruebas para verificar que la aplicación funciona correctamente si el sistema operativo usa rutas con barras "\"? Nosotros argumentamos que no. Basta con hacer tests en los que las rutas usen la barra "/" de Linux.

El número de entornos en los que se va a ejecutar un proyecto también dependen de cuánto código/desarrolladores van a emplear nuestro código. Si lo que estamos construyendo es un código que sólo tiene que ejecutarse en el servidor de nuestra empresa, tenemos un entorno mucho más controlado que si lo que estamos construyendo es una librería para la que no sabemos a priori en qué entorno exacto (versión del sistema operativo, versión de la base de datos, ...) se va a ejecutar. Esto enlaza con el criterio de "¿Cuánto código va usar este proyecto?" Como norma general, existe una relación directa entre la cantidad de código que va a usar nuestro proyecto y la heterogeneidad de entornos donde este se va a ejecutar. Todo ello redundará en la necesidad de conseguir un nivel de calidad más elevado.

2.2. Factores extrínsecos

Sería ideal si el tipo de proyecto fuese el único factor que influyese en el nivel de calidad que vamos a poner como objetivo. Pero el mundo no es ideal, tenemos que lidiar con limitaciones en el presupuesto, deadlines en las entregas impuestos por necesidades del negocio, inestabilidad en la definición de requisitos... Todos éstos son factores externos a la propia naturaleza del proyecto que influyen en el nivel de calidad que en la práctica podemos aspirar a conseguir.

A continuación, veremos algunos de los factores extrínsecos más reseñables.

2.1.1. El equipo de desarrollo

Aunque suene de perogrullo, debemos decir que cuanto mejor sea un equipo de desarrollo, mayor será la calidad que podamos intentar conseguir en el proyecto. En este sentido, algunas variables que van a influir en el nivel de calidad que se puede conseguir son: el nivel de conocimiento y de madurez del equipo; si se trata de un equipo bien cohesionado o de un equipo donde hay tensiones internas; y el propio interés de los miembros del equipo en el proyecto y en mejorar como desarrolladores.

Si contamos con un equipo maduro, con un alto nivel de conocimiento, bien cohesionado y con interés, el objetivo puede ser la luna. Si no estamos trabajando en estas condiciones ideales, habrá que hacer compromisos para adecuar los objetivos a la realidad de lo que el equipo puede conseguir.

Un entorno de desarrollo en el que se emplea Jenkins para la integración continua; Findbugs y Checkstyle para el análisis estático de código; Google Guice para inyectar dependencias, Mockito para crear mocks y poder realizar test; JUnit para los test de unidad y Selenium para los de integración; Git para el sistema de control de versiones... puede que a alguno de los lectores le suene al paraíso. Pero si el equipo de desarrollo está compuesto por personal junior, y ninguno de ellos ha

trabajado con ninguna de las herramientas mencionadas en el párrafo anterior, tratar de introducir todas esas tecnologías de golpe, además de realizar el proyecto que tenemos entre manos, no es una solución óptima: hay que priorizar. Quizás lo más importante para este proyecto sea tener test que nos ayuden a mantenerlo en el futuro. Nos olvidaremos por ahora de la integración continua y emplearemos CVS, que (supongamos) ya lo conocen todos los integrantes del equipo.

2.2.2. Estabilidad de los requisitos

Si los requisitos con los que trabajamos son estables, es más probable que el código que estamos desarrollando sea el que se va a poner en producción y el que se mantendrá en el futuro y por lo tanto, exigiremos un nivel de calidad más alto. Pero si los requisitos pueden cambiar en mayor o menor medida, puede que el código que desarrollemos se termine desechando.

Supongamos que 3.000 líneas de código que proporcionaban una determinada funcionalidad, finalmente se van a tirar al cubo de la basura porque el cliente una vez que ha visto los prototipos iniciales ha decidido que ya no quiere esa funcionalidad. Ha sido un desperdicio de tiempo, pero mucho más tiempo se habría desperdiciado si además de las 3.000 líneas de código hemos escrito 6.000 líneas adicionales de test de unidad, que también habrá que tirar a la basura.

Si los requerimientos no son estables, invertir recursos en obtener una elevada calidad en el código no tiene sentido. No al menos hasta que los requerimientos se hayan estabilizado un poco. Poniendo un ejemplo concreto, desde nuestro punto de vista, escribir tests automáticos exhaustivos en un proyecto no tiene sentido hasta que los requisitos están claros y son relativamente estables. Hay proyectos en los que esto sucede desde el principio. Hay otros proyectos en los que no.

Uno de los autores, Abraham Otero, investiga en ingeniería biomédica. En el campo de la investigación, cuando uno tiene una idea para resolver un problema no sabe si va a funcionar o no a priori. Muchas veces se prueban varias ideas hasta que se da con una que lo resuelve de un modo aceptable. Por eso se le llama "investigar". Tiempo atrás, yo (Abraham) quise probar el desarrollo TDD (Test Driven Development) en mi trabajo científico. En un par de semanas me quedó claro que no tenía sentido. Ya es bastante doloroso tirar al cubo de la basura líneas de código que has escrito durante varios días, o incluso semanas, porque la idea no funciona, para encima tener que tirar otro tanto código, o más, de tests. Con esto no digo que los tests no sean útiles, o que no deban usarse. Los tests automáticos son muy útiles para identificar regresiones en el código. Pero sólo hay que plantearse si merece la pena escribirlos cuando ya se ha implementado la idea, cuando se sabe que resuelve el problema y que ese código es que va a ir finalmente a producción. Desde mi punto de vista, TDD no encaja en la investigación (al menos en la que yo hago), ni en general en proyectos en los que los requisitos no están bien definidos desde el principio.

2.2.3. Deadlines

En ocasiones requerimientos del negocio obligan a tomar atajos en los proyectos de desarrollo de software. Y tomar atajos significa aceptar que vamos a crear un resultado final con una calidad menor de la que creemos que debería tener. Esto sucede en el mundo real, y no se puede hacer nada para evitarlo. Lo que se debe de hacer, como muchos autores ya han escrito antes de nosotros, es tratar de recuperar esa "deuda técnica" que hemos adquirido¹.

Tomar esos atajos significa que hemos "tomado prestado" tiempo del futuro, y en el futuro tendremos que "devolver" ese tiempo corrigiendo los atajos. Es decir, en el futuro deberemos trabajar para incrementar el nivel de calidad del proyecto hasta el nivel donde debería idealmente haber estado desde el principio, pero al que no pudimos llegar por las limitaciones temporales. No

¹ Acerca de este concepto, véase: http://es.wikipedia.org/wiki/Deuda_t%C3%A9cnica.

¹ Como curiosidad, los autores de este documento también manejamos el concepto contrario: "Crédito Tecnológico" que se refiere a ese trabajo extra y posiblemente innecesario que a veces se hace en previsión de que pueda resultar útil en el futuro.

profundizaremos más en este tema, ya que hay múltiples autores que han hablado largo y tendido sobre el problema de la deuda técnica.

2.2.4. Presupuesto

En contra de la creencia de algunos técnicos y de muchos gestores, crear software de calidad no tiene por qué ser más caro, de hecho, normalmente abarata el coste total de un proyecto a lo largo de la vida del mismo, ya que:

- Se minimiza la *Deuda Técnica*, y de la existente se conoce su dimensión y alcance, lo que es crucial (piénsese sino en el caso contrario)
- Los tiempos de implantación disminuyen
- Los costos de mantenimiento disminuyen
- El aplicativo es más extensible y por lo tanto se incrementa su longevidad
- La transferencia del conocimiento es mucho más fluida
- Se incrementa la longevidad del proyecto
- Se reduce el TCO²

Sin embargo, siempre, en todos los proyectos (al menos en todos los que yo, Francisco, me he encontrado), contamos con menos presupuesto del que desearíamos, es decir, con menos personal del que necesitamos y con menos tiempo del que hemos pedido. Es por ello que resulta crucial hacer un buen uso de los recursos de los que se dispone (no entraremos en cómo gestionar equipos porque queda fuera del objetivo de este documento).

Entendemos que la cantidad de recursos (tiempo y dinero) es inversamente proporcional al esfuerzo que hay que invertir en la *Planificación Inicial*, la *Ingeniería de Requisitos* y en la *Gestión del Cambio*:

- Hacer una buena *Gestión del Cambio* es siempre importante porque a poca envergadura que tenga un proyecto, siempre van a aparecer cambios. Esta es especialmente importante cuando el proyecto es externo.
- En un proyecto externo (cuando el cliente no es nuestra propia empresa) resulta crucial hacer una buena *Ingeniería de Requisitos*.
- Cuanto más ajustados estemos de recursos, mayor debe ser la planificación inicial.

Dicho en roman paladino: el cliente siempre va intentar que todos los cambios que se le ocurran (tanto mejoras como ampliaciones) le salgan gratis, y por ello lo achacará a que “eso” (sea lo que sea que pide) formaba parte de las especificaciones iniciales del proyecto. Por eso es crucial que éstas estén claras, es decir, que la *Ingeniería de Requisitos* se haya hecho de modo impecable; de otro modo vamos a tener discusiones muy serias y muy desagradables con el cliente. Supongamos que se ha hecho así, que le podemos demostrar al cliente que lo que nos pide no es parte del proyecto que él contrató; de todos modos, aún nos toca abordar otra difícil tarea: evaluar el impacto de lo que se nos pide y negociarlo tanto con el cliente como con el equipo, porque mientras que al primero el tiempo/coste le va a parecer una barbaridad, al segundo le va a parecer lo mismo, pero por el lado opuesto (a uno muchísimo y al otro poquísimo).

Es por ello que cuando los recursos escasean, la *Gestión del Cambio* y la *Ingeniería de Requisitos* deben ser de gran calidad.

² Acerca de este concepto, véase: http://es.wikipedia.org/wiki/Coste_total_de_propiedad

2.2.5. Cualquier otra cosa que influya en el nivel de calidad

Los factores externos que hemos recogido aquí no son una lista exhaustiva, sino una recopilación de los factores que más comúnmente influyen. Pero puede haber otros, por ejemplo, cuán crítico es el proyecto para el futuro de la empresa. Y otras muchas cosas que los autores de este documento no conocemos del entorno particular donde el lector trabaja. Por ejemplo, el hecho de que el desarrollador más senior de un equipo de tres personas esté pasando por un duro divorcio y que otro compañero acabe de romperse una pierna y vaya a estar 2 meses de baja, influyen a la hora de sacar adelante el próximo proyecto. Además, es muy posible que el manager se niegue a cubrir la baja, ya que a veces no tendría mucho sentido: ese equipo es el que siempre se encarga de este tipo de proyectos porque es el único que tiene experiencia haciéndolos. Incorporar alguien nuevo no va a ayudar a dos meses vista; todo lo contrario, lo retrasaría. Esa pierna rota y ese divorcio probablemente sean factores a considerar a la hora de determinar la calidad que se puede alcanzar en el proyecto.

Cada equipo debe hacer el ejercicio de analizar las características de su entorno particular y tener en cuenta todos los factores a hora de determinar el nivel de calidad que pueden aspirar a alcanzar en cada proyecto.

3. Reglas del Pulgar

En inglés, se conoce como **Regla del Pulgar** (*Rule of Thumb*) a esas reglas “que designan un principio o criterio de amplia aplicación que no necesariamente es estrictamente preciso ni fiable en cada situación. Establece una especie de fórmula u observación generalmente aceptada como conocimiento práctico basado en la experiencia, sin embargo no se trata de una proposición científica. Es un procedimiento de fácil aprendizaje, destinado a recordar o calcular aproximadamente un valor o tomar una decisión³”.

A continuación vamos a dar algunas reglas del pulgar sobre los niveles de calidad que deben tener distintos procesos involucrados en la producción de software para distintos tipos de proyectos. Es cierto que no están todos los procesos, también es cierto que otros han sido agrupados. Y probablemente la clasificación que se ha hecho de los proyectos no sea la más adecuada, pero al fin y al cabo, esto son reglas del pulgar.

No hemos distinguido entre desarrollos internos (realizados para la empresa para la que se trabaja) y externos (realizados para clientes), porque entendemos que los primeros son demasiado particulares de cada caso, por ello, nos limitamos a los segundos. Aunque sí que podemos dar una regla del pulgar también para los primeros: en general los desarrollos internos requieren un nivel de calidad que varía entre una y dos estrellas menos para todos los procesos.

Finalmente deseamos aclarar que lo que a continuación se expone no es más que el fruto de la experiencia de los autores, por lo que, con toda seguridad, el lector disenterá en algunas (o muchas) de estas recomendaciones.

3.1. Framework o librería

Entendemos aquí por framework o librería cualquier pieza de software que va a ser utilizada por otros desarrolladores para a su vez construir nuevas piezas de software o productos finales. Se trata por tanto de código que va a ser usado por un número posiblemente alto de desarrolladores para construir aplicaciones, las cuales seguramente se ejecutarán en un conjunto heterogéneo y diverso de

³ Tomado de: http://es.wikipedia.org/wiki/Rule_of_thumb

plataformas. Además, el proyecto en cuestión seguramente tendrá que ser mantenido durante un periodo largo de tiempo.

Proceso	Nivel calidad	Reglas del pulgar
Tomar requisitos	★★★★☆☆	<ul style="list-style-type: none"> Definir claramente qué ámbito que el framework pretende abarcar. Definir aún más claramente lo que el framework <u>no</u> va a abarcar. Al contrario que el “ámbito”, la funcionalidad debe definirse de forma más bien genérica; ya que ésta se irá concretando durante la implementación del producto.
Arquitectura y Diseño	★★★★★★	<ul style="list-style-type: none"> Escoger personal con experiencia previa en la resolución de problemas para el área del framework. Deben construirse aplicaciones de ejemplo probando el framework y las APIs antes de dar por válida una determinada decisión de diseño.
Doc. externa	★★★★★★	<ul style="list-style-type: none"> Debe existir un manual de usuario/tutorial del framework. Deben estar documentadas todas las partes públicamente accesibles del código (clases, métodos, interfaces...) empleando, en el caso de Java, javadoc, o en otros lenguajes una herramienta similar. Deben proporcionarse buenas prácticas a la hora de emplear el framework en desarrollos. Es deseable contar con código de ejemplo para mostrar cómo llevar a cabo tareas comunes empleando el framework.
Doc. interna	★★★★★★	<ul style="list-style-type: none"> Las decisiones de diseño internas deben estar bien documentadas. El código debe estar bien documentado.
Código	★★★★★★	<ul style="list-style-type: none"> Deben definirse y estandarizarse en toda la base de código convenios de codificación. Es deseable emplear una herramienta automática como Checkstyle para identificar violaciones en estos convenios. Es imprescindible emplear un sistema de control de versiones y definir una política clara de con qué frecuencia deben realizarse commits, si es o no aceptable hacer un commit de código que no pasa todos los test, comentarios a incluir al hacer el commit, etc. El versionado será doble: evolutivo y correctivo. El primero supone saltos mayores en el número de versión y el segundo saltos menores. Ambos flancos deben llevarse simultáneamente. Antes de realizar cambios profundos (en la arquitectura de la aplicación, actualización

		tecnológica, refactorización profunda, etc), se creará un fork para no afectar a los desarrollos actuales. Haciendo un join si se demuestra la idoneidad del cambio.
Testing	★★★★★	<ul style="list-style-type: none"> ● Emplear herramientas de análisis estático de código como FindBugs. ● Desarrollar test automáticos de unidad alcanzando una alta cobertura del proyecto (>90%). ● Es deseable contar con test de integración, empleando una herramienta como, por ejemplo, Selenium. ● Especialmente si no se cuenta con test de integración, debe desarrollarse y mantenerse a lo largo de la vida del proyecto una aplicación que ejercite toda la funcionalidad del framework; el funcionamiento correcto de la aplicación da cierta garantía de que no se introducen regresiones. ● Los test deben ejecutarse en distintas plataformas (probar varios sistemas operativos, varias bases de datos, varios servidores de aplicaciones...)

A menudo, desarrollar un framework para afrontar una casuística, requiere de amplia experiencia resolviendo el tipo de problemas a afrontar, antes de ser capaz de abstraer a partir de esta experiencia un diseño adecuado para el framework.

La toma de requisitos en estos proyectos no suele ser crucial, porque normalmente, podremos en futuras versiones añadir nuevas funcionalidades. No obstante, debemos ser cuidadosos con que los nuevos requisitos que se implementen en el futuro no rompan la compatibilidad hacia atrás.

Los errores en el análisis y diseño serán acarreados por el proyecto durante toda su vida y además dificultarán la inclusión de mejoras y nueva funcionalidad en versiones posteriores, de ahí que este proceso sea crucial.

Respecto a documentación externa del código: puesto que el proyecto va a ser utilizado por otros desarrolladores, tiene que estar muy bien documentado y profuso en ejemplos, para facilitar su utilización. Dado que el código va a mantenerse durante mucho tiempo, es importante que sea legible y esté bien estructurado, y su documentación interna sea buena. La extensa longevidad del proyecto hace más probable que la composición del equipo de desarrollo cambie a lo largo del tiempo, lo que incrementa las necesidades de una buena documentación, claridad en el código, y disponer de una buena batería de test.

3.2. Aplicación crítica

Algunas de las aplicaciones típicas que pueden caer en esta categoría son: los servidores de aplicaciones, servidores web, control de dispositivos médicos e industriales, control de maquinaria, aplicaciones que manejan datos sanitarios, aplicaciones de banca, etc.

Proceso	Nivel calidad	Reglas del pulgar
Tomar requisitos	★★★★★	<ul style="list-style-type: none"> • Se hará de forma exhaustiva y utilizando alguna de las metodologías existentes especializadas en el área • No sólo se recogerán los requisitos funcionales sino también (si procede) los de niveles de servicio: tiempos máximo de respuesta para cada proceso, número mínimo de transacciones por unidad de tiempo, etc. • Es útil hacer fichas para recoger los niveles de servicio, en ellas, además se los niveles deseados, se añadirán los niveles obtenidos tras las pruebas de rendimiento.
Arquitectura y Diseño	★★★★☆	<ul style="list-style-type: none"> • Cumplir los requisitos está por encima de la calidad del diseño. Éste supedita a los requisitos y se podrán infringir algunas buenas prácticas si con ello, por ejemplo, se garantiza que la aplicación procesa la información en los tiempos requeridos.
Doc. externa	★★★★★	<ul style="list-style-type: none"> • La documentación externa suele ser poco más que un manual de instalación y/u operaciones; pero tiene que ser clara y completa, para que cualquiera ajeno al desarrollo pueda instalarla (si procede) y operar con la aplicación.
Doc. interna	★★★★★	<ul style="list-style-type: none"> • Las decisiones de diseño internas deben estar bien documentadas. • El código debe estar bien documentado. • Puesto que el diseño y el código a veces no siguen las buenas prácticas, es imprescindible que la documentación interna contrarreste estas deficiencias siendo impecable y explicando los compromisos que se han hecho para alcanzar los niveles de servicio.
Código	★★★★☆	<ul style="list-style-type: none"> • Deben definirse y estandarizarse en toda la base de código convenios de codificación. • Es deseable emplear una herramienta automática como Checkstyle para identificar violaciones en estos convenios. • Es imprescindible emplear un sistema de control de versiones y definir una política clara de con qué frecuencia deben realizarse commits, si es o no aceptable hacer un commit de código que no pasa todos los test, comentarios a incluir al hacer el commit, etc. • El código debe ser todo lo claro y legible que se pueda, pero en estas aplicaciones a menudo es necesario primar la eficiencia a la claridad. • Antes de realizar cambios profundos (en la arquitectura de la aplicación, actualización tecnológica, refactorización profunda, etc), se

		creará un fork para no afectar a los desarrollos actuales. Haciendo un join si se demuestra la idoneidad del cambio.
Testing	★★★★★	<ul style="list-style-type: none"> • Emplear herramientas de análisis estático de código como FindBugs. • Desarrollar test automáticos de unidad alcanzando una alta cobertura del proyecto (>90%). • Es deseable contar con test de integración, empleando una herramienta como por ejemplo Selenium. • Se harán test no sólo de funcionalidad, sino también de rendimiento para verificar los requerimientos relacionados con nivel de servicio (por ejemplo, test de stress). • Los distintos test deben ejecutarse en distintos entornos de ejecución (siempre y cuando la aplicación vaya a ejecutarse en distintos entornos).

Este tipo de aplicaciones, por manejar información muy sensible o por su carácter de infraestructura tecnológica, son muy delicadas. Se les exige un gran nivel de calidad en todos sus aspectos. No deben fallar, pero sobre todo, no pueden procesar erróneamente los datos que manejan. En lo único en lo que se relajan los criterios de calidad suele ser en el interfaz de usuario, que normalmente no es muy llamativo ni sigue el “state of the art” del momento.

Cumplir los requisitos (tiempos de respuesta, número de transacciones por unidad de tiempo, etc) es imperativo. Por tanto, seguir las buenas prácticas no es tan importante como alcanzar la funcionalidad esperada. Cualquier truco, atajo y mala práctica es aceptable si con ello se consiguen los tiempos requeridos. Esto no quiere decir que no se deba imprimir la máxima calidad posible, simplemente quiere decir que si siguiendo las buenas prácticas no se consiguen los resultados deseados, habrá que saltárselas, tratando de mitigar esto a través de una buena documentación.

3.3. Aplicación de gestión estándar

Aplicaciones típicas que consideramos que caen en esta categoría son: aplicaciones orientadas al manejo de información, B2B (business-to-business), B2C (business-to-customer), etc.







Proceso	Nivel calidad	Reglas del pulgar
Tomar requisitos	★★★★★	<ul style="list-style-type: none"> • Por mucho que nos esforcemos en hacerlo bien desde el comienzo en estas aplicaciones siempre se incorpora nueva funcionalidad durante el proceso de creación de las mismas. Esto no quiere decir que no se deba abordar este proceso con un alto nivel de calidad, pero debemos estar preparados para recibir nuevos requisitos.
Arquitectura y Diseño	★★★★★	<ul style="list-style-type: none"> • Estas aplicaciones suelen ser bastante grandes y por lo tanto hacer un buen análisis y diseño de la

		<p>aplicación va a permitir que escribir el código de la misma resulte más sencillo y más rápido.</p> <ul style="list-style-type: none"> • Si bien la arquitectura no suele sufrir muchos cambios, el diseño sí los sufre, por lo que es conveniente disponer de mecanismos (y herramientas) que faciliten la gestión de cambios en este área.
Doc. externa	★★★★★	<ul style="list-style-type: none"> • La documentación externa suele ser poco más que un manual de despliegue y uno de operaciones, cuantos más claros y concisos mejor.
Doc. interna	★★★★☆	<ul style="list-style-type: none"> • El principal propósito de esta documentación en este tipo de aplicaciones es el facilitar el mantenimiento evolutivo. • Las decisiones de diseño internas deben estar bien documentadas. • El código debe estar razonablemente documentado.
Código	★★★★★	<ul style="list-style-type: none"> • Deben definirse y estandarizarse en toda la base de código convenios de codificación. • Es deseable emplear una herramienta automática como Checkstyle para identificar violaciones en estos convenios. • Es deseable emplear un sistema de control de versiones.
Testing	★★★★★	<ul style="list-style-type: none"> • Es deseable emplear herramientas de test automático como FindBugs. • Es deseable contar con test automáticos de unidad alcanzando una cobertura razonable del proyecto (>70%). • Es deseable contar con test de integración, empleando una herramienta como por ejemplo Selenium. • Es deseable ejecutar los test en distintos entornos de ejecución (siempre y cuando la aplicación vaya a ejecutarse en distintos entornos).

El nivel de calidad requerido por este tipo de aplicaciones estará en relación directa a la longevidad prevista para la aplicación y al número de entornos de ejecución diferentes en los que tendrá que correr. Una aplicación de gestión que se va a vender como producto a múltiples empresas y sobre la cual esperamos hacer negocio durante muchos años deberá al menos cumplir los estándares de calidad recogidos en la tabla anterior. Una aplicación de gestión que no prevemos que vaya a tener una vida larga y que sólo se va a realizar una instalación de ella, podría rebajar entre una y dos estrellas en el nivel de calidad de cada uno de los procesos recogidos en la tabla anterior.

3.4. Aplicación informativa

Aplicaciones típicas que consideramos caen en esta categoría son web corporativa, tablón de anuncios, magazine, wikis, etc.

Proceso	Nivel calidad	Reglas del pulgar
Tomar requisitos		<ul style="list-style-type: none"> En este tipo de aplicaciones el cliente tiende a adaptar sus requerimientos a la funcionalidad proporcionada por la herramienta (habitualmente un CMS o similar) que se ha decidido utilizar. Esto suele simplificar la toma de requisitos.
Arquitectura y Diseño		<ul style="list-style-type: none"> Debido a que habitualmente estas aplicaciones se apoyan fuertemente en alguna herramienta (CMS o similar), no suele ser necesario escribir mucho código, por lo que las labores de análisis y diseño son mínimas o inexistentes.
Doc. externa		<ul style="list-style-type: none"> La mayor parte de la documentación externa suele ser proporcionada por la herramienta de base (CMS o similar) empleada.
Doc. interna		<ul style="list-style-type: none"> El principal propósito de esta documentación debe ser facilitar el mantenimiento evolutivo. El código debe estar razonablemente bien documentado.
Código		<ul style="list-style-type: none"> Es importante seguir las buenas prácticas que marca el fabricante del producto (CMS o similar) utilizado. Es deseable definir y estandarizar en toda la base de código convenios de codificación. Es deseable emplear un sistema de control de versiones.
Testing		<ul style="list-style-type: none"> Es deseable emplear herramientas de test automático como FindBugs. Es deseable contar con test automáticos de unidad alcanzando una cobertura razonable del proyecto (>70%). Aunque no óptimo, este tipo de aplicaciones a menudo pueden testarse razonablemente de un modo manual interaccionando con su interfaz de usuario.

Normalmente en estas aplicaciones se codifica bastante poco comparativamente con el resto de aplicaciones. Buena parte de las labores a realizar están relacionadas con la configuración del sistema y con el trasvase de información desde y hacia otros sistemas. Por otro lado, la funcionalidad de estas aplicaciones no suele presentar una gran variabilidad, lo que ha fomentado que haya un buen número de productos (tipo CMS y similares) que facilitan enormemente su creación y mantenimiento evolutivo, cubriendo “out-of-the-box” (en la mayor parte de los casos) el 90% de las necesidades del cliente.

3.5. Aplicación de usar y tirar

Aplicaciones típicas que consideramos caen en esta categoría son pruebas de concepto, aplicaciones para la obtención de datos o para el análisis de datos cuando estos procesos no van a repetirse en el futuro, etc.

Proceso	Nivel calidad	Reglas del pulgar
Tomar requisitos	★☆☆☆☆	<ul style="list-style-type: none"> Los requisitos suelen ser bastante sencillos, tanto que menudo no merece la pena definirlos de un modo formal; con que los tengamos en la mente es suficiente.
Arquitectura y Diseño	Prueb. concepto ★★★★★ Otros casos ★☆☆☆☆	<ul style="list-style-type: none"> Si vamos a hacer pruebas de concepto, lo mejor es no tener ideas prefijadas, cuanto más libres seamos, mejores pruebas haremos. Si se trata de cualquier otra cosa, entonces es recomendable dedicar un par de horas a enfocar cómo vamos a estructurar el código, qué librerías de terceros necesitaremos, etc. Pero lo haremos de un modo informal, sin utilizar herramientas de análisis ni de diseño: bastará con tomar algunas notas y algunas búsquedas por Internet.
Doc. externa	★☆☆☆☆	<ul style="list-style-type: none"> Sólo en el caso en el que preveamos que en el futuro se necesitará volver a ejecutar este código, generar la documentación mínima para permitir su ejecución.
Doc. interna	★★★★★	<ul style="list-style-type: none"> No ha lugar.
Código	★☆☆☆☆	<ul style="list-style-type: none"> Con que seamos capaces de entender lo que hemos escrito durante el proceso de desarrollo de la aplicación es suficiente.
Testing	★★★★★	<ul style="list-style-type: none"> No ha lugar.

En ocasiones, sobre todo si el propósito de la aplicación no es ser una prueba de concepto, sino importar, exportar, extraer o analizar datos, para estos desarrollos "todo vale" con tal de que se consiga ejecutar una vez de modo correcto el programa; porque entonces habrá terminado su vida útil.

4. Conclusiones

Los autores asumimos que probablemente la mayor parte de los lectores de este documento no estarán de acuerdo con muchos de los detalles aquí presentados, en especial con las reglas del pulular de la sección 3. Pero esperamos que al menos este documento pueda servir para hacer explícito en la mente del lector algo que, con toda seguridad, ya estaba en ella, al menos de modo implícito: las restricciones del entorno, los recursos limitados y la propia naturaleza del proyecto a desarrollar influyen en el nivel de calidad que es posible, y que tiene sentido, pretender alcanzar. Y que para alcanzar niveles de calidad diferentes, tiene sentido emplear procesos de desarrollo diferentes. Si hemos conseguido esto, nos damos por satisfechos.