Colexio Profesional de Enxeñaría en Informática de Galicia

CPEIG

XUNTA DE GALICIA

I Jornada sobre Calidad del Producto Software e ISO 25000, Santiago de Compostela, 10 de junio de 2014

Colexio Profesional de Enxeñaría en Informática de Galicia

Coordinadores: Fernando Suárez Lorenzo

Javier Garzás Parra

I Jornadas sobre Calidad del Producto Software e ISO 25000

Santiago de Compostela, 10 de junio de 2014

Edita: 233 Grados de TI S.L
www.233gradosdeti.com

Coordinan: Fernando Suárez Lorenzo
Javier Garzás Parra

Colaboran:

XUNTA DE GALICIA

Diseño y Maquetación:
233 Grados de TI , S.L

Santiago de Compostela, 2014.
ISBN: 978-84-617-0409-5

Índice de contenidos

ÍNDICE DE CONTENIDOS **4**

PRÓLOGO **8**

PRESENTACIÓN **10**

SOBRE LOS AUTORES **12**

INTRODUCCIÓN **20**

1. CALIDAD DEL PRODUCTO SOFTWARE, TU EMPRESA TENDRÁ UN PROBLEMA SI NO SE PREOCUPA POR ELLA. **23**

1.1 LA CALIDAD DEL PROCESO 23
1.2 LA CALIDAD DEL PRODUCTO 24
1.3 LA CALIDAD DEL EQUIPO Y/O PERSONAS 24
1.4 ¿QUÉ PERSPECTIVA DE CALIDAD ES MÁS IMPORTANTE? 26
1.5 RAZONES POR LAS QUE LA CALIDAD DEL PRODUCTO SOFTWARE DEBIERA PREOCUPARTE MUCHO 27
1.5.1 Calidad del producto no es lo mismo que testing 27
1.5.2 La calidad del proceso no garantiza la calidad del producto 28
1.5.3 CMMI no asegura un producto de calidad 30
1.5.4 La mala calidad del producto siempre tiene un coste 30
1.5.5 El cliente puede detectar la mala calidad del producto software 36
1.5.6 Las buenas prácticas no aseguran calidad del producto 36
1.6 REFERENCIAS 37

2. CERTIFICACIÓN CALIDAD DEL PRODUCTO SOFTWARE CON ISO 25000. ..... **44**

2.1 SITUACIÓN ACTUAL DE LA CALIDAD SOFTWARE 44
2.2 LA NORMA ISO/IEC 25000 46

2.3 ESTRUCTURA DE LA FAMILIA ISO/IEC 25000 46
2.4 ECOSISTEMA DE EVALUACIÓN Y CERTIFICACIÓN DEL PRODUCTO SOFTWARE 48
2.5 CERTIFICACIÓN DEL PRODUCTO SOFTWARE 50
2.6 AQC LAB: LABORATORIO ACREDITADO PARA LA EVALUACIÓN DEL PRODUCTO SOFTWARE 51
2.6.1 Modelo de calidad del producto 52
2.6.2 Entorno de evaluación del producto 54
2.7 PROYECTO PILOTO DE CERTIFICACIÓN DEL PRODUCTO SOFTWARE 56
2.8 REFERENCIAS 58

3. EL PAPEL HUMANO EN LA CALIDAD DEL SOFTWARE. INTRODUCCIÓN 60

3.1 LAS PERSONAS O LOS PROFESIONALES 64
3.2 EL RESPONSABLE DE CALIDAD 67
3.3 QUALITY COACH. UN NUEVO ORDEN 71
3.4 CONCLUSIONES 76
3.5 REFERENCIAS 79

4. BDD: UNIENDO NEGOCIO Y PRUEBAS TÉCNICAS PARA MEJORAR LA CALIDAD DEL SOFTWARE. 80

4.1 EL GRAN PROBLEMA DEL SOFTWARE 80
4.2 LENGUAJE UBICUO 83
4.3 BDD 85
4.4 GHERKIN 86
4.5 CUCUMBER 89
4.5.1 ¿Cómo funciona? 89
4.6 ¿CÓMO IMPLEMENTAR UN PROCESO BDD? 91
4.6.1 Definir los escenarios a desarrollar 91
4.6.2 Crear tests automáticos que validen la funcionalidad 91
4.6.3 Crear un entorno de trabajo 92
4.6.4 Evolución y adaptación 92
4.7 CONCLUSIONES 93

4.8 REFERENCIAS 94

5. LA CALIDAD DEL PRODUCTO DESDE LAS PRUEBAS Y EN RENDIMIENTO..... 96

5.1 INTRODUCCIÓN 96
5.2 ELEMENTOS BÁSICOS 97
5.3 LAS PRUEBAS SON SIMULACIÓN 97
5.4 NECESITAMOS HERRAMIENTAS 98
5.5 DEFINIMOS OBJETIVOS 99
5.6 CONSTRUIMOS EL ENTORNO 100
5.7 EXPLOTANDO LAS PRUEBAS DE RENDIMIENTO 101
5.8 PRUEBAS DE COMPONENTES 102
5.9 PRUEBAS DE PROFILING 103
5.10 PRUEBAS DE DIMENSIONAMIENTO 104
5.11 PRUEBAS DE RESILIENCIA 105
5.12 PRUEBAS DE EVOLUCIÓN 106

6. MIDIENDO LA CALIDAD DE CÓDIGO EN WTF/MINUTO. EJEMPLOS DE CÓDIGO REAL QUE PODRÍAN OCASIONARTE UN DERRAME CEREBRAL. 108

6.1 LA ANTOLOGÍA DEL DISPARATE 108
6.2 MEDIDAS DE CALIDAD FORMALES E INFORMALES. 109
6.3 LA ÚNICA MEDIDA VÁLIDA DE CALIDAD DE CÓDIGO: WTF/MIN 110
6.4 EJEMPLOS Y CONTRAEJEMPLOS PARA EVITAR WTFS. 112
6.4.1 Comentarios. 113
6.4.2 Excepciones. 115
6.4.3 Nombrado 116
6.4.4 Código innecesario. 117
6.5 MEJORA DEL PROCESO DE DESARROLLO Y ASEGURAMIENTO DE LA CALIDAD DE CÓDIGO. 118
6.5.1 Cuando se mira la calidad sólo al final. 118
6.5.2 Integración progresiva de la calidad en el proceso 119

7. LA EXPERIENCIA DEL CORTE INGLÉS EN LA GESTIÓN DE LA CALIDAD SOFTWARE. 124

7.1 CALIDAD DEL SOFTWARE 125
7.1.1 Dimensiones de la Calidad del Software 125
7.1.2 Normativas en el ámbito Empresarial (TI) 127
7.1.3 ¿Qué es un Producto? 127
7.1.4 Calidad 128
7.2 REFERENCIA EN LAS ORGANIZACIONES PARA EL CONTROL DE LA CALIDAD 130
7.3 ÁMBITO EMPRESARIAL 131
7.3.1 Plan de Aseguramiento de la Calidad 131
7.3.2 Modelos de desarrollo orientados a Producto: Calidad Concertada 132
7.3.3 catalogación de productos 132
7.3.4 Plan de Certificación 134
7.3.5 Grupo de Quality assurance (QA) 136
7.4 AUTOMATIZACIÓN DEL CICLO DE VIDA DE DESARROLLO 137
7.4.1 Integración Continua 138
7.4.2 Entrega Continua 143
7.5 TECHNICAL QUALITY MANAGEMENT PRO-ACTIVO Y CONTINUO 144
7.6 REFERENCIAS 146

8. ¿QUÉ CALIDAD TIENE REALMENTE EL SOFTWARE? LA EXPERIENCIA DE EVALUAR LA CALIDAD DE 1000 PROYECTOS BAJO LA ISO 25000. 147

8.1 ¿POR QUÉ ANALIZAR LA CALIDAD DE 1000 PROYECTOS SOFTWARE? 147
8.2 EL CASO DE ESTUDIO 150
8.3 ¿CÓMO SE HAN ELEGIDO LOS PROYECTOS ANALIZADOS? 154
8.4 ANALIZANDO LOS RESULTADOS 155
8.4.1 Carencias de calidad más repetidas entre los proyectos 156
8.4.2 Relaciones entre métricas 159
8.4.3 Relación entre la complejidad ciclomática y el código repetido. 162
8.5 REFERENCIAS 163

Prólogo

La calidad del software es un aspecto fundamental a tener en cuenta por las organizaciones que desarrollan y compran aplicaciones informáticas. El software debe funcionar como se espera, con un buen rendimiento, con un nivel adecuado de usabilidad, fiabilidad y seguridad. Además, se debe poder evolucionar adecuadamente con el paso del tempo, envejeciendo bien según se le van haciendo las inevitables modificaciones para irse adaptando a nuevas realidades técnicas y de negocio. Por último debe ser portable y compatible con otros sistemas.

Son estas las características que se esperan de un software de calidade según el modelo definido en la serie de normas ISO 25000. Esta norma es la espina dorsal de la temática a tratar en este libro e indica qué aspectos se deben tener en cuenta para alcanzar un buen nivel en cada una de esas características.

Las empresas y las administraciones públicas, así como casi cualquier organización a día de hoy, son grandes consumidores de software, tanto del hecho a medida como de produtos comerciales. Por lo tanto, la calidad del software debe ser un aspecto relevante para todas ellas, incluyendo por supuesto a la Xunta de Galicia, entre otros motivos por el importante impacto que puede llegar a tener la utilización de software de baja calidad, en muchos casos con repercusiones económicas relevantes.

El Colexio Profesional de Enxeñaría en Informática de Galicia, en colaboración con la Amtega, organismo que gestiona la política tecnológica

de la Xunta de Galicia, abordará este tema en el "Foro para a calidade do software", que tendrá lugar en junio de 2014 en la Cidade da Cultura de Galicia. Este libro resume el contenido de los relatorios que se llevarán a cabo en este foro, por lo que será de gran interés para todos los agentes involucrados e interesados en el desarrollo de software.

Mar Pereira Álvarez,

Directora da Axencia para a Modernización Tecnolóxica de Galicia

Presentación

La calidad del software es una preocupación a la que se dedican cada vez mayor número de recursos. Todo proyecto informático tiene como objetivo producir software de la mejor calidad posible, que cumpla, y si puede supere las expectativas de los usuarios

Aunque la calidad del software es un concepto relativamente reciente, son muchos los esfuerzos desarrollados en este campo en las últimas décadas, consiguiendo grandes avances y mejorar la convergencia con la calidad tradicional.

La calidad del producto, junto con la calidad del proceso, son algunos de los aspectos más importantes actualmente en el desarrollo de Software. De este modo, recientemente ha aparecido la familia de normas ISO/IEC 25000, relacionada con la calidad del producto y que proporciona una guía para el uso de la nueva serie de estándares internacionales llamada Requisitos y Evaluación de Calidad de Productos de Software.

A su vez, ISO/IEC 25000 constituye una serie de normas basadas en ISO/IEC 9126 y en ISO/IEC 14598 cuyo objetivo principal es guiar el desarrollo de los productos de software mediante la especificación de requisitos y evaluación de características de calidad. Todas estas normas se recogerán en detalle en la presente publicación.

El Colexio Profesional de Enxeñaría en Informática de Galicia, cuenta entre sus fines con la promoción y fomento del progreso de las actividades propias de la profesión, de la investigación, del establecimiento y uso de estándares. Fruto de este intento de mejora continua del nivel de calidad de las prestaciones profesionales en su ámbito de actuación, organiza el "I Foro para a Calidade do Software"

El objetivo de estas jornadas es crear un foro que reúna toda la información relativa a la mejora de la calidad del software, con el fin de proporcionar un acercamiento a las distintas familia de normas a particulares y empresas, facilitando la obtención de información y experiencias en la materia tanto a grandes empresas como a micropymes interesadas en mejorar su producto software. Para ello contamos con la participación de destacados expertos en la materia, cuyas aportaciones se recogen a modo de resumen en el presente libro.

Desde el CPEIG queremos agradecer a la AMTEGA su esfuerzo y apoyo en esta iniciativa que entendemos de gran valor para mejorar la confianza en los medios electrónicos e informáticos y, por consiguiente, para el desarrollo de la Sociedad de la Información en nuestro país.

Fernando Suárez Lorenzo

Presidente del Colexio Profesional de Enxeñaría en Informática de Galicia

Sobre los Autores

JAVIER GARZÁS PARRA

- javier.garzas@urjc.es
- twitter: @jgarzas
- web: www.javiergarzas.com

Cursó estudios postdoctorales y fue investigador invitado en la Universidad Carnegie Mellon (Pittsburgh, EE.UU). Doctor (Ph.D.) (cum laude por unanimidad) e Ingeniero en Informática (premio extraordinario).

Actualmente trabaja en Kybele y es profesor en la Universidad Rey Juan Carlos. Posee una experiencia de más de 15 años, trabajando para más de 80 organizaciones.

Edita el blog www.javiergarzas.com con más de 6 años y más de 900 post.

Scrum Master certificado por Jeff Sutherland (quien creó Scrum).

Además, cuenta con certificaciones de Auditor Jefe de TICs (Calificado por AENOR) para ISO 15504 SPICE – ISO 12207, Auditor ISO 20000 por ITSMF, especialización en Enterprise Application Integration (premiado por Pricewaterhousecopers), CISA (Certified Information Systems Auditor), CGEIT (Certified in the Governance of Enterprise IT) y CRISC (Certified in Risk and Information Systems Control) por la ISACA, CSQE (Software Quality Engineer Certification) por la ASQ (American Society for Quality),

Introduction CMMI-Dev y Acquisition Supplement for CMMI v1.2 (CMMI-ACQ) e ITIL V3 Foundation.

Ha trabajado en, o para, más de 80 organizaciones como INDITEX, TELEFÓNICA MÓVILES, INDRA, RENFE, DIRECCIÓN GENERAL DE TRÁFICO (DGT), MINISTERIO DE ADMINISTRACIONES PÚBLICAS (MAP), SISTEMAS TÉCNICOS DE LOTERÍAS (STL), AENOR, SIEMENS, INFORMÁTICA DE LA COMUNIDAD DE MADRID (ICM), EL MUNDO, BBVA, OCASO, etc.

Comenzó su carrera profesional como consultor senior y responsable del centro de competencias en ingeniería del software de ALTRAN, desde donde participa en proyectos para TELEFÓNICA MÓVILES CORPORACIÓN, INDRA tráfico aéreo o en la automatización de la simulación de la rotativa de EL MUNDO. Más tarde fue responsable de calidad software y de proyectos de mCENTRIC. Posteriormente, DIRECTOR EJECUTIVO Y DE INFORMÁTICA de la empresa de desarrollo de ERPs para la gestión universitaria con mayor número de clientes en España.

Experto en gestión y dirección de departamentos y fábricas software (realizando implantaciones de fábricas y mejoras en España, Colombia, Chile y Venezuela), con una amplia experiencia en ingeniería del software, calidad y mejora de procesos (participación en la mejora, evaluación o auditoría de procesos CMMI o ISO 15504 en más de 40 empresas).

Ha participado en numerosos proyectos de I+D nacionales e internacionales, ponencias, editado varios libros y publicado más de 100 trabajos de investigación. Evaluador de la ANEP (Agencia Nacional de Evaluación y Prospectiva) y experto certificado por AENOR y EQA para la valoración de proyectos I+D.

MOISÉS RODRÍGUEZ MONJE

Ingeniero Superior en Informática y Máster en Tecnologías Informáticas Avanzadas por la Universidad de Castilla-La Mancha. CISA (Certified Information System Auditor) por ISACA y Auditor Jefe por AENOR (ISO 15504). Formación oficial del SEI CMMI 1.2 Y CMMI-ACQ, ScrumManager Certified y TMAP Next Certified.

Ha trabajado como consultor para más de 20 empresas y administraciones, especializándose en la mejora de procesos software y la evaluación de la calidad del producto. Desde 2012 es CEO de Alarcos Quality Center, spin-off de la Universidad de Castilla-La Mancha orientada a prestar servicios de consultoría para la mejora de la calidad del software. Además, dirige AQC Lab, primer el laboratorio acreditado para la evaluación de la calidad del producto software según ISO/IEC 25000.

En el ámbito investigador, ha participado en más de 10 proyectos de I+D, dirigido varios Proyectos Fin de Carrera y escrito varios artículos y capítulos para revistas, libros y congresos sobre calidad software. Desde 2008 es miembro del SC7/GT26 de AENOR, para la elaboración de la ISO/IEC 29119.

DOMINGO GAITERO

Estudió informática en la UPM y en la UC3M de Madrid donde además ejerció como profesor auxiliar; lleva más de 31 años en el sector de las Tecnologías de la Información. Desde muy pronto se identificó e involucró con la Calidad del software. Ha trabajado tanto por empresas clientes finales como de servicios, y ha impartido clases y ponencias en universidades como Deusto, Politécnica de Cataluña y Politécnica de Sevilla en temas de Ingeniería del Software, Calidad y Lean IT. Ha diseñado y puesto en marcha dos Factorías de Software y Testing en España.

Ha colaborado en la práctica de SQA en las olimpiadas de Beijing, Vancouver y Londres donde dirigió el proyecto de certificación CMMI del proceso de desarrollo de software. Es Vicepresidente del Comité CSTIC en la AEC (Asociación Española de Calidad) y autor de varios libros, entre ellos la metodología Métrica v3 de la que fue uno de sus creadores.

En la actualidad es Socio Fundador de Proceso Social, Star up especializada en analizar el nivel de optimismo y pasión profesional de los trabajadores en proyectos de innovación o transformación tecnológica. Certificado como Excecutive Coach por la Escuela Europea de Coaching en Madrid (ECC), se ha especializado en la aplicación del coaching como herramienta motriz de motivación en personal TIC, denominado Qcoaching.

Es Blogero (www.procesosocial.com) y futuro TED talker.

Al margen del ámbito laboral, tiene una agitada vida donde además de educar y disfrutar de sus dos estupendos hijos, estudia magia, practica

aikido, juega al fútbol, y mantiene sus grandes pasiones por encima de todo: el cine y Star Wars.

PEDRO SEBASTIAN MINGO

Experto independiente en testing de software. Da soporte a equipos de desarrollo para facilitar la construcción y puesta en producción productos software de calidad y alto valor añadido.

Con 15 años trabajando en TI, ha dedicado los últimos al testing de software, especializándose en pruebas de rendimiento. Poco amigo de las verdades universales, su filosofía es adaptar el testing al contexto y reforzar las relaciones entre los equipos de testing y desarrollo para que compartan objetivos y optimicen resultados.

Muy interesado en la responsabilidad social para con la profesión, interviene en cursos, seminarios y ponencias. Últimamente ha empezado a escribir un blog con la idea de compartir conocimientos y reflexionar sobre el mundillo del testing y la calidad del software.

DAVID GÓMEZ GARCÍA

David Gómez es ingeniero técnico en informática de Sistemas por la UPM. Actualmente trabaja como Consultor Tecnológico en Autentia. Con más de 14 años de experiencia, ha participado en el desarrollo, ha liderado y realizado mentoring de proyectos en Banca, Seguros, Defensa y Transporte Marítimo y Terrestre.

David combina su labor de desarrollo con la formación, siendo instructor certificado oficial de SpringSource y de JavaSpecialists.eu, habiendo impartido más de 30 cursos desde 2009 hasta 2013. También ha participado con charlas en eventos como Spring IO (2011 y 2012), Codemotion Madrid 2012 o Mediterranea API Days 2013; en universidades (Universidad Jaume I de Castellón, Universidad de Valencia y Universidad Politécnica de Madrid); y ha sido ponente en comunidades de desarrollo como MadridJUG, MadridJS o MilanoJUG.

Dentro de los eventos, David ha sido co-organizador de Codemotion Madrid en las dos primeras ediciones de 2012 y 2013 y del OpenSpace SaveInformaticOS, así como colaborador en JCrete.org 2013.

JESÚS HERNANDO CORROCHANO

Es Director del Departamento de Ingeniería de Software de Sistemas de Información de El Corte Inglés, Profesor Asociado de la Universidad Carlos III de Madrid. Licenciado en Ciencias Matemáticas (Computación), Universidad Complutense, Madrid Mater en Gobierno TI. CEURA, Experto en Calidad Industrial UNED, Master Redes y Comunicaciones, FYCSA, Madrid.

ENRIQUE SÁNCHEZ

Ingeniero en Informática y Máster en Ingeniería de Decisión por la URJC, actualmente está realizando doctorado en Ciencias Computacionales e Inteligencia Artificial en esta universidad. Technical Team Leader en Mediante con más de 4 años de experiencia en Testing y Calidad en empresas como BBVA, Tuenti y Telefónica.

Apasionado por la mejora de los procesos de creación de software y la automatización de tareas, investiga constantemente sobre nuevas técnicas y metodologías que permitan ayudar a los equipos a mejorar constantemente.

BDD Evangelist, participa activamente en la comunidad de QA de Madrid (MADQA) dando charlas y ayudando en la organización de los eventos.

En sus ratos libres le encanta viajar, la comida tailandesa y el cine.

ANA MARIA DEL CARMEN GARCÍA OTERINO

Participa con Kybele Consulting en proyectos relacionados con la implantación de metodologías ágiles, Scrum y calidad del software para importantes organizaciones.

También colabora en The smart software quality (http://www.thesmartsoftwarequality.com/), evaluando la calidad del software y la cobertura de la ISO 25000 de distintos proyectos.

Apasionada `por la calidad del software y buenas prácticas en general. Ha contribuido en blogs de referencia en esta temática y está escribiendo un libro sobre SonarQube.

Ha escrito publicaciones y realizado ponencias relacionadas con calidad del software, entre las que destacan la participación en el Academic ITGSM14 (organizado por el itSMF) y la revista NovATIca.

INTRODUCCIÓN

En el mundo del software, y supongo que en otras disciplinas también, cuando nos referimos al amplio concepto de "calidad software" hemos de ser muy conscientes de que en realidad ese "etéreo" concepto de calidad se subdivide, principalmente, en tres tipos de calidad: la del proceso, la del producto y la de las personas/equipos.

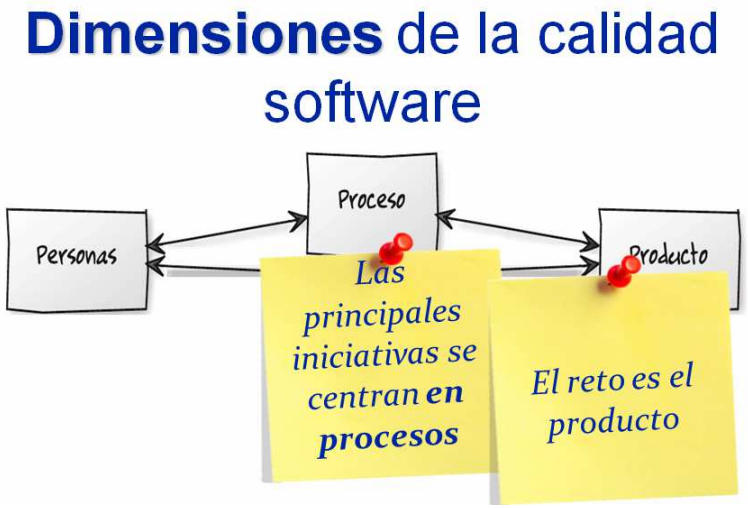


Figura 1. Dimensiones de la calidad de software

A pesar de que este tema es muy antiguo, en el mundo empresarial es algo que no está tan claro y maduro. Y, a la hora de la verdad, cuando se entrega el software, aquello que en la oferta / contrato parecía tan claro resulta que

produce enormes desengaños. “¿Pero cómo puede ser qué la empresa “x” con CMMI nivel “y” nos entregue este producto tan malo?” “¿Pero no usaban la metodología “z”?”

Si quieres calidad en general, vas a necesitar conocer los tres ámbitos anteriores. Aunque depende del rol que juegues te puede importar más una perspectiva de calidad u otra. Si eres un cliente que solo compra software no puedes perder la prioridad, procesos, productos y personas son importantes, pero la calidad del producto es para ti determinante. Si eres fabricante de software, vas a necesitar los tres y no puedes olvidar ninguno.

Por todo esto es preciso diferenciar y tener claro cuáles son las distintas áreas que abarca la calidad de software.

Javier Garzás Parra

Capítulo

1

1. CALIDAD DEL PRODUCTO SOFTWARE, TU EMPRESA TENDRÁ UN PROBLEMA SI NO SE PREOCUPA POR ELLA.

Autor: Javier Garzás Parra

"- ¿Es posible revertir la entropía?

- Los dos sabemos que la entropía no puede revertirse. No puedes volver a convertir el humo y las cenizas en un árbol."

-- Isaac Asimov (La última pregunta).

1.1 LA CALIDAD DEL PROCESO

La calidad vista desde el mundo de los procesos nos dice que la calidad del producto software está determinada por la calidad del proceso. Por proceso se entienden las actividades, tareas, entrada, salida, procedimientos, etc., para desarrollar y mantener software.

Modelos, normas y metodologías típicas aquí son CMMI, ISO 15504 / ISO 12207, el ciclo de vida usado; incluso las metodologías ágiles entran aquí.

1.2 LA CALIDAD DEL PRODUCTO

Existen modelos de calidad de producto, destacando entre ellos la ISO 9126 (ISO, 2001), o la nueva serie ISO 25000 (ISO, 2005a), que especifica diferentes dimensiones de la calidad de producto. Aunque aquí la dura tarea de evaluación recae en el uso de métricas software.

1.3 LA CALIDAD DEL EQUIPO Y/O PERSONAS

Si hubiese que elegir, de entre las claves que determinan el éxito (o fracaso) de un proyecto software, me aventuraría a decir que este sería el papel que juegan “las personas”.

En mayor o menor medida, prácticamente todo aquel que ha estudiado el éxito o fracaso de un proyecto software ha destacado el papel que las personas, el equipo de desarrollo, juegan en el mismo. Concluyendo, en la mayoría de ocasiones, con que las personas son el factor más determinante.

Como decía Glass (Glass, 2003), no hay que olvidar que las personas son las que hacen el software. Las herramientas ayudan, las técnicas también, los procesos, etc. Pero sobre todo esto están las personas. “Las personas son la clave del éxito”, que dijera Davis en su genial libro (Davis, 1995). El equipo humano que, como decía Cockburn (Cockburn, 2000), es el componente no lineal de primer orden en el desarrollo software. Decía McConnell (S. McConnell, 1996) que las personas son los que tienen más potencial para

recortar el tiempo de un proyecto, y que quienes han trabajado en software "han observado las enormes diferencias que hay en los resultados que producen entre desarrolladores medios, mediocres y geniales".

Después de analizar 69 proyectos Boehm (Boehm, 1981) comprobó que los mejores equipos de desarrolladores eran hasta 4 veces más productivos que los peores. Por su parte DeMarco y Lister, en su Peopleware (DeMarco & Lister, 2010), identifican diferencias de productividad de 5.6 a 1, extraídas de un estudio con 166 profesionales de 18 organizaciones. En la NASA (Weinberg & Schulman, 1974), observaron diferencias de hasta 3 a 1 en productividad entre sus diferentes proyectos. E incluso mucho antes, en el 74, había ya estudios (Valett & McGarry, 1989) que habían observado diferencias de 2.6 a 1 en equipos a la hora de realizar las mismas tareas de desarrollo.

Pero aunque para algunos esto esté muy claro, no siempre es así en todos los proyectos. Recuerdo que hace años trabajaba en una empresa que desarrollaba productos software… en la que no pensaban exactamente así. Allí se pensaba que los desarrolladores no eran lo más determinante, eran "piezas" intercambiables. Que lo determinante era la parte comercial y la funcional, los técnicos eran una "commodity". De ahí que gran parte del equipo técnico no tenía el perfil, la cualificación y estudios acordes para desarrollar el software que se les encomendaba, por cierto, una aplicación bastante grande. Los jefes de proyecto, y directores técnicos, jamás habían estudiado alguna carrera técnica. Lo curioso era que, aunque todo el mundo veía los grandes problemas de calidad software y de gestión del proyecto, nadie parecía ver la principal causa de los mismos.

La gente que participa en un desarrollo software, no son como obreros en una cadena de montaje. No son tan fácilmente intercambiables, y el trabajo no es tan repetible como en estos otros trabajos. Más aún si no se ha implantado una calidad software de verdad (no sólo una certificación).

Podemos encontrar decenas de aproximaciones para mejorar la calidad de las personas, que van desde el tan de moda coaching, a la filosofía ágil de lograr la auto-organización de los equipos, estrategias de motivación, combinaciones de los anteriores, etc. E incluso hasta hay modelos, como son los TSP y PSP.

1.4 ¿QUÉ PERSPECTIVA DE CALIDAD ES MÁS IMPORTANTE?

Como siempre, la necesidad de enfatizar en una u otra perspectiva (producto – proceso) depende del contexto en que se desenvuelva la organización, y de no caer en aquello que tanto sucede en ingeniería del software y que es llevar los últimos métodos, modelos, estrategias, etc., al extremo, sin pensar en su adecuación e idoneidad a nuestro negocio y al contexto de nuestra empresa.

Por ejemplo, para una empresa que no desarrolla, que adquiere productos software desarrollados por terceros (externalización), la certificación de la calidad del proceso de su subcontratista puede ser condición necesaria e importante como garantía de calidad, sobre todo en procesos de selección de proveedores, cuando aún no está el software desarrollado, pero puede no ser suficiente para garantizar la calidad del producto. Será la calidad del producto la que evidenciará inequívocamente la calidad del mismo, sin necesidad de suposiciones; un conjunto coherente de métricas e indicadores del producto estructurados según un modelo tipo ISO 9126

mostrará la calidad exacta del mismo. O, por ejemplo, en otro caso diferente, en una empresa de desarrollo, habiendo determinado la calidad del producto con, por ejemplo, un modelo basado en la ISO 9126, será un modelo de procesos el que nos ayude a mejorarlo (complementando ambos modelos sin olvidar ninguno).

Si una empresa que desarrolla software debe preocuparse de la calidad del proceso y del producto que desarrolla y entrega, una empresa que solo compra software (el típico cliente) debería, principalmente, preocuparse de la calidad del producto que compra. Aunque vemos que en la realidad, las empresas que compran software lo hacen al revés, se preocupan por el proceso que usa su proveedor (CMMI, ISO, etc.) y apenas del producto que les llega. Cosas de la industria.

1.5 RAZONES POR LAS QUE LA CALIDAD DEL PRODUCTO SOFTWARE DEBIERA PREOCUPARTE MUCHO

1.5.1 CALIDAD DEL PRODUCTO NO ES LO MISMO QUE TESTING

Que el software "funcione" (lo que normalmente intentas saber con el testing) no significa que por ello… ¡el producto este bien hecho! Y si está mal hecho no te vas a dar cuenta si no lo miras por dentro (miras sus fuentes, código, diseño, cómo de mal o bien está programado, si está muy acoplado, etc.), es decir, si no miras lo que se llama calidad del producto software.

1.5.2 LA CALIDAD DEL PROCESO NO GARANTIZA LA CALIDAD DEL PRODUCTO

En el desarrollo software, de entre las diferentes perspectivas con que se puede observar la calidad, hay dos especial y tradicionalmente importantes: la calidad del producto en sí y la calidad del proceso para obtenerlo (o actividades, tareas, etc., para desarrollar y mantener software). Dos dimensiones esenciales, estudiadas desde hace tiempo por los grandes "padres" de los modelos y teorías de calidad en general y también aplicables a la construcción de software, y que giran e interactúan en torno a la idea de que, como comenta Humphrey, "padre" del modelo CMMI, "la calidad del producto está determinada por la calidad del proceso usado para desarrollarlo"(Humphrey, 2005). Aunque en el área del desarrollo software, que siempre ha ido un poco más atrás en temas de calidad, y en España, que en los últimos años se ha empezado a tratar en las empresas este tipo de aspectos, la popularidad e importancia a nivel industrial ha recaído casi por completo en los modelos de calidad de procesos, destacando el conocido modelo CMMI, que en los últimos años se ha extendido considerablemente.

Así modelos de calidad de procesos como CMMI son bastante populares en el mundo del desarrollo, y se están convirtiendo poco a poco en requisito imprescindible para un centro de desarrollo o fábrica software. En algunos casos hasta el punto en que se ha llegado a asumir que cumplir cierto modelo o nivel de madurez de procesos asegura productos de calidad (que es lo más importante para ciertas empresas y entidades, sobre todo si han externalizado el desarrollo, donde lo que reciben periódicamente son productos de desarrollos de sus proveedores). Pero… ¿realmente es

garantía suficiente? ¿Una certificación sobre la calidad del proceso garantiza un producto de calidad?

Con respecto a este tema ha habido mucha controversia. Por ejemplo, hace tiempo comentaban Kitchenham y Pfleeger en un artículo en IEEE software (Kitchenham & Pfleeger, 1996) que la principal crítica a esta visión es que hay poca evidencia en que cumplir un modelo de procesos asegure la calidad del producto, la estandarización de los procesos garantiza la uniformidad en la salida de los mismos, lo que "puede incluso institucionalizar la creación de malos productos". Más recientemente Maibaum y Wassyng, en Computer (Maibaum & Wassyng, 2008), comentaban, siguiendo la misma línea, que las evaluaciones de calidad deberían estar basadas en evidencias extraídas directamente de los atributos del producto, y no en evidencias circunstanciales deducidas desde el proceso. Un proceso estándar, o institucionalizado, según sea la terminología del modelo de uso, no necesariamente concluye con un producto de calidad.

Si bien modelos como CMMI han gozado de mucha popularidad, no por ello los modelos o estándares de calidad de producto tienen menos madurez, destacando el menos popular pero igualmente importante ISO 9126, o la nueva serie ISO 25000, que especifica diferentes dimensiones de la calidad de producto.

En nuestra experiencia nos hemos encontrado bastante frustración en ciertas empresas debido a las esperanzas depositadas en los modelos de calidad de procesos que ofrecían sus proveedores y que finalmente no han servido como garantía de calidad de los productos que recibían.

1.5.3 CMMI NO ASEGURA UN PRODUCTO DE CALIDAD

Tener un "sello" de CMMI no siempre asegura un producto software de calidad. Así es. El "sello" es una evidencia "indirecta" de calidad, la calidad del producto software es evidencia "directa". El sello, la certificación, evaluación, o como cada uno lo llame (para más detalles tienes la guía de supervivencia CMMI (Garzás et al., 2011)), en modelos como CMMI…

a) Se basa en un muestreo (no se ven todos los proyectos de la empresa), así que puedes tener mala suerte y que te toque un proyecto – equipo de desarrollo que no se evaluó.

b) Las auditorías CMMI no miran la calidad del producto software, sólo miran si se cumplen buenas prácticas del proceso, si se gestionan requisitos, si se verifica, si se planifican los proyectos, etc., pero no si esos requisitos tomados están bien, si ese plan de proyecto está bien… y mucho menos ¡cómo está el código!

c) Si eres un cliente y contratas a alguien porque tiene algún CMMI te mostrará un "sello" concedido en el pasado, y tu producto te lo entregará muchos meses después de la concesión del "sello" y en ese tiempo… pueden pasar muchas cosas.

1.5.4 LA MALA CALIDAD DEL PRODUCTO SIEMPRE TIENE UN COSTE

Porque la mala calidad del producto software (es decir, código espagueti, código repetido, diseño acoplado, etc.) siempre, siempre, alguien la paga (euros), tiene un coste (lo que llamamos deuda técnica, sección 1.5.4.1). Y solo pueden pagarla uno de dos: el cliente o la empresa que desarrolló el software. Lo que nos lleva al concepto de deuda técnica…

1.5.4.1 Deuda técnica

La metáfora de la "deuda técnica" aplicada al desarrollo software la introdujo hace dos décadas Ward Cunningham (Cunningham, 1992) para explicar a los "no técnicos" la necesidad de "refactorizar".

> Porque la mala calidad del producto software siempre alguien la paga (euros), tiene un coste.

Desde entonces, la deuda técnica se ha utilizado para describir muchos otros tipos de deudas o males del desarrollo de software, y se ha aplicado a cualquier cosa que aumente innecesariamente los esfuerzos de desarrollo, se interponga en la futura evolución o venta de un sistema de software. Hoy podemos encontrar la deuda de las pruebas, la deuda de las personas, la deuda de la arquitectura, la deuda de los requisitos, la deuda de la documentación, etc.

La deuda técnica es el coste y los intereses a pagar por hacer mal las cosas. El sobre esfuerzo a pagar para mantener un producto software mal hecho, y lo que conlleva, como el coste de la mala imagen frente a los clientes, etc. Hay quien no es ni siquiera consciente de que está pagando intereses por hacer mal el software, y continua así hasta el "default".

La deuda técnica al final siempre alguien la paga. O la paga el proveedor que desarrolla el software o la paga el cliente que lo usa o compra.

Principal causa de la deuda técnica: la presión de las fechas

La mayoría de los autores coinciden en que la principal causa de la deuda técnica es la presión en fechas y planes.

Sin embargo, hay muchas otras causas, como la falta de cuidado, falta de formación, la no verificación de la calidad, o la incompetencia.

Con el tiempo, el término deuda técnica se ha perfeccionado y ampliado, principalmente por Steve McConnell con su taxonomía (S. McConnell, 2007) y Martin Fowler con sus cuatro cuadrantes (M. Fowler, 2009).

La pequeña taxonomía de McConnell, habla de que:

- No hay deuda técnica, si… Hay retrasos, recortes, etc., que no requieren el pago de intereses. No todo el trabajo incompleto es deuda.
- Si hay deuda técnica… puede ser (I) Deuda incurrida involuntariamente debido a trabajos de baja calidad o (II) Deuda incurrida intencionalmente.

Dándole una vuelta más al término, la Tabla 1 muestra cuatro tipos de posibles mejoras o tareas a realizar en el futuro para aumentar el valor del producto software, como pueden ser ampliar funcionalidades (color verde, que es en lo que suelen fijarse las empresas), o invertir en arquitectura (amarillo), invertir reducir los defectos (rojo) o la deuda técnica (negro), que es invisible y tiene un efecto negativo.

	Visible	Invisible
Valor positivo	- Nuevas características - Funcionalidad añadida	- Características arquitectónicas, estructurales
Valor negativo	Defectos	Deuda técnica

Tabla 1.Tareas de mejora para aumentar el valor del producto software (Kruchten, 2012)

1.5.4.2 ¿Cómo se calcula la deuda técnica? El problema de las fórmulas de deuda técnica.

La metáfora de la deuda técnica, nos da la gran posibilidad de transmitir a personas no-técnicas, y desde un punto de vista económico, los problemas que implica el software de mala calidad, la necesidad de hacer buen código, la necesidad de refactorizar el código, etc.

Sin embargo, aún nos queda un "pequeño" problema por resolver… actualmente no hay una única manera de cuantificar la deuda técnica de un software.

Tampoco hay un consenso sobre qué debe considerarse deuda técnica. Por ejemplo podemos encontrar que dentro de deuda técnica queda incluida la falta de cobertura de pruebas, pero también que nuestro software dependa de una plataforma específica, que el desarrollador no entienda bien como aplicar un patrón de diseño, o incluso que la arquitectura de nuestro software no sea la más adecuada.

Y entonces, ¿hay alguna fórmula para calcular la deuda técnica? Sí, varias. ¿Y cuáles son? Varios autores proponen diferentes fórmulas para cuantificar la deuda técnica. Pero aún así, podemos agrupar las principales fórmulas en dos grupos:

Estimación del valor de deuda técnica del proyecto, sin tener en cuenta los intereses que genera la deuda.

Por un lado, hay autores que solo enuncian fórmulas que se centran en dar un valor de la deuda técnica actual del proyecto, sin calcular sus intereses en el futuro. Dentro de este grupo, nos encontramos a los creadores del plugin de deuda técnica de la herramienta Sonar (SonarQube, 2011) o Jean-Louis Letouzey con el método SQALE (Letouzey & Ilkiewicz, 2012).

A grandes rasgos, ambos cuantifican la deuda técnica basándose en encontrar ciertas malas prácticas en el código y los costes que conlleva solucionar esas malas prácticas.

En el caso del plugin de Sonar se detecta el código duplicado, violaciones (como por ejemplo nombrar a un método que no es un constructor igual que a una clase o no emplear métodos get() y set() etc.), no comentar APIs públicas, complejidad ciclomática que supera un cierto umbral, cobertura de pruebas menor de un cierto valor y dependencias entre paquetes que superen un valor concreto.

Con intereses

Otros autores, pretenden llevar la metáfora de deuda técnica más allá, y no sólo dan mecanismos para calcular el valor de deuda técnica actual (al que

llaman principal), sino que empiezan a hablar de distintos tipos de intereses que genera la deuda técnica.

A pesar de que en este grupo los autores tienen en común que están de acuerdo en diferenciar entre deuda técnica principal e intereses, plantean distintos enfoques:

Chin, Huddleston y Gat (Chin et al., 2010), consideran que la mayoría del software atraviesa períodos de desarrollo activo al comienzo y mantenimiento después. Por ello, su fórmula de deuda técnica es la suma de la deuda técnica acumulada en el período de desarrollo más la deuda técnica que se acumulará en el período de mantenimiento. Todo ello teniendo en cuenta distintos tipos de intereses.

Para Bill Curtis, Jay Sappidi y Alexandra Szynkarski (Curtis et al., 2012), la deuda técnica principal, se calcula obteniendo los problemas estructurales del código (a través de una herramienta de análisis), clasificando dichos problemas según su grado de severidad (alto, medio, bajo), estableciendo qué tanto por ciento de cada tipo de problema se va a solucionar, y el tiempo que tardaríamos en solucionar dicho problema.

Gary Short (Short, 2010), calcula la deuda técnica principal desde otra perspectiva, teniendo en cuenta aspectos como el número total de empleados que trabajan eliminando la deuda técnica, sus salarios, el coste de comprar e instalar el hardware que se necesite o incluso una estimación del daño que puede provocar la deuda técnica a la imagen de la empresa.

Por último, (Nugroho et al., 2011), calculan la deuda técnica considerando los intereses y basándose en un modelo de calidad propio creado por el

grupo SIG (Software Improvement Group) basado a su vez en la ISO 9126. En este caso, el "interés" es el coste extra invertido en mantener el software por tener una mala calidad técnica. Además, también tienen en cuenta que tanto la deuda técnica como los intereses aumentan con el tiempo y proponen una fórmula para calcular ese crecimiento.

1.5.5 EL CLIENTE PUEDE DETECTAR LA MALA CALIDAD DEL PRODUCTO SOFTWARE

Si el cliente detecta mala calidad del producto software será el proveedor quien acabe pagando el trabajo mal hecho, aunque lo normal es que el cliente no se dé cuenta del mal desarrollo software por el que acaba de pagar y acabe pagando él mismo la mala calidad del desarrollador… que normalmente la paga en sobre exceso de horas y horas de mantenimiento.

1.5.6 LAS BUENAS PRÁCTICAS NO ASEGURAN CALIDAD DEL PRODUCTO

Las buenas prácticas de ITIL, ISO 20000, son muy buenas para detectar que los usuarios están teniendo problemas con en el software, las incidencias que los usuarios tienen con él, para organizar los pasos a producción de los parches, controlar cuantas veces "se ha caído" el software en producción, su disponibilidad, etc. Pero por muchos y muy buenos indicadores que tenga "tu coche" sobre si se está "calentando el motor", "el aceite que queda", etc., los problemas se reparan en "el motor" (el desarrollo software) no poniendo indicadores del nivel de servicio. Y el motor se arregla trabajando la calidad del producto software. Una certificación de la calidad de los procesos no siempre asegura un producto de calidad

1.6 REFERENCIAS

Asimov, I. (1956), La última pregunta. *Science Fiction Quarterl)*

Atwood, J. (2004). *When good comments go bad.* 2013. Disponible en: http://www.codinghorror.com/blog/2004/11/when-good-comments-go-bad.html

Beck, K., & Fowler, M. (1999). Bad smells in code. *Refactoring: Improving the design of existing code* Addison Wesley.

Belady, L., & Lehman, M. M. (1976). A model of large program development.

Belady, L., & Lehman, M. M. (1985). *Program evolution: Processes of software change*

Boehm, B. (1981). *Software engineering economics* Prentice Hall PTR.

Chin, S., Huddleston, E., Bodwell, W., & Gat, I. (2010). The economics of technical debt.

Cockburn, A. (2000). Characterizing people as non-linear, first-order components in software development. Artículo presentado en *4th International Multi-Conference on Systems, Cybernetics and Informatics,* Orlando, Florida.

Cunningham, W. (1992). The WyCash portfolio management system. Artículo presentado en *ACM SIGPLAN OOPS Messenger, , 4.* (2) pp. 29-30.

Curtis, B., Sappidi, J., & Szynkarski, A. (2012). Estimating the size, cost, and types of technical debt.

Davis, A. M. (1995). *201 principles of software development* McGraw-Hill Inc.,US.

DeMarco, T., & Lister, T. R. (2010). *Peopleware: Productive projects and teams* Addison Wesley Pub Co Inc.

Fowler, M. (2009). *Technical debt.* 2013. Disponible en: http://martinfowler.com/bliki/TechnicalDebt.html

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2000). *Refactoring: Improving the design of existing code* (1st edition ed.) Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns*. New York, NY, USA. Addison-Wesley Professional.

Garzás, J., Irrazábal, E., & Santa Escolástica R. (2011, Guía práctica de supervivencia en una auditoría CMMI. *Boletín De La ETSII, Universidad Rey Juan Carlos, 002*, 1-33.

Garzás, J., & Piattini, M. (2006). A catalog of object oriented design rules. In J. Garzás, & M. Piattini (Eds.), *Object-oriented design knowledge: Principles, heuristics, best practices* (pp. 307-347). Hershey (EEUU): Idea Group Inc.

Glass, R. L. (2003). *Facts and fallacies of software engineering* Addison Wesley.

Humphrey, W. H. (2005). Acquiring quality software (online on http://www.stsc.hill.af.mil/CrossTalk/2005/12/0512Humphrey.html). *CrossTalk,*

ISO/IEC 9126. Software Product Evaluation–Quality Characteristics and Guidelines for their use. (International Organization for Standarization 2001).

ISO. (2005a). *ISO/IEC 25000 software and system engineering – software product quality requirements and evaluation (SQuaRE) –Guide to SQuaRE.*

ISO/IEC 25010:2011- Systems and Software Engineering -- Systems and Software Quality Requirements and Evaluation (SQuaRE) -- System and Software Quality Models, (2005b).

ISO/IEC 25012:2008 - Software Engineering -- Software Product Quality Requirements and Evaluation (SQuaRE) -- Data Quality Model, (2005c).

ISO/IEC 25020:2007 - Software Engineering -- Software Product Quality Requirements and Evaluation (SQuaRE) -- Measurement Reference Model and Guide, (2005d).

ISO/IEC 25021:2012 - Systems and Software Engineering -- Systems and Software Quality Requirements and Evaluation (SQuaRE) -- Quality Measure Elements, (2005e).

ISO/IEC 25030:2007 - Software Engineering -- Software Product Quality Requirements and Evaluation (SQuaRE) -- Quality Requirements, (2005f).

ISO/IEC 25040:2011 - Systems and Software Engineering -- Systems and Software Quality Requirements and Evaluation (SQuaRE) -- Evaluation Process, (2005g).

ISO/IEC 25041:2012 - Systems and Software Engineering -- Systems and Software Quality Requirements and Evaluation (SQuaRE) -- Evaluation Guide for Developers, Acquirers and Independent Evaluators, (2005h).

ISO/IEC FDIS 25000 - Systems and Software Engineering -- Systems and Software Quality Requirements and Evaluation (SQuaRE) -- Guide to SQuaRE, (2005i).

ISO/IEC FDIS 25001 - Systems and Software Engineering -- Systems and Software Quality Requirements and Evaluation (SQuaRE) -- Planning and Management, (2005j).

ISO/IEC TR 25060:2010 - Systems and Software Engineering -- Systems and Software Product Quality Requirements and Evaluation (SQuaRE) -- Common Industry Format (CIF) for Usability: General Framework for Usability-Related Information, (2005k).

Kernighan, B. W., & Plauger, P. J. (1974). *The elements of programming style* McGraw-Hill.

Kitchenham, B., & Pfleeger, S. L. (1996). Software quality: The elusive target. *IEEE Software, 20*(1), 12-21.

Kruchten, P. (2012). *Interview with philippe kruchten on technical debt.* 2013. Disponible en: http://blog.techdebt.org/interviews/156/interview-

with-philippe-kruchten-on-technical-debt-rup-ubc-decision-process-architecture

Larbi, S. (2008). *Common excuses used to comment code and what to do about them.* 2013. Disponible en: http://www.codeodor.com/index.cfm/2008/6/18/Common-Excuses-Used-To-Comment-Code-and-What-To-Do-About-Them/2293

Letouzey, J., & Ilkiewicz, M. (2012). Managing technical debt with the SQALE method.

Maibaum, T., & Wassyng, A. (2008). A product-focused approach to software certification.

Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship* Prentice Hall.

McBreen, P. (2001). *Software craftsmanship: The new imperative* Addison-Wesley Professional.

McCabe, T. (1976). A complexity measure.

McConnell, S. (2004). *Code complete: A practical handbook of software construction: Second edition* Microsoft Press.

McConnell, S. (2007). *Technical debt.* 2013. Disponible en: http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx

McConnell, S. (1996). *Rapid development* Microsoft Press.

McConnell, S. (2006). *Software estimation: Demystifying the black art* Microsoft Press.

Nugroho, A., Kuipers, T., & Visser, J. (2011). An empirical model of technical debt and interest.

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules.

Putnam, L. H., & Myers, W. (2003). *Five core metrics: The intelligence behind successful software management* Dorset House.

Riel, A. J. (1996). *Object-oriented design heuristics* Addison-Wesley Professional.

Roberts, D. B. (1999). *Practical analysis for refactoring*. Champaign, IL, USA. University of Illinois at Urbana-Champaign.

Short, G. (2010). *Paying back the technical debt.* 2013. Disponible en: http://www.slideshare.net/garyshort/technical-debt-2985889

SonarQube. (2011). *Technical debt plugin.* Fecha de consulta: 07/26 2013. Disponible en: http://docs.codehaus.org/display/SONAR/Technical+Debt+Calculation

Tokuda, L., & Batory, D. (2001). Evolving object-oriented designs with refactorings. *Automated Software Engg., 8*(1), 89-120.

Valett, J. D., & McGarry, F. E. (1989). A summary of software measurement experiences in the software engineering laboratory.*Journal of Systems and Software 9(2)*

Weinberg, G. M., & Schulman, E. L. (1974). Goals and performance in computer programming.

Wiederhold, G. (2006). Whati s your software worth?*Communications of the ACM 49: 65-75*

Capítulo

2

2. CERTIFICACIÓN CALIDAD DEL PRODUCTO SOFTWARE CON ISO 25000.

Autor: Moisés Rodríguez Monje

"La calidad es gratis, pero sólo para aquellos que están dispuestos a pagar un alto precio por ello."

-- T. DeMarco y T. Lister (Peopleware)

2.1 SITUACIÓN ACTUAL DE LA CALIDAD SOFTWARE

Hoy en día la demanda de software se ha disparado, puesto que está presente en todos los dispositivos que manejamos, en los sistemas de gestión, en el transporte, en las comunicaciones, en la energía, en la banca, en nuestro ocio y en nuestro entretenimiento. Este aumento del software ha dado lugar a un crecimiento de las empresas encargadas de su desarrollo, lo que se conocen como "factorías de software". A su vez, la

falta de personal especializado para ciertas tareas del desarrollo software, así como la búsqueda de la reducción de costes han dado lugar a lo que se conoce como "*outsourcing*" del desarrollo software. Sin embargo, cuando se externalizan actividades de desarrollo software, también aumentan los riesgos y la falta de control sobre la calidad del software que la empresa contratada entrega, surgiendo la necesidad de evaluar y asegurar la calidad del software de dichas empresas desarrollan.

Desde sus inicios, la evaluación de la calidad del software se ha centrado en controlar la calidad de los procesos que se utilizan para su desarrollo, surgiendo así modelos y estándares como CMMI o ISO/IEC 15504, que en España han calado profundamente, siendo el cuarto país a nivel mundial en número de evaluaciones oficiales de CMMI y uno de los más relevantes en ISO/IEC 15504 con más de 50 empresas certificadas. Sin embargo, a pesar de la calidad de los procesos utilizados en el desarrollo, se siguen leyendo noticias sobre los problemas de calidad que los productos software tienen una vez puestos en producción. Y es que hay poca evidencia de que cumplir un modelo de procesos asegure la calidad del producto software desarrollado, y aunque la estandarización de los procesos garantiza la uniformidad en la salida de los mismos, podría llegar a darse el caso de que institucionalizara la creación de malos productos [1]. En este sentido, nosotros estamos de acuerdo con que las evaluaciones deberían basarse en evidencias directas del propio producto, y no solo en evidencias del proceso de desarrollo [2].

La calidad de los procesos no es suficiente, es necesario también evaluar y mejorar las características del propio producto software.

Por todo lo anterior, es cada día mayor el número de organizaciones que se interesan, no solo por la calidad de los procesos que se siguen en el desarrollo de software, sino también por la calidad de los productos que desarrollan y/o adquieren. Surgiendo así la necesidad de normas y estándares que definan las características de calidad del producto software, así como el proceso que se debe seguir para poder realizar la evaluación de dichas características.

2.2 LA NORMA ISO/IEC 25000

A lo largo de los últimos años se han elaborado trabajos de investigación, normas y estándares, con el objetivo de crear modelos, procesos y herramientas de evaluación de la calidad del propio producto software, entre los que se pueden destacar los presentados en [3-6]. Precisamente para dar respuesta a estas necesidades nace la nueva familia de normas ISO/IEC 25000 conocida como SQuaRE (*Software Product Quality Requirements and Evaluation*), que tiene por objetivo la creación de un marco de trabajo para evaluar la calidad del producto software, sustituyendo a las anteriores ISO/IEC 9126 e ISO/IEC 14598 y convirtiéndose así en el referente a seguir.

2.3 ESTRUCTURA DE LA FAMILIA ISO/IEC 25000

La ISO/IEC 25000 se encuentra compuesta de varias partes o divisiones, entre las que podemos destacar:

- La ISO/IEC 25010 [7] que determina las características de calidad del producto software que se pueden evaluar (Figura 1). En total son 8 las

características de calidad que identifica: funcionalidad, rendimiento, compatibilidad, usabilidad, fiabilidad, seguridad, mantenibilidad y portabilidad.

- La ISO/IEC 25040 [8] que define el proceso de evaluación de la calidad del producto software, compuesto por cinco actividades:

    - Establecer los requisitos: para determinar cuáles son los requisitos de calidad que se deben considerar a la hora de evaluar el producto.

    - Especificar la evaluación: indicando las métricas, criterios de medición y evaluación a tener en cuenta.

    - Diseñar la evaluación: definiendo el plan de actividades que se realizarán para evaluar el producto.

    - Ejecutar la evaluación: realizando las actividades de medición y evaluación del producto, considerando los criterios identificados en las fases previas.

    - Concluir la evaluación: elaborando el informe de evaluación y realizando la disposición de resultados e ítems de trabajo.

Figura 1. Modelo de calidad del producto software según la ISO/IEC 25010

- La ISO/IEC 25020 que será la encargada de definir las métricas de calidad del producto software y que todavía está en pendiente de publicación, por lo que no existe un acuerdo respecto a los indicadores y umbrales que se deben considerar para poder determinar la calidad de un producto software de manera estandarizada.

2.4 ECOSISTEMA DE EVALUACIÓN Y CERTIFICACIÓN DEL PRODUCTO SOFTWARE

Por otro lado, los modelos y normas anteriores relacionados con la evaluación del producto software, no tratan el proceso posterior de la certificación, que permita a las empresas superar una auditoría realizada por una entidad acreditada y obtener un certificado que refleje la calidad de su producto software. Por ello, durante el año 2012 realizamos una revisión sistemática [9] siguiendo la guía propuesta por Kitchenham en [10]. Como resultado se obtuvo un conjunto de 10 estudios primarios que cumplían con los requisitos de búsqueda. La principal conclusión que se extrajo de estos estudios fue que existe interés y necesidad por extender la

certificación de la calidad de los procesos a la calidad del producto software, pero no existe ninguna propuesta firme para ello basada en la nueva ISO/IEC 25000, así como que tampoco se tiene claro el conjunto de entidades involucradas en el proceso de certificación.

Por estas razones, Alarcos Quality Center (spin-off de la Universidad de Castilla-La Mancha) en colaboración con AENOR (Asociación Española de Normalización y Certificación) hemos definido el "Ecosistema para la Evaluación y Certificación del Producto Software" (Figura 2), en el que se identifican a todos los participantes en el proceso de certificación de la calidad del producto software:

1. Las empresas interesadas en evaluar, mejorar y certificar la calidad de sus productos software (propios o adquiridos), piedra central del ecosistema y sin las cuales el resto no tendría cabida.
2. AENOR, que como entidad auditora, se encarga de emitir la certificación de calidad del producto software.

3. AQC Lab, que como laboratorio acreditado de evaluación, dispone del entorno necesario para poder medir y evaluar el producto software y emitir un informe de evaluación siguiendo la norma ISO/IEC 25000.

4. Consultores de calidad software, que considerando la juventud de este tipo de evaluaciones, puedan dar soporte a las empresas a mejorar su producto software y alinearlo con las características de calidad para después poder ser certificado.

5. Empresas desarrolladoras de herramientas de medición, responsables de construir el software utilizado por los consultores de calidad y

empresas para medir y controlar la calidad del producto antes de presentarse a la certificación.

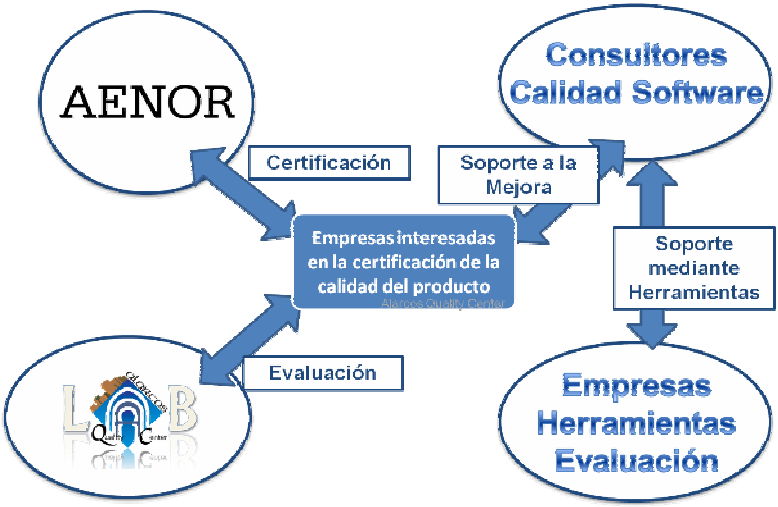


Figura 2. Ecosistema para la Evaluación y Certificación del Producto Software

2.5 CERTIFICACIÓN DEL PRODUCTO SOFTWARE

Una vez expuesta la importancia de la calidad del producto software y los principales estándares para su evaluación, es este apartado se presentarán las principales características del proceso para la certificación del producto software.

2.6 AQC LAB: LABORATORIO ACREDITADO PARA LA EVALUACIÓN DEL PRODUCTO SOFTWARE

Una de las necesidades que se identificaron inicialmente para poder evaluar la calidad del producto software, fue disponer de una entidad externa capaz de emitir una evaluación independiente sobre el producto software. Con esta idea, en 2009 comienza la construcción de AQC Lab, un laboratorio que basado en la ISO/IEC 25000 permita, tanto a empresas desarrolladoras de software como a entidades que externalizan o adquieren software, disponer de un informe independiente que refleje la calidad del producto software.

Con el objetivo de obtener un reconocimiento a la validez de las evaluaciones realizadas por AQC Lab, se decidió elaborar toda la infraestructura de gestión necesaria para conseguir la acreditación, siguiendo además para ello prácticas del desarrollo ágil como se expone en [11]. El resultado fue que en 2012 AQC Lab conseguía la acreditación de ENAC (Entidad Nacional de Acreditación) en la norma ISO/IEC 17025, como el primer laboratorio para la evaluación de la calidad de aplicaciones software bajo la familia de normas ISO/IEC 25000.

La acreditación de acuerdo a la Norma ISO/IEC 17025 confirma la competencia técnica del laboratorio y garantiza la fiabilidad en los resultados de los ensayos realizados.

El foco de la auditoría de acreditación fueron los tres elementos principales de AQC Lab utilizados durante la evaluación del producto:

- El Proceso de Evaluación, que adopta directamente la ISO/IEC 25040, y la completa con los roles concretos del laboratorio y los procedimientos de trabajo desarrollados.
- El Modelo de Calidad, que define las características y métricas para evaluar el producto software.
- El Entorno de Evaluación, que permite automatizar en gran medida las tareas de la evaluación.

2.6.1 MODELO DE CALIDAD DEL PRODUCTO

De entre las características de calidad propuestas por la ISO/IEC 25010, inicialmente se decidió centrar el modelo de calidad en la característica de la mantenibilidad, entendida como el grado de efectividad y eficiencia con el que un producto puede ser modificado, debido principalmente a las siguientes razones:

- El mantenimiento supone una de las fases del ciclo de vida de desarrollo más costosa, sino la más, llegando a alcanzar el 60%.
- La mantenibilidad es una de las características más demandadas hoy en día por los clientes de software, que piden que el producto software que se les desarrolle pueda ser después mantenido por ellos mismos o incluso por un tercero.
- Las tareas de mantenimiento sobre productos con poca mantenibilidad tienen más probabilidad de introducir nuevos errores en el producto.

El modelo de calidad definido para la mantenibilidad parte exactamente de las cinco subcaracterísticas de calidad definidas en la ISO/IEC 25010, que son:

- **Analizabilidad**. Se define como la facilidad para identificar las partes de un sistema que se deben modificar debido a deficiencias o fallos, o la capacidad de evaluar el impacto que puede provocar un cambio en el sistema.
- **Modularidad**. Se define como el grado, en el que un sistema se encuentra dividido en módulos de forma que el impacto que causa una modificación en un módulo sea mínimo para el resto.

- **Capacidad de ser Modificado**. Se define como el grado en el que se pueden realizar cambios en un producto software de forma efectiva y eficiente, sin introducir defectos ni degradar su rendimiento.

- **Capacidad de ser Reutilizado**. Se define como el grado en que un activo (módulo, paquete, clase, etc.) puede ser usado en más de un sistema o en la construcción de otros activos.

- **Capacidad de ser Probado**. Se define como la facilidad para establecer criterios de prueba para un sistema y realizar las pruebas que permitan comprobar que se cumplen esos criterios.

Sin embargo, la familia de normas ISO/IEC 25000 todavía no ha definido el conjunto de métricas e indicadores que afectan a cada una de estas subcaracterísticas, los umbrales para las mismas, ni las funciones de medición a aplicar para poder calcular el valor de calidad de cada una de ellas. Por ello, para completar este modelo de calidad y hacerlo operativo,

se han identificado un conjunto de propiedades de calidad, que obtienen su valor a partir de métricas del código fuente, y se ha establecido la relación que existe con las subcaracterísticas anteriormente indicadas. El objetivo al identificar estas propiedades de calidad y métricas no ha sido que fuera el conjunto mayor posible, sino que fuera un grupo completo y sin lugar a controversias, basado en los estudios e investigaciones previas y aceptados por la comunidad científica. Estas propiedades de calidad son:

- Incumplimiento de reglas de programación.
- Complejidad Ciclomática.
- Estructuración de paquetes y clases.
- Tamaño de unidades.
- Código duplicado.
- Documentación de código.
- Acoplamiento y Cohesión.
- Ciclos de dependencia.

2.6.2 ENTORNO DE EVALUACIÓN DEL PRODUCTO

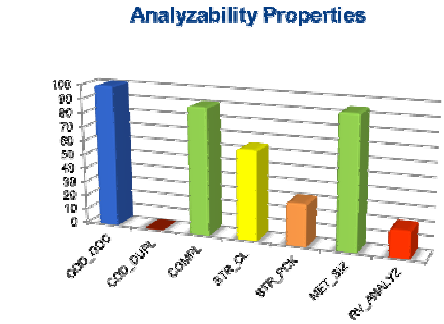
El entorno construido permite automatizar hasta en un 90% las evaluaciones y está formado por tres niveles diferenciados:

1. **Herramientas de medición**. Constituyen el primer nivel y su misión es analizar el código fuente y generar archivos con los datos sobre

métricas base. La ventaja de este nivel es que es fácilmente ampliable, añadiendo nuevas herramientas que permitan analizar nuevos lenguajes de programación o calcular nuevas métricas para otras características de calidad.

2. **Sistema de evaluación**. Supone el nivel intermedio del entorno y su objetivo es analizar el conjunto de archivos generados por el nivel inferior y aplicar los criterios de evaluación del modelo de calidad, obteniendo como resultado los valores para las propiedades, subcaracterísticas y características de calidad.

3. **Entorno de visualización**. Representa el nivel superior del entorno y permite presentar de manera comprensible la información obtenida tras la evaluación del producto software. Además de mostrar los valores de calidad para propiedades, subcaracterísticas y características de calidad (Figura 3), este entorno permite la generación de informes con el resultado de la evaluación.



Property	Acronym	Value
Code Documentation	COD_DOC	100.0
Code Duplication	COD_DUPL	0.0
Complexity	COMPL	80.0
Class Structuring	STR_CL	66.33
Package Structuring	STR_PCK	30.0
Method Size	MET_SIZ	80.33
Analyzability Rule Violations	RV_ANALYZ	16.33

Figura 3. Ejemplo de Resultados de la Evaluación del Producto Software

2.7 PROYECTO PILOTO DE CERTIFICACIÓN DEL PRODUCTO SOFTWARE

Una vez alcanzada la acreditación del laboratorio, se estableció un proceso de trabajo con AENOR (Asociación Española de Normalización y Certificación) que permitiera que los productos software una vez se hubieran evaluado y obtenido un nivel adecuado de calidad, pudieran también conseguir un certificado. Como resultado AENOR incluyó dentro de su modelo con normas ISO para TICs la nueva familia ISO/IEC 25000 [12] y creó un procedimiento que, a partir del informe del laboratorio acreditado y tras una auditoría, permitía certificar la calidad del producto software bajo estudio (Figura 4).

Figura 4. Ciclo de Evaluación y Certificación del Producto Software

Todo lo anterior permitió realizar a lo largo de 2013 un proyecto piloto de evaluación y certificación de los primeros productos software. Gracias a este piloto, tres empresas españolas pudieron evaluar, mejorar y finalmente certificar la calidad de sus productos software en base a la ISO/IEC 25000, tal y como se presenta en [13]. Entre los beneficios obtenidos, dichas empresas destacaron haber reducido hasta en un 75% las incidencias correctivas, hasta en un 40% el tamaño de sus productos y hasta en un 30% los tiempos en las tareas de mantenimiento.

Tras este piloto, son varios los productos que a nivel nacional e internacional ya se han evaluado siguiendo este nuevo esquema, estando algunos de ellos ya en fase de certificación.

Por otro lado, indicar que AENOR y AQC Lab siguen trabajando para ampliar el alcance de las evaluaciones y certificaciones del producto software, de manera que se puedan abordar poco a poco todas las características de calidad identificadas por la ISO/IEC 25000.

2.8 REFERENCIAS

1. Kitchenham, B. y S.L. Pfleeger, *Software Quality: The Elusive Target.* IEEE Software, 1996. **20**(1): p. 12-21.

2. Maibaum, T. y A. Wassyng, *A Product-Focused Approach to Software Certification.* Computer, 2008. **41**(2): p. 91-93.

3. Boehm, B.W., et al., *Characteristics of Software Quality*. 1978: North-Holland.

4. ISO, *ISO/IEC 9126, Software Product Evaluation–Quality Characteristics and Guidelines for their Use*. 2001, International Organization for Standardization.

5. ISO, *ISO/IEC 14598-5:1998 - Information technology -- Software product evaluation -- Part 5: Process for evaluators*. 1998, International Organization for Standardization: Ginebra.

6. Heitlager, I., T. Kuipers, y J. Visser, *A Practical Model for Measuring Maintainability*, in *Quality of Information and Communications Technology, 2007. QUATIC 2007.* 2007. p. 30-39.

7. ISO, *ISO/IEC 25010, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. 2011: Ginebra, Suiza.

8. ISO, *ISO/IEC 25040 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Evaluation process*. 2011: Ginebra, Suiza.

9. Rodríguez, M. y M. Piattini, *Systematic review of software product certification*, in *CISTI 2012: 7th Iberian Conference on Information Systems and Technologies*. 2012: Madrid. p. 631-636.

10. Kitchenham, B., *Guideline for performing Systematic Literature Reviews in Software Engineering. Version 2.3*. 2007, University of Keele (Software Engineering Group, School of Computer Science and Mathematics) and Durham (Department of Computer Science).

11. Verdugo, J., M. Rodríguez, y M. Piattinni, *Using Agile Methods to Implement a Laboratory for Software Product Quality Evaluation*, in *15th International Conference on Agile Software Development (XP 2014)*. 2014: Roma (Italia).

12. Fernández, C.M. y M. Piattini, *Modelo para el gobierno de las TIC basado en las normas ISO*, ed. AENOR. 2012, Madrid: AENOR.

13. Rodríguez, M., C.M. Fernández, y M. Piattini, *ISO/IEC 25000 Calidad del Producto Software.* AENOR. Revista de la Normalización y la Certificación, 2013(288): p. 30-35.

Capítulo

3

3. EL PAPEL HUMANO EN LA CALIDAD DEL SOFTWARE. INTRODUCCIÓN.

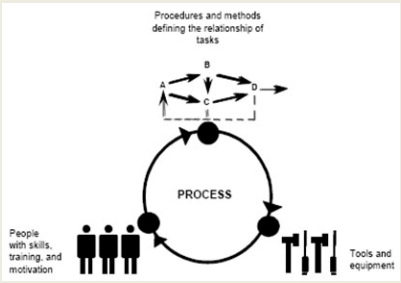
Autor: Domingo Gaitero

"No puedes volver atrás y hacer un nuevo comienzo. Pero puedes empezar de nuevo y hacer un nuevo final"

-- Diego Pablo Simeone

Tradicionalmente en las empresas que se dedican a las Tecnologías de la Información, y las comunicaciones, TIC, hemos crecido con el triángulo que hace años dibujó Dave Kitson del SEI cuando el CMM estaba todavía desarrollándose y que a día de hoy es mantenido en muchas organizaciones.

Este dibujo se centra en el aspecto de proceso de la triada, la tecnología y las personas son igualmente importantes (no existe ningún producto sin ellos).



Pero la tecnología cambia a su ritmo a lo largo del tiempo (veáse Redwine Y Riddle, 1985) y el factor humano se ha tratado por disciplinas como el desarrollo organizativo y la Gestión de la calidad Total (TQM) y por un modelo de madurez de los Recursos humanos, desarrollado también por el SEI, y conocido como People CMM.

En cuanto al tercer elemento de la triada, los problemas principales presentados por los productos software parecen ser, en gran parte, debidos a cuestiones del proceso.

Situaciones como entregas tardías de producto, sobrecostes, productos imperfectos, productos prometidos y nunca entregados, no fiables, etc., demasiado cotidianos desgraciadamente para nosotros, ha constituido el punto de mayor influencia en el triángulo durante casi tres décadas; es decir las empresas de servicios y por ende sus profesionales, nos hemos dedicado en cuerpo y alma ha escribir, reescribir, inventar, volver a inventar, pintar, dibujar y mil cosas más proceso, procedimientos y metodologías que aparte de acabar casi con la profesión de consultor informático, han creado un clima de desconfianza hacia esta técnica de manera casi insalvable.

Todo esto sumado a los ya conocidos cambios tecnológicos de versiones, o incluso de productos que el sector viene teniendo casi desde sus comienzos. Con un pequeño, gran matiz, además ahora los paradigmas como Internet, Cloud, redes sociales, mobile, Internet de las cosas y algunos más nos mantiene en un estado de cambio continuo sin posibilidad de respiro, donde hemos pasado de trabajar con un eje denominado Tecnología, a otro que podemos llamar "tendencias".

Este cambio, muy profundo, potente y rápido, ha sido caldo de cultivo espectacular para convivir con el "bug", con el parche, con el error. Ahora somos capaces de comprar productos que sabemos no funcionan en su totalidad, productos en beta tester, el cual el proveedor no sólo no oculta sino que presume, que es su insignia de servicio.

Un servicio en el que el usuario también ha cambiado, ahora es una persona con conocimiento, que no se deslumbra ante la tecnología sino que exige su uso y sus resultados. Un usuario globalizado que tiene una red demasiado extensa donde elegir proveedores y donde negociar precios muy inferiores a los expuestos os deseados por los profesionales.

Este panorama de "producto rápido" a "precios baratos", es más peligroso que el "bueno bonito y barato" de los años 80, y sobre todo deja en un lugar inmerecido al nombre de la "calidad". Es demasiado frecuente escuchar frase como "…es que hoy la palabra calidad no vende…", "...ya no se cree nadie lo de la calidad…".

Abrir un debate sobre esto ni siquiera merece la pena, cualquier empresa líder hoy en día en cualquier mercado, tiene a la calidad como un punto clave de su estrategia; y esas empresas, líderes de verdad, invierten

continuamente en mejorar no sólo sus productos y servicios sino la manera de hacerlos y su control sobre ese proceso.

Otra cosa diferente es la visión “cortoplacista”, o si me permiten “ignorante” que muchas empresas, y por lo tanto sus directivos tienen acerca de la calidad pensando simplemente en “normas”, “sellos” y “certificaciones”; hecho al cual hemos contribuido todos aunque de manera especial las grandes empresas y la administración, donde muchas veces “Calidad” es sinónimo de “requisito” para presentarse a una oferta.

Sin embargo no quiero “escurrir el bulto” y centrarme en lo que es el objeto de mi presentación y de este artículo en consecuencia; el eje de la triada denominado personas.

3.1 LAS PERSONAS O LOS PROFESIONALES

Desde mi humilde punto de vista, llevamos tiempo equivocándonos con las acciones que tomamos, es decir seguimos invirtiendo en los ejes de la tecnología, que consideramos indispensable, y en la de los Procesos que nunca terminan de implementarse al cien por cien.

Repetimos una y cien veces proyectos de reingeniería, de gestión del cambio, de transformación, o como viene siendo ahora la moda de Lean para tratar de conseguir que en nuestra organización se funcione de una manera estandarizada, homogénea, conocida y donde todos puedan colaborar y participar.

Y creo que esas excusas que solemos poner todos en nuestras presentaciones PowerPoint ya no son suficiente; ¿a cuales me refiero?, muy fácil:

- Colaboración por parte de la alta dirección
- Comunicación a todos los implicados.

¿A qué nos resultan conocidas?, por eso creo que debemos indagar un poco más en porque se puede producir este caos organizacional que no sólo significa grandes pérdidas económicas sino de productividad y credibilidad de las empresas, pero también de desánimo y esperanza en los trabajadores.

Les propongo la siguiente formula:

$$P * P = R$$

Donde la primera P es la “persona”; una persona a la que como hemos contratado para trabajar y que consideramos bien preparada, con estudios y capacitada (esto se llama expectativas), le encargamos una serie de trabajos que desarrolla en base a la segunda P, Procesos.

Por ejemplo a una persona (un jefe de proyecto) le hacemos responsable de la planificación del proyecto, de la planificación de las pruebas, de la gestión documental del proyecto y del seguimiento del mismo, es decir ejecutamos la operación 1*4 cuyo resultado (R) es 4.

Pero como necesitamos más le pedimos que se haga cargo del plan de calidad del proyecto y de la gestión de la configuración del mismo controlando las diferentes entregas que se lleven a cabo. 4+2= 6 y 6*1=7.

Pero es que el negocio avanza muy deprisa y le damos la responsabilidad de gestionar la relación con la factoría y además queremos que supervise los temas de seguridad del proyecto. 6+2=8, pero 8*1 empieza a ser algo parecido a un 4,5.

Finalmente le pedimos un pequeño esfuerzo y que se haga cargo de los SLA y que revise la planificación para ver si podemos reducir el tiempo de pruebas porque en desarrollo van mal de tiempo. 8+2=10 y lo peor de todo es que 10*1= 0. Nuestro jefe de proyecto ha muerto profesionalmente, bueno realmente no, ha sido peor, se ha convertido en un zombie para el resto del tiempo que trabaje en esa empresa.

Esta transformación le afecta profundamente, ya que este ex profesional, busca respuestas en ITIL, en CMMI, en PMBOOK y lejos de argumentar sólo salen quejas de su boca donde implora ayuda ya que lejos del proyecto en el que trabaja y la empresa que le paga, se centra en el pago de una hipoteca, en deudas personales, en bodas, relaciones de pareja, hijos y demás, es decir ahora es el momento en el que realmente aparece la "persona", hasta entonces existía el profesional.

Donde radica la diferencia en un axioma muy sencillo, la empresa ha invertido en que ese jefe de proyecto crezca en procesos, en conocimiento, pero no en habilidades.

Es decir si cuando los resultados eran igual a 4 en vez de invertir en gestión de la configuración y de las pruebas se hubiera invertido en habilidades de gestionar tiempo y de liderazgo por ejemplo, la operación había quedado en un 3*12 cuyo resultado "Extraordinario" hubiera sido 12.

Es decir, no sólo hay que invertir en los vértices Proceso y Tecnología que suman, sino en el de las personas que a diferencia del resto multiplica, y casi casi podía decir que en algunos casos, incluso potencia, ya que el bienestar y el futuro azul en vez del "gris zombie" hace sentirse feliz a ese trabajador que empieza a desarrollar su trabajo con Pasión y ahí es el momento donde la empresa gana de verdad.

Este ejemplo es válido absolutamente para casi cualquier proceso en la organización TIC, sin embargo el responsable de calidad tiene unas connotaciones muy especiales que vemos a continuación.

3.2 EL RESPONSABLE DE CALIDAD

Hace casi treinta años me tocó viajar con mi empresa a Italia, en concreto a Nápoles. Íbamos a conocer una empresa que vendía unos compiladores de COBOL que el banco para el que trabajamos iba a empezar a usar. Tranquilos no voy a hablar de COBOL ni de programación ahora sino de una visión que tuve entonces y que me ha acompañado hasta los días de hoy.

Me habían hablado mucho del tráfico en la ciudad, y la verdad caos se me queda corto para lo que allí viví realmente. Desorden, peligro, temeridad… se me ocurren muchos adjetivos que pensaba cada vez que tenía que cruzar una calle; sin embargo llegamos a una gran rotonda por el centro de la ciudad cerca de la Via dell'Epomeo y allí en medio de un gran atasco y un lío de coches como ni siquiera he visto en la India, vi a un policía de trafico perfectamente uniformado, de blanco, con unas condecoraciones incluso, haciendo gala de su silbato indicando a todos los coches hacia donde tenían que dirigirse.

Por el aspecto de su tez, era un señor maduro con experiencia en la materia, su forma de comportarse también destilaba caballerosidad; sin embargo nadie, absolutamente ningún conductor le hacía el más mínimo caso; y precisamente en ese momento me vino a la cabeza el nombre de Juan Ezcurra, el director de Calidad de mi empresa al cual igual que a ese pobre agente de tráfico nadie hacia ni el más mínimo caso; y eso que Juan ha sido para mí uno de los mejores profesionales que he conocido en toda mi carrera.

¿Qué le podía ocurrir a Juan entonces? ¿Qué fallaba hace casi 30 años y sigue fallando hoy en día?, porque lo que nadie puede negar es que la

figura del responsable, o incluso, del profesional que se dedica a la calidad se ha deteriorado de manera considerable en los últimos años.

Se ha perdido, si es que alguna vez se tuvo, la influencia directiva, ya casi todos los responsables de Calidad están fuera de los comités de dirección. Su producto suena a “rollo metodológico” a “más de lo mismo” incluso y con sin ánimo de ofender “algo casposo”.

Muchas veces su mensaje no se considera “aterrizado” y en definitiva han perdió credibilidad.

Aquí surge otra fórmula mucho más conocida por todos:

$$Profesional = (C + H) * A$$

Donde C es el conocimiento. Cuando contratamos a alguien vamos a su currículo rápidamente a ver sus carreras, sus master, sus títulos, que áreas de proceso conoce y cuáles son sus habilidades (H), su experiencia, su habilidad dirigiendo equipos, los proyectos que ha gestionado, etc.

Es decir que un licenciado, con un master, con dos idiomas, experto en gestión de la calidad, de proyectos, con las certificaciones PMbook, Auditor y certificador ITIL suma en la primera operación un 9 (conocimiento más habilidades). Es perfecto en definitiva.

Pero nos queda la A de actitud y que encima multiplica; es decir ese brillante ingeniero es un tipo triste que carece de empatía y encima quiere ser director por encima de todo; resultado 9*0 Igual a cero. A lo mejor como decimos ahora, en este mundo asquerosamente “políticamente

correcto", -ya, pero es buena persona-, lo que la menos nos permite multiplicar por uno.

Mi amigo Raúl Baltar dedica todo un capitulo en su magnífico libro "el arte del ser humano (en la empresa)", a este tema donde afirma de manera contundente "que la actitud siempre supone una diferencia" y lo deja muy claro al principio del mismo:" Si quieres mejorar, si quieres avanzar, entonces ¡pica adelante!

Hace unas líneas yo he hablado de Juan, un buen profesional, pero que carecía de actitudes necesarias para la gestión de la calidad, un profesional que aparecía cuando todo había fallado para recordarnos que deberíamos haber leído los procedimientos, un hombre que renovar el sello iba directamente a su bonus, por lo que era su prioridad, un hombre que pensaba que formar a gente que había hecho la carrera era un error porque les hacía perder tiempo, un director que cuando el proceso subía a la red para que todos los leyeran pensaba que ya estaba implementado.

Creo sinceramente, que los responsables de Calidad, y yo he sido uno de ellos, tenemos gran parte de culpa de esta situación, por eso quizás existen tantas personas desanimadas por no decir frustradas que deambulan, otra vez los zombies, por las empresas como simples mantenedores de renovar certificaciones que no es lo mismo que responsables de las normas aunque su cargo siga siendo el de responsable de Calidad.

Evidentemente estas personas, que poseen una gran conocimiento y sobre todo visión, han perdido la ilusión por su trabajo, han olvidado la pasión y por lo tanto no son felices, estancándose en una peligrosa zona de confort

que han reforzado con amargura y justificación, emitiendo discursos cargados de naftalina y buscando culpables por todas partes.

Aquí es donde debemos empezar a trabajar, y sobre todo a pensar de manera diferente, ya que si seguimos haciendo las cosas como venimos haciéndolo pues posiblemente consigamos lo que venimos consiguiendo.

La profesora de psicología de la universidad de California, Sonja Lyubomirsky, nos cuenta en su fascinante libro “Los mitos de la felicidad” que “los momentos críticos en nuestra vida lejos de ser aterradores o deprimentes, pueden ser oportunidades para renovarse, crecer o cambiar significativamente. Sin embargo lo que importa es la manera en los que uno los recibe”.

Las investigaciones recientes ponen al descubierto que las personas que han experimentado alguna “adversidad” son al final más felices que aquellos que no han sufrido ningún infortunio. Tener un historial de resistencia a diversos momentos devastadores “nos curte” y nos hace estar mejor preparados para manejar a posteriori los desafíos y traumas, grandes y pequeños. Además de fomentar la resistencia en general, los investigadores han demostrado que encontrarle sentido a los desafíos de la vida nos ayuda a definir y afianzar nuestras identidades, lo que a su vez apuntala el optimismo sobre nuestro futuro y nos permite manejar mejor las incesantes fuentes de preocupación.

Precisamente de situaciones negativas como la que nos afecta, nuestra manera de reaccionar puede ser el desencadenante de nuestra propia felicidad. Si nosotros seguimos quejándonos sobre el “ser” y no pasamos a la acción, “hacer”, vamos a caer en un pozo sin fondo; sin embargo

intentado hacer cosas nuevas, viendo cómo se pueden hacer de manera diferente encontramos ilusión y por lo tanto tenemos retos ante nosotros.

Para hacer esto herramientas como el coaching, PNL, Mentoring, liderazgo u otras más pueden ayudarnos mucho a elaborar nuestro propio itinerario que nos marque se nuevo horizonte y ese cambio de perspectiva del que tanto hablo.

¿Y qué características debe tener ese cambio? ¿Hacia dónde debe dirigirse?, bueno vamos a verlo a continuación.

3.3 QUALITY COACH. UN NUEVO ORDEN

El polifacético mago y humorista Jandro, conocido colaborador del programa de televisión "El hormiguero", nos indica en su delicioso y divertido libro "La oreja verde" que "equivocarse es habitual. Forma parte del proceso. Sin fallos no hay aciertos y sin fracasos no hay éxitos". Y curiosamente recomienda varias veces a lo largo de toda la obra que nos alejemos de la gente con yate. Si, con yate. Todos tenemos muchos conocidos, amigos y compañeros de trabajo que se pasan el día en el yate. No hacen otra cosa, les encanta y no se dan cuenta de lo destructivo que es. El yate al que me refiero no es una embarcación de lujo que surca los mares. No, es un barco más peligroso, surca las oficinas, los despachos y merodea cualquier intento de probar ideas nuevas. Es el yate común, más conocido como el "ya te lo dije". Si, "ya te lo dije" cuatro palabras malignas cuya combinación es mortífera.

La gente con estos "yates" tiene un ego grande y una capacidad de destrucción psicológica alta, son tóxicas y es mejor tenerlas lejos, su veneno

se va introduciendo dentro de nosotros poco a poco hasta conseguir que no intentemos nada nuevo porque ¿para qué?, total si va a salir mal… Pero esto no debe significar el final, simplemente una calle cortada por lo que tendremos que dar marcha atrás y salir por otra salida de la rotonda anterior.

Para ello debemos dotar a estos profesionales de la calidad de un nuevo baúl de habilidades y competencias, para conseguir de nuevo lograr la sostenibilidad, influyendo no sólo en lo que la organización "HACE de manera diferente", sino además en lo que la organización "ES".

Estos profesionales no deben solamente crear los procesos y la forma en que hacen las cosas, sino además transformar la cultura, los valores y la forma de pensar, liderar personas y equipos.

En resumen: "Pasar de la cultura del control y la exigencia a un paradigma de compromiso y sostenibilidad".

Tal y como sale reflejado en la anterior figura nuestras empresas han crecido con la necesidad de hacer los proyectos perfectos y a la primera (PERFECCION), por lo cual visto el historial de nuestro sector casi nunca se ha conseguido. ¿Y qué ocurre cuando no triunfas? Pues que las cosas salen mal (FRACASO), entonces vienen las lamentaciones. Cuando esto se produce la empresa o nuestro responsable nos advierte de que esto no puede ser, de que hay que hacerlo mejor, y que se va a poner "encima" del tema. Situación curiosa ya que los líderes deben ponerse "delante" cuando hay problemas y no "encima"; y empieza a controlarlo todo (CONTROL), lo que produce que el trabajador haga su función obligado sin ilusión (OBLIGACIÓN); todo esto además se produce en un estado de "Presión" que podemos denominar EXIGENCIA.

Tristemente casi todos hemos vivido alguna vez estas situaciones, pero hay otras, por ejemplo en vez de buscar es perfección a la primera podemos trabajar de manera incremental y mejorando poco a poco hasta conseguirlo (MEJORA CONTINUA), ese estado nos lleva a descubrir y aprender cosas nuevas (APRENDIZAJE); cuando estamos aprendiendo además de felices adquirimos confianza en nuestro sistema (CONFIANZA) lo que es básico para que nazca el compromiso (COMPROMISO) y renazca la pasión por el trabajo. Este estado conocido con el nombre de EXCELENCIA no es un paraíso, ni un mundo feliz como muchos se suponen; sino que debe ser un sitio donde haya "tensión" que haga que los profesionales estén atentos a cada oportunidad y a cada situación nueva.

Este modelo nos da carisma y nos produce empatía en las personas que están deseando nuevos retos y nuevos enfoques.

Este nuevo modelo debe conseguir que cambie la forma en que la organización percibe a los técnicos y responsables del área de calidad, que ellos mismos dejen de ser controladores para convertirse en facilitadores y líderes de la mejora continua.

Y que ese vértice de la triada: Procesos, deje de ser un fin a perseguir, para convertirse en un medio de apoyo para que las personas de la organización logren la mejora continua. Los procesos de calidad se diseñan con las personas que van a utilizarlos, y estos están al servicio de la organización y no la organización al servicio de los procesos.

Este cambio de perspectiva, que permita a los profesionales de la calidad en las empresas TIC ayudar de manera directa las direcciones de igual manera que sucede en otros sectores como el automovilístico o industrial, debe de producirse de una forma natural y bajo unos parámetros necesarios.

La Dra. Silvia Leal nos explica en su obra "Ingenio y Pasión" "que esta transformación (ella se dirige directamente a la innovación) no es fruto directo de la financiación, sino el resultado de la energía creadora de las personas. Para ello se basa en una sencilla metáfora, la innovación es el resultado de la combustión de tres elementos: las personas, la organización y la motivación. Si estos se gestionan con eficiencia el resultado será una potente energía creadora, pero si no se gestionan bien, el resultado será una simple incineración".

Y para ello crea un modelo de gestión en tres dimensiones: el método Innova 3DX.

Este método se considera un proceso esencialmente humano que debe ser gestionado a través de tres dimensiones:

- El ecosistema creativo
- El potencial innovador
- Y la pasión, motor biológico que nos impulsa a actuar.

Desde otra perspectiva, el condicionamiento del entorno, nuestra capacidad y bloqueos creativos y nuestra motivación por la Calidad.

Quiero hacer especial hincapié a la primera dimensión: el ecosistema creativo. Un entorno que levante barreras frente al comportamiento de Calidad inhibirá el esfuerzo de todos aquellos que tengan el talento y las habilidades para hacerlo. En estas organizaciones la chispa de la cultura de calidad se apagará. Eso debe llevar a las empresas a prepararse para identificar y eliminar los frenos a los que se enfrenta el proceso y, por supuesto, para poner en marcha los mecanismos necesarios para su aceleración.

En otras palabras, las empresas que aspiren a liderar este codiciado terreno deberán asegurarse de construir un ecosistema creativo capaz de desatar su fuerza innovadora y permitir desarrollar una cultura de calidad sostenible.

Para ello necesitaran monitorizar tres factores:

- La cultura corporativa (tecnológica, de innovación y Calidad)
- El clima o entorno laboral, y

- El estilo de liderazgo y gestión.

Ojo a estos puntos porque por ejemplo la cultura corporativa, normas, valores y forma de pensar, es un elemento con clara influencia sobre nuestro comportamiento profesional. Si es buena puede guiarnos, motivarnos e incluso apasionarnos, por el contrario puede convertirse en un verdadero lastre si es confusa o esta desorientada.

3.4 CONCLUSIONES

Para terminar me gustaría dejar claro que en ningún momento trato de faltar al respeto o menospreciar el trabajo de miles de profesionales de la Calidad en nuestro país, sino todo lo contrario quiero devolverle el prestigio y el valor que debe tener.

Yo mismo, en los últimos años de mi vida como profesional asalariado, era responsable de la calidad en una multinacional europea. Allí se llamaba Excelencia operacional, por aquello de que el nombre de Calidad “no vende”, llegado un momento fui despedido y tuve que comenzar una nueva vida, lo cual aprovecho para confirmar lo anteriormente expuesto, fantástica que me está permitiendo desarrollar mis propios paradigmas.

Esta función de Calidad en principio fue desempeñada por un grupo relativamente numeroso, para terminar siendo desarrollado por la persona que ejercía como secretaria de mi departamento. Por un lado, me alegro de no estar allí, pero como profesional de la Calidad me apena profundamente este hecho que encima, no es aislado.

Por lo que me reafirmo en que no es necesario crear un nuevo puesto o un nuevo proceso sino simplemente en hacer un cambio de observador, el responsable de Calidad debe ser alguien útil a la empresa, que les haga pensar, que trabaje en primera línea y que participe de manera activa con responsabilidad y autoridad dentro de los proyectos.

Para ello ese profesional además de cualificado debe estar altamente motivado, aquí sí que pega eso de que "la calidad es nuestra razón de ser". La Calidad debe ser el elemento diferenciador de nuestra empresa. Y como dijo Walt Disney "Entendamos la Calidad como el conjunto de acciones y actividades que consiguen que un cliente que se ha gastado el dinero en nosotros, lo vuelva a hacer".

Hace falta tiempo pero debemos ofrecer resultados ligados al negocio directamente, debemos ser capaces de "poner números al software" y cuando digo números no me refiero a métricas sino a dinero contante y sonante.

Una vez un discípulo del famoso director de orquesta Von Karajan, se quejó enérgicamente a él recriminándole que el siendo igual de bueno cobraba menos cuando dirigía un concierto. El gran Karajan le respondió: "a lo mejor lo que ocurre es que cuando a usted le contratan usted les habla de música, mientras que yo solo hablo de dinero".

Quizás debamos de dejar de ser tan "listos" y en vez de hablar tanto del PDCA y de usar siglas debamos hablar más de negocio y demostrar los que durante años hemos venido predicando solamente.

Y por último, renovarnos, aquí sí que todo ese movimiento Agile, o Lean nos puede ayudar mucho; siempre me resulta paradójico que herramientas y técnicas con más de cincuenta años nos resulten extraordinariamente nuevas hoy en día, quizás debemos de dejar de leer para empezar a actuar.

Martin Seligman en su obra “la auténtica felicidad” deja claro que “cuando en el trabajo hacemos los que se nos da bien y nos sentimos útiles, nuestras posibilidades de ser felices aumentan”. Así de sencillo y, tantas veces, así de difícil.

Aunque para terminar ya que estamos en tierras gallegas lo quiero hacer con una persona a la cual yo considero el verdadero embajador de la felicidad en España: Santiago Vázquez quien hace un par de años publicó una obra que en mi modesta opinión tardará tiempo en ser superada, y lo digo no sólo como profesional sino como persona afectada por su lectura; su mensaje hizo cambiar mi forma de enfrentarme al mundo y de ver las cosas.

El libro se titula: “La felicidad en el trabajo…y en la vida”; en él hay cientos de ideas página tras página; pero yo solo quiero utilizar literalmente un párrafo que espero os golpee de la misma forma que hizo conmigo:

“Si vamos a trabajar cuarenta o más años, ¿podemos renunciar e intentar ser felices en el trabajo?, ¿es posible plantearse ser felices en la vida sin ser felices en el trabajo?

3.5 REFERENCIAS

Dymond, K. *A Guide to the Cmm: Understanding the Capability Maturity Model for Software* . 1995. Process Inc US

Lyubomirsky , S. *Los mitos de la Felicidad*. 2014. Ed. Urano

Jandro. *La Oreja Verde*. 2013. Ed. Alienta

Baltar, R. *El arte de ser humano (en la empresa).* 2012. Ed. Algón

Leal, S, y Urrea, J. *Ingenio y Pasión*. 2013. Ed. LID.

Seligman, M. *La auténtica felicidad*. 2005. Ed B

Vázquez, S. *La felicidad en el trabajo…y en la vida*. 2012. Ed. Actualia.

Capítulo

4

4. BDD: UNIENDO NEGOCIO Y PRUEBAS TÉCNICAS PARA MEJORAR LA CALIDAD DEL SOFTWARE.

Autor: Enrique Sánchez

"Behaviour" is a more useful word than "test"

-- Dan North

4.1 EL GRAN PROBLEMA DEL SOFTWARE

Hay dos problemas fundamentales a la hora de empezar un desarrollo software:

1. Construir el software correcto
2. Construir el software correctamente

En los últimos años ha habido un importante avance en cómo realizar software de manera correcta. La aparición en los últimos años de conceptos como Continuous Integration (Fowler, 2009) y más recientemente Continuous Deployment (Jez Humble, 2006) junto a la aparición de técnicas para mejorar la calidad del software como TDD (Beck), XP (Beck, Extreme Programming Explained: Embrace Change, 1999); y la aparición de frameworks de desarrollo como Symfony para PHP, Spring para Java o Rails para Ruby, han hecho que sea más fácil construir el software correctamente simplificando el proceso y evitando la duplicidad del código y reduciendo la complejidad del mismo.

A pesar de ello, según Standish Group Report en el 2009 (StudyMode.com, 2009) el 68% de todos los proyectos software empezados en Estados Unidos en el año anterior habían fracasado, se habían cancelado o simplemente habían resultados insatisfactorios.

En su artículo, *Why did your project fail?* (Cerpa, 2009), los autores reflexionan sobre las causas más comunes que se han encontrado a la hora de definir un proyecto como "fallido". En la tabla podemos observar como las principales razones de fallo de un proyecto son problemas relacionados con las fechas de entrega, la sobreestimación y los riesgos no controlados dentro del proyecto. Además, dentro de los motivos que aparecen, podemos ver la importancia que tienen la definición por parte del cliente de unos requisitos no realistas o inadecuados y el cambio en el *scope* del proyecto durante la realización del mismo.

Software Project Failure Factors	Percentage of Projects (%)		
	In-house	Outsourced	Overall
Delivery date impacted the development process	93.9	90.5	92.9
Project under-estimated	83.7	76.2	81.4
Risks were not re-assessed, controlled, or managed through the project	73.4	80.9	75.7
Staff were not rewarded for working long hours	81.6	57.1	74.3
Delivery decision made without adequate requirements information	83.7	47.6	72.9
Staff had an unpleasant experience working on the project	83.7	47.6	72.9
Customers/Users not involved in making schedule estimates	69.4	76.2	71.4
Risk not incorporated into the project plan	65.3	80.9	70.0
Change control not monitored, nor dealt with effectively	63.3	85.7	70.0
Customer/Users had unrealistic expectations	69.4	66.7	68.6
Process did not have reviews at the end of each phase	75.5	47.6	67.1
Development Methodology was inappropriate for the project	71.4	52.4	65.7
Aggressive schedule affected team motivation	69.4	57.1	65.7
Scope changed during the project	67.3	57.1	64.3
Schedule had a negative effect on team member's life	71.4	42.9	62.9
Project had inadequate staff to meet the schedule	63.3	57.1	61.4
Staff added late to meet an aggressive schedule	61.2	61.9	61.4
Customers/Users did not make adequate time available for requirements gathering	61.2	57.1	60.0

Table 1. Porcentaje de errores en proyectos software

Todos estos problemas tienen un mismo origen: la falta de comunicación y la separación que existe entre la parte de negocio y el equipo de desarrollo.

4.2 LENGUAJE UBICUO

Como Eric Evans describe en su libro Domain Driven Design [13]:

> *"Un proyecto tiene problemas muy serios cuando sufre de fracturación en el lenguaje. Los expertos de negocio usan su propia jerga mientras que el equipo técnico tiene su propio lenguaje en el que entienden el negocio en términos de diseño... Por esta división lingüística, los expertos de negocio apenas saben describir de una manera vaga qué es lo que quieren y los desarrolladores son forzados a adoptar un lenguaje nuevo para ellos que apenas llegan a entender."*

Evans habla de la dificultad que aparece entre los miembros de un equipo para comunicarse debido a la separación que existe entre los puntos de vista de cada uno de los implicados. Una mala comunicación provoca malentendidos que hacen que el equipo pierda tiempo en reuniones poco claras reinterpretando requisitos; hace que aparezcan defectos que provocan retrasos en la entrega y frustración en el equipo lo que hace que la planificación empeore y se genere un efecto de "ventana rota" muy peligrosa para el proyecto.

Cuanto más grande es esta separación, más difícil es conseguir detectar los grandes problemas que mencionábamos en el punto anterior y más complicado se vuelve su resolución.

Analizando esta separación podemos observar cómo la mayoría de los errores se produzcan en la primera fase del proyecto: en la toma de requisitos. Tavolato y Vincena, citando a Tom DeMarco, comprobaron

que el 56% de todos los bugs se gestan dentro de la etapa de requisitos (Tavolato, 1984) provocados por una mala comunicación entre ambas partes del proyecto. Si analizamos esto en función de la curva de coste de un cambio respecto al tiempo podemos observar como corregir estos errores en las primeras etapas de un proyecto es mucho más barato que hacerlo en las últimas: darse cuenta de que una funcionalidad es poco útil o tiene un coste muy grande en términos de desarrollo, es más sencillo y barato de corregir en las etapas de requisitos y diseño que cuando la funcionalidad ya ha sido implementada.

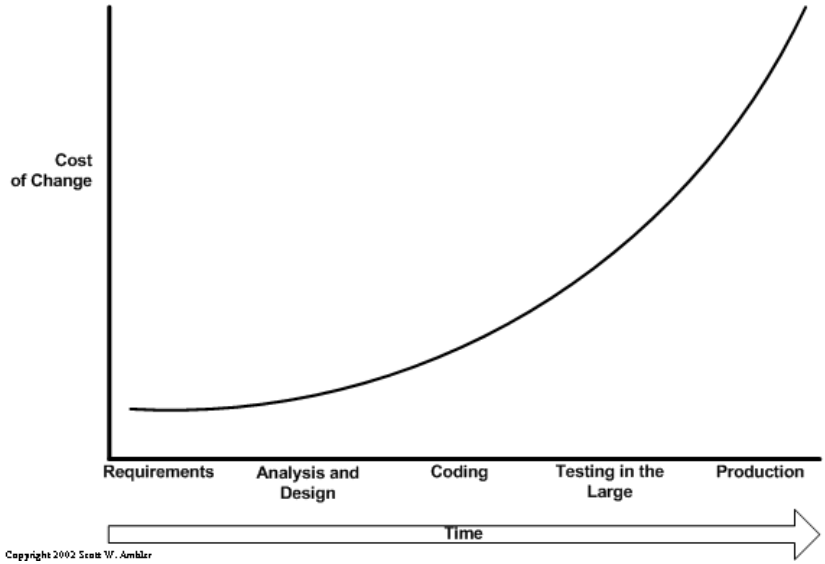


Figure 1. Coste del cambio respecto al tiempo

Para evitar esto el equipo puede llevar a cabo un esfuerzo y adoptar un lenguaje "*ubicuo*" que pueda ser entendido por todo el mundo implicado en el proyecto. De esta manera, cuando el equipo utiliza este lenguaje de

manera natural en sus conversaciones, en la documentación y en el código, la brecha que separa a las diferentes partes se minimiza y las posibilidades de malentendidos se reducen.

4.3 BDD

En el año 2006 Dan North publica un artículo en la revista Better Software (North, 2006) explicando los problemas que tenía a la hora de empezar un proyecto. North trazaba un paralelismo con las técnicas de TDD y decía que le gustaría saber antes de empezar a programar por dónde debería de empezar, cómo debería de probar y cómo podría saber si lo que estaba probando era lo correcto o no. Para conseguir esto el primer paso, según North, era que había que olvidarse del concepto "probar" y que había que empezar a pensar en el concepto ‘‘comportamiento" ya que de esta forma se pone el foco en cómo el software debería de actuar sin pararse a pensar en cómo implementarlo. El autor defiende que hay que hablar de un lenguaje común a todos los stakeholders del proyecto que todos pudieran entender y que permitiera detectar los problemas mucho antes de lo que se estaban detectando.

A esta técnica North la llamó *Behavior Driven Development* (BDD) y la definió de la siguiente forma

‘‘BDD es una metodología ágil de segunda generación, *outside-in, pull-based con múltiples stakeholders implicados, multiescala y altamente automatizada. Describe un ciclo de interacciones con unas salidas bien*

definidas que concluyen en la entrega de un software que funciona, bien probado y válido.'' (North, Introducing BDD, 2006)

Esto se resume en definir el software en términos del comportamiento deseado utilizando un lenguaje *"Outside-In"* (desde negocio a desarrollo) y centrado en definir unos elementos comunes que todo el mundo en el proyecto puedan comprender sin dificultad y que haga que el desarrollo del software sea más sencillo y eficiente.

4.4 GHERKIN

Una vez definido el proceso que se va a seguir, lo siguiente a definir es el cómo ha de ser este proceso. Para ello, BDD define una forma de especificación con un formato muy sencillo y que es compartido por la especificación de las historias de usuario: la narrativa.

Una narrativa sirve para definir el rol al que ese dirige la historia, cuál es el efecto que busca que tenga la historia en el rol y cuál será el valor de negocio que la historia aportará. De esta manera se consigue situar a todos los stakeholders del proyecto en el mismo plano y entender qué es lo que se persigue por cada una de las historias evitando requerimientos inútiles o contradictorios.

A partir de una narrativa es necesario definir los escenarios que la cubren en forma de ejemplos sencillos que describan las historias de usuario que se desarrollarán. En principio, no existe un requisito formal de cómo una historia debe de ser escrita y se deja a los stakeholders que

eligieran el suyo propio. Para hacerlo más sencillo existe una plantilla muy sencilla por la que se puede empezar:

Título: claro y sencillo, lo más descriptivo posible.

Narrativa: una pequeña sección introductoria que especifica lo que se quiere realizar, qué actor será quien lo realice y qué valor de negocio derivará de la historia.

Escenarios: Una descripción de cada caso específico que define la narrativa. Deben de tener la siguiente estructura:

- ***Descripción*** de la situación inicial en la que se encuentra el actor. Esto puede considerarse un paso único o varios pasos dependiendo del escenario que se quiera probar.
- ***Acción*** que realiza el usuario y que provoca el inicio del escenario.
- ***Estado*** final esperado.

Para simplificar más la escritura de narrativas, BDD utiliza un lenguaje semi-formal llamado Gherkin que permite describir el comportamiento del software sin entrar en el detalle de cómo el comportamiento es implementado. Fue definida por North en el 2007 (North, dannorth.net, 2007) para que pudiera ser adoptado por las diferentes herramientas software de BDD que estaban empezando a desarrollarse. Se basa en el concepto de *Business Readable DSL* y define una pequeña gramática forma con muy pocos elementos pero muy claros en cuánto a uso.

Story: Nombre de la historia

In order to definir un objetivo a realizarse

As a rol/actor implicado en la historia

I want to descripción de lo que se quiere obtener

Scenario: Nombre del escenario

Given un estado previo en el que se encuentra el actor

When el usuario realiza una acción o interacción

Then comprobamos que se ha producido un resultado correcto

La única sintaxis que tiene este DSL[1] es la que aparece en negrita y que sirven para marcar un punto de inicio común para todas las historias de usuario.

Para redactar un escenario Ben indica en su artículo *"Imperative vs. Declarative Scenarios in user stories"* (Mabey, 2008) que la forma de redactar debe de ser mediante el uso del declarativo en lugar del imperativo y pensar en un lenguaje de negocio evitando el uso de todos los elementos técnicos de manera que todo el mundo pueda entender

[1] Domain-Specific Language: http://martinfowler.com/dsl.html

cuál es el objetivo del escenario y haciéndolo independiente de las tecnologías o de la codificación.

4.5 CUCUMBER

Cucumber es una herramienta para ejecutar test automáticos en estilo BDD y escrito en Ruby. Cucumber ayuda a la tarea de definir un lenguaje ubicuo proporcionando un terreno en el que negocio y desarrollo puedan encontrarse.

Los tests de Cucumber interaccionan directamente con el código desarrollado, pero son escritos en un lenguaje que los stakeholders de negocio puedan entender. Esto permite la colaboración en la definición de estos tests y permitir un feedback rápido por parte de todo el mundo involucrado en el proyecto. Al escribir estos test antes de empezar el desarrollo podemos explorar y erradicar los malentendidos antes de que estos lleguen a producirse o a codificarse dentro del código.

4.5.1 ¿CÓMO FUNCIONA?

Cucumber es una herramienta de línea de comandos. Cuando se ejecuta busca archivos de texto llamados "*features*" y trata de leer las especificaciones buscando escenarios que probar.

Cada uno de los escenarios es una lista de "*steps*" que se traducen desde *Gherkin* a un conjunto de funciones en Ruby que delegan en una librería que ejecutará las acciones contra el sistema a probar. Esto suele implicar el utilizar una librería de automatización como librerías de *Browser*

Automation (Capybara, Watir...), *Mobile Automation* (Calabash) o interactuar directamente con el sistema a probar. Podemos ver un diagrama del funcionamiento de Cucumber en la siguiente imagen

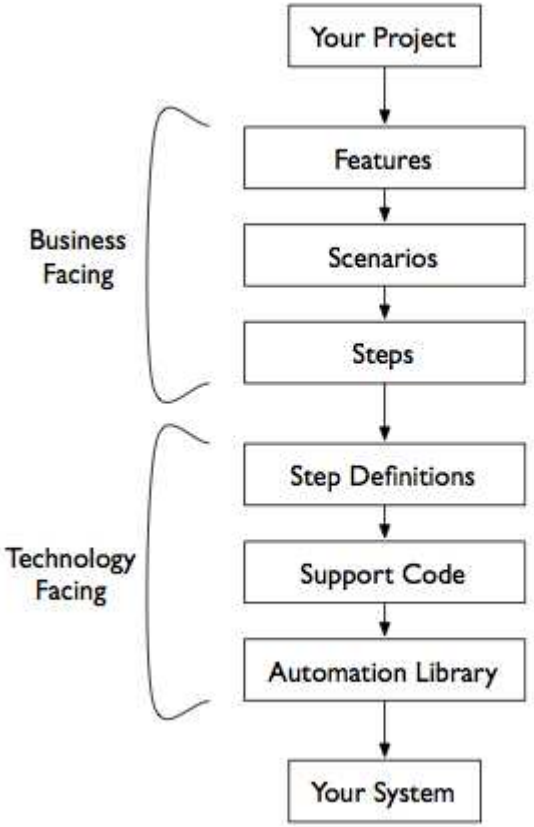


Figure 2. Funcionamiento de Cucumber

4.6 ¿CÓMO IMPLEMENTAR UN PROCESO BDD?

Una vez que hemos hablado de la base de BDD es hora de pensar en cómo llevarla a cabo dentro de un proyecto.

4.6.1 DEFINIR LOS ESCENARIOS A DESARROLLAR

Es la parte fundamental del proceso de BDD ya que en este caso saldrán los escenarios que guiarán todo el desarrollo del producto y serán los que generen el lenguaje común que se usará en el proyecto.

Los Acceptance Criteria (AC) deben de definirse antes de cada sprint y deben de definir de forma clara e inequívoca cuales serán las historias de usuario que servirán para dar por válido un sprint, por tanto, debe de implicar a todos los stakeholders del proyecto.

La definición debe de realizarse utilizando el lenguaje Gherkin y al final de la sesión debe de existir uno o varios archivos "features" que incluyan las features y los escenarios que deberán de ejecutarse en ese sprint. Estos archivos no deberían de modificarse a lo largo del desarrollo salvo para una refactorización o una simple aclaración o corrección de los parámetros.

4.6.2 CREAR TESTS AUTOMÁTICOS QUE VALIDEN LA FUNCIONALIDAD

Una vez que ya se han definido las features el siguiente paso es definir una serie de tests automáticos que validen estos escenarios. Esto ha de realizarse antes de empezar la codificación y debe de seguir los principios de TDD:

Escribir un test -> Ver como todos los test fallan -> Escribir código -> Reescribir el test

4.6.3 CREAR UN ENTORNO DE TRABAJO

Casi tan importante como tener crear test automáticos que validen el producto es tener un entorno que permita que estos test se ejecuten de manera correcta, eficiente y eficaz. Si un test existe pero no se ejecuta nunca, es inestable por culpa del entorno o es excesivamente lento, no se utilizará y el esfuerzo por crear un proceso eficiente no servirá de nada.

Para crear un entorno de trabajo eficiente hay que pensar en dónde se ejecutarán los tests (en una máquina local, en una máquina de integración continua...) y con qué frecuencia se ejecutarán (se ejecutará en cada commit, de manera nocturna, cada cierto tiempo...). La idea que debe de primar es que estos tests serán los que digan si el sprint está terminado y es válido por lo que los desarrolladores han de poder ejecutarlos cuando quieran y que el resultado que proporcionen debe de ser real y agnóstico de dónde lo estés ejecutando. Para ayudar con esto existen herramientas como Vagrant o Docker que permiten definir máquinas virtuales de manera sencilla y ligera que permiten al desarrollador tener un entorno "de producción" en su propio ordenador.

4.6.4 EVOLUCIÓN Y ADAPTACIÓN

Siguiendo los principios que Beck define en el Agile Manifesto (Beck, Manifesto for Agile Software Development, 2001) hay que poner a las personas antes que a las herramientas. Es importante analizar cómo se adapta el equipo a este proceso e ir adaptándolo según vaya

evolucionando el proyecto para cubrir las necesidades y las peculiaridades de cada equipo.

Es importante que las retrospectivas de cada sprint sirvan para detectar cuales han sido los errores y las mejores que han de producirse en el siguiente sprint y han de servir para ir definiendo mejor cómo han de escribirse los escenarios, qué han de cubrir y qué trabajo ha de mejorarse.

4.7 CONCLUSIONES

Hemos visto como BDD se adapta muy bien a las metodologías ágiles, ayudando a los equipos que lo practican a mejorar la comunicación entre las diferentes partes involucradas, definiendo para ello, un lenguaje ubicuo llamado Gherkin.

La definición de este lenguaje, junto a la creación de test automáticos antes de la codificación, hace que la codificación se más sencilla ya que el equipo sabe con ejemplos qué hay que hacer y qué se espera de ellos.

Además, simplificamos el proceso de desarrollo aumentando el feedback sin tener que añadir carga de reuniones y conseguimos tener una documentación constantemente actualizada sin necesidad de tener que realizar un esfuerzo extra.

En resumen, BDD se puede aplicar en cualquier proceso reduciendo costes operativos y mejorando la implicación de la gente en el proyecto.

4.8 REFERENCIAS

Beck, K. (1999). *Extreme Programming Explained: Embrace Change.* Addison-Wesley.

Beck, K. (2001). *Manifesto for Agile Software Development.* Agile Alliance.

Beck, K. (s.f.). *Test-Driven Development by Example.* 2003: Addison Wesley - Vaseem.

Cerpa, N. a. (2009). Why Did Your Project Fail? *Commun. ACM*, 130-134.

Evans, E. (2004). *Domain-Driven Design — Tackling Complexity in the Heart of Software.* Addison-Wesley.

Fowler, M. (2009). Continuous Integration. *Practices*.

Jez Humble, C. R. (2006). The Deployment Production Line. *Proceedings of Agile*.

Mabey, B. (19 de May de 2008). *benmabey.com.* Obtenido de http://benmabey.com/2008/05/19/imperative-vs-declarative-scenarios-in-user-stories.html

North, D. (March de 2006). Introducing BDD. *Better Software*.

North, D. (17 de Jun de 2007). *dannorth.net.* Obtenido de Introducing rbehave: http://dannorth.net/2007/06/17/introducing-rbehave/

StudyMode.com. (2009). *Chaos Report.*

Tavolato, P. a. (1984). A Prototyping Methodology and Its Tool. *In Approaches to Prototyping*.

Capítulo

5

5. LA CALIDAD DEL PRODUCTO DESDE LAS PRUEBAS Y EN RENDIMIENTO.

Autor: Pedro Sebastián Mingo

"Las pruebas de rendimiento anticipan el comportamiento del software en un entorno real"

-- Pedro Sebastián Mingo

5.1 INTRODUCCIÓN

En este capítulo está descrita una aproximación a las pruebas de rendimiento de software desde una perspectiva organizativa y utilitarista: describimos los elementos necesarios para realizar pruebas de rendimiento y proponemos unos ejemplos que ilustran cómo se aplican para su buen entendimiento y aplicación. La idea es que cualquiera que conozca el aporte de valor que proporcionan las pruebas sea capaz de identificar las situaciones en las que las pruebas de rendimiento forman parte de la solución.

5.2 ELEMENTOS BÁSICOS

El primer elemento en que ahondará este capítulo es la SIMULACIÓN que es la forma que tienen las pruebas de rendimiento, es decir, en qué hay que pensar para diseñar unas buenas pruebas. A partir de ahí salen el resto de los elementos, que también están descritos en cada una de las secciones: HERRAMIENTAS para llevar las pruebas a cabo, OBJETIVO con el que se harán las pruebas y ENTORNO en el que se van a efectuar. Con todos estos elementos podemos articular y comprender unas pruebas de rendimiento que estén a la altura de las expectativas.

5.3 LAS PRUEBAS SON SIMULACIÓN

Lo primero que hay que tener en cuenta a la hora de planificar pruebas de rendimiento es que las pruebas nos van a permitir conocer el comportamiento del software en condiciones de uso. Estas pruebas serán clave para recabar la información necesaria para tomar decisiones sobre la arquitectura software y hardware de la aplicación.

Por lo tanto en el origen de las pruebas está el análisis de las condiciones en las que el software prestará servicio: cuantos usuarios la utilizarán, con qué frecuencia, qué funcionalidades serán más costosas desde el punto de vista de los recursos del sistema, con qué infraestructura se cuenta, qué comunicaciones estarán disponibles, etc.

En resumen, se trata de crear un escenario de pruebas que refleje las características significativas de la realidad, del día a día de software en producción. Las pruebas deben simular las situaciones a las que se puede

tener que enfrentar el sistema: picos de trabajo, funcionalidades que consumen muchos recursos, caída de elementos de infraestructura, conocer el tiempo de recuperación del sistema en condiciones de demanda, etc. El número de escenarios es infinito y responde a las necesidades de conocimiento que tengan los equipos de desarrollo de las aplicaciones.

5.4 NECESITAMOS HERRAMIENTAS

El punto de partida de la simulación de escenarios a los que va a tener que hacer frente la aplicación nos define el primer reto: ¿Cómo hago para simular la carga de trabajo que 100 usuarios generan en el sistema? Obviamente no es fácil emplear usuarios reales para hacer las pruebas, así que la respuesta a este reto nos la tiene que dar la tecnología: Utilizando herramientas que generen concurrencia y carga de trabajo en función de las necesidades del escenario.

Las herramientas que necesitamos deben cumplir con las siguientes características: posibilidad de parametrización de la funcionalidad del software a probar (uso de datos diferentes y usuarios diferentes), ejecutar dicha funcionalidad de manera concurrente para generar carga de trabajo (múltiples hilos de ejecución), configurar el escenario para simular diferentes perfiles de ejecución (picos y valles de trabajo así como carga incremental sostenida) y, por último, debe ser capaz de recolectar los resultados de las pruebas para su posterior análisis.

No es el objetivo de este texto recomendar ni señalar ninguna herramienta en particular, ya que la elección depende del contexto de uso, de las necesidades y del presupuesto disponible. Se trata simplemente identificar este elemento, justificar su necesidad y glosar sus características. Lo que sí

es de utilidad y conviene remarcar es que es posible tanto utilizar una sola herramienta que cubra de extremo a extremo las necesidades del equipo, como una combinación de herramientas especializadas en distintas funcionalidades (herramientas de scripting, de monitorización, etc.).

5.5 DEFINIMOS OBJETIVOS

Resulta evidente señalar que para la realización de las pruebas de rendimiento, así como de cualquier otro tipo de prueba, debemos marcar un objetivo, ¿qué pretendemos conseguir? ¿Qué queremos demostrar? ¿Qué resultado nos va a validar la prueba? Normalmente las respuestas a estos objetivos están en los requisitos de la aplicación.

Si pensamos que los requisitos son la descripción funcional y técnica de la aplicación, la base sobre la que se construye el software, es lógico llegar a la conclusión de que los requisitos deberán ser nuestra guía para diseñar las pruebas y para validar sus resultados. No obstante, incluso en el mejor escenario posible, es frecuente encontrarse con requisitos imprecisos desde el punto de vista de rendimiento.

Desde aquí me permito recomendar a lector que piense en la simulación como la prueba que hay que hacer para que el objetivo sea el resultado mínimo que debemos obtener. Algo así como el nivel que la aplicación tiene que superar para considerar la prueba como válida. Pretendemos simular una situación y para ello hay que ejecutar 3 funcionalidades con un número concreto de usuarios y durante un periodo de tiempo, el objetivo sería verificar si la aplicación responde siempre, incluso en el momento de máxima carga, en menos de 1 segundo. Necesitamos saber qué hay que

hacer y qué resultado indicará que el resultado de la prueba ha sido satisfactorio.

Un ejemplo para ilustrar esta recomendación: la simulación sería el deporte que tiene que practicar la aplicación y el objetivo podría ser la marca necesaria, algo como salto de altura superando los 2,20 metros.

5.6 CONSTRUIMOS EL ENTORNO

Hemos empezado planificando las pruebas como una simulación de una situación real, hemos llegado a la conclusión que para llevar a cabo la simulación necesitamos herramientas y objetivos y nos falta el último elemento: El entorno.

El entorno es el laboratorio donde vamos a ejecutar las pruebas, así que deberá reunir una serie de características definidas por las pruebas que se pretenden llevar a cabo. Al igual que sucede con los objetivos, la caracterización del entorno tiene que ser conocida y muy precisa para que los resultados obtenidos sean concluyentes.

Lógicamente podemos partir de la clasificación clásica de los entornos: Desarrollo, Integración, Preproducción y Producción, pero hay que ser conscientes de que esta es una clasificación limitada al uso que se la da al entorno. Cuando hablamos de entorno como elemento básico se trata de algo más profundo, más preciso: hablamos de la funcionalidad del entorno y también de los componentes de la infraestructura, de los controladores de dispositivos, de las comunicaciones, de topologías de red,… en resumen, hablamos de las características técnicas que pueden definir el tipo de simulación que queremos llevar a cabo.

Volviendo a la metáfora deportiva usada en el punto anterior, el salto de altura de 2,20 metros, podríamos considerar al entorno como la pista en la que se va a producir ese salto y definir una pista de tierra, de tartán, en altitud, con lluvia, con viento… Todas estas condiciones individualmente o combinadas hacen que la prueba sea diferente.

5.7 EXPLOTANDO LAS PRUEBAS DE RENDIMIENTO

Los elementos anteriormente descritos se combinan de distintas formas y resultan en distintas pruebas. Como es lógico todo este conjunto de pruebas trabajan al servicio de la calidad del producto software ya que cuanta más simulaciones hayamos previsto y ejecutado, mayor información tendremos del comportamiento de la aplicación y mejor preparado estará el software para hacerle frente.

En este punto vamos a detallar algunos ejemplos típicos de pruebas que se pueden realizar, cada una con su propia combinación de elementos. Como se ha mencionado anteriormente, el número de combinaciones y de escenarios posibles es prácticamente infinito, así que los ejemplos aquí expuestos pretenden servir de puerta de entrada para que el lector sea capaz de adaptarlos a sus circunstancias y necesidades. Pensando a más largo plazo, incluso sería posible que el lector fuese capaz de diseñar sus propias pruebas de rendimiento basándose en los cuatro elementos (simulación, herramientas, objetivos y entornos) y en los ejemplos descritos a continuación.

5.8 PRUEBAS DE COMPONENTES

Entorno. Desarrollo

Objetivo. Verificar el punto de ruptura que cada componente externo (drivers y otros elementos) tiene cuando la carga de trabajo aumenta sin límite. Se alcanza un punto de ruptura (el componente deja de responder) o el tiempo de respuesta supera un límite preestablecido.

Herramientas. Las definidas por el product-owner.

Simulación. El componente está instalado y recibe peticiones dentro del marco del sistema.

Descripción. Este tipo de pruebas se lleva a cabo sobre cada componente de manera aislada antes de integrarlo en la infraestructura para identificar y catalogar el rendimiento de cada uno. La idea es aproximarse a los límites de cada uno de estos componentes para conocer los puntos débiles y la capacidad máxima que podrá afrontar el sistema.

Este tipo de pruebas se puede hacer también con el objetivo de verificar que la infraestructura base del sistema está bien diseñada o si necesita algún tipo de ampliación o refuerzo en alguna de las partes que la componen.

5.9 PRUEBAS DE PROFILING

Entorno. Puede ser el de Desarrollo o incluso el de Integración. Es importante que el módulo esté aislado y trabajando en las condiciones más asépticas posible.

Objetivo. Son unas pruebas parecidas a las anteriores ya que su objetivo es someter a la máxima exigencia a los módulos construidos por el equipo de desarrollo en un entorno aislado para verificar su diseño funcional y su estructura interna. En condiciones normales optimiza la construcción mejorando el uso de recursos (memoria, disco, acceso a datos, etc.) y los tiempos de respuesta.

Herramientas. Las definidas por el product-owner.

Simulación. Queremos simular la máxima carga que soporta el módulo para verificar si su diseño es el que proporciona un mayor rendimiento. Este tipo de pruebas también ofrece la posibilidad de enfrentar dos construcciones diferentes para elegir la que mejor cumpla con el objetivo de rendimiento.

Las pruebas de profiling y las pruebas de componentes se pueden aplicar en el caso de migraciones y mantenimientos para asegurar que el nuevo componente o módulo no degrada el rendimiento del anterior antes incluso de integrarlo.

Un tema importante a tener en cuenta en este caso, es que el tamaño de los módulos lo debe determinar el contexto: puede haber módulos muy pequeños que lancen transacciones importantes que tienen que ser

probadas y módulos más grandes que realicen tareas de soporte en los que el rendimiento no sea un factor crítico.

5.10 PRUEBAS DE DIMENSIONAMIENTO

Entorno. Se puede usar un entorno de Integración siempre que disponga de todas las conexiones con sistemas externos y que tenga una composición escalable. Si no cumpliera con estas características el entorno más adecuado sería el de Preproducción.

Objetivo. El objetivo de estas pruebas es establecer cuál es el tamaño de la infraestructura que soportará al software de la aplicación. Lógicamente, además del tamaño, este tipo de pruebas debería servir para probar las configuraciones de cada elemento de la infraestructura (componentes) o la arquitectura que se debe adoptar para optimizar la respuesta de la aplicación (servidores en cluster, balanceadores, red de backup, etc.).

Herramientas. Las definidas por el product-owner.

Simulación. En este tipo de prueba se trata de simular diferentes circunstancias a las que tendrá que hacer frente la aplicación como conjunto. El planteamiento estándar sería hacer pruebas de carga con varios niveles de carga, pruebas de volumen y pruebas de estrés buscando el punto en el que la infraestructura deja de absorber carga de trabajo.

Este tipo de pruebas está especialmente recomendadas en despliegues de aplicaciones nuevas ya sean tipo bigbang (toda la aplicación en una sola fase) o en distintas fases (por país, por calendario, etc.) ya que permiten conocer la infraestructura mínima incluso sin datos anteriores. Además en

despliegues por fases optimizan la infraestructura necesaria para cada fase del despligue.

5.11 PRUEBAS DE RESILIENCIA

Entorno. Preproducción o Integración. Requieren de un entorno lo más completo posible ya que vamos a jugar con la disponibilidad de la infraestructura y es necesario disponer de elementos reales o muy similares a los reales.

Objetivo. Conocer el comportamiento de la aplicación en condiciones de pérdida de alguno de los elementos de la infraestructura. Se trata de verificar que se puede seguir prestando servicio y en qué condiciones, si algún elemento se cae, deja de prestar servicio o hay que reiniciarlo.

Herramientas. Las definidas por el product-owner.

Simulación. En las pruebas de resiliencia se pone a funcionar el sistema en condiciones normales y, cuando alcanza un determinado nivel de carga de trabajo se corta o se limita el servicio de alguno de los componentes: eliminar un servidor del cluster, tirar la base de datos, ralentizar las respuestas, forzar eliminación de caché, etc.

La idea es probar que el sistema sigue operativo dentro de los parámetros esperados en condiciones de carencia o restricción de servicio por circunstancias más o menos fortuitas.

Resulta más que evidente señalar que este tipo de pruebas son especialmente útiles en sistemas de alta disponibilidad, comercio electrónico, seguridad, redundancia, etc.

5.12 PRUEBAS DE EVOLUCIÓN

Entorno. Producción.

Objetivo. Este tipo de pruebas verifican que la aplicación no tiene problemas de rendimiento tiempo después de salir a producción. El rendimiento de un sistema de información una vez que sale a producción, se puede deteriorar por un aumento en el número de usuarios, por una mala gestión del almacenamiento, por deterioro en las estructuras de almacenamiento. El objetivo de estas pruebas es comprobar cómo ha evolucionado el sistema, en qué nivel de servicio está después de salir a producción. En mi opinión hay dos beneficios directos:

1. Toma de decisiones proactiva. En un formato gráfico esta línea hace posible la predicción a medio y largo plazo de ampliaciones de capacidad del sistema. Incluso si el algún momento el deterioro el mayor del esperado se pueden establecer niveles de alerta para activar protocolos de actuación.
2. Ajustar el coste de infraestructura a las necesidades del servicio. Si conocemos y monitorizamos el nivel de servicio podemos ajustar aumentando o disminuyendo la capacidad del sistema en función de las circunstancias. Sin adelantar inversiones ni asumir el riesgo de quedarnos sin presupuesto, Podemos establecer pautas de estacionalidad que permitan fluctuar la infraestructura de soporte en función de la demanda.

Herramientas. Las definidas por el product-owner.

Simulación. En este tipo de pruebas es posible utilizar las mismas pruebas que habremos realizado en entornos privados con anterioridad a la salida a producción. Evidentemente lanzaremos las pruebas en circunstancias controladas (fuera de horario comercial, en horas de baja demanda) y con un límite de concurrencia para evitar pérdidas de servicio en real.

Otro enfoque para realizar pruebas en producción es utilizar sondas en tiempo real que monitoricen el tráfico de los usuarios reales. Incluso podríamos usar como sonda las pruebas, lanzándolas periódicamente y verificando que los resultados están dentro de los valores esperados para cada rango horario.

Sea cual sea el enfoque elegido, en este caso la simulación lo que hace es captar información real del nivel de servicio para compararla a posteriori con los resultados de las pruebas y facilitar la toma de decisiones con respecto a la infraestructura.

Capítulo

6

6. MIDIENDO LA CALIDAD DE CÓDIGO EN WTF/Minuto. EJEMPLOS DE CÓDIGO REAL QUE PODRÍAN OCASIONARTE UN DERRAME CEREBRAL.

Autor: David Gómez

6.1 LA ANTOLOGÍA DEL DISPARATE.

Hace ya algunos años pasó por mis manos una referencia a un libro titulado "La antología del disparate", una obra escrita con el objetivo de recoger las respuestas más variopintas dadas por alumnos en exámenes.

Cuando leí algunos de sus contenidos, pensé que eran respuestas chistosas y ocurrentes, pero que probablemente no serían reales. Tiempo después pude comprobar que la realidad supera muchas veces la ficción y que, los alumnos de cualquier edad y asignatura, eran capaces de respuestas como las que recoge el libro… y de mucho más.

La “antología del disparate” es un fenómeno que puede aplicarse, no sólo a las respuestas de alumnos en exámenes, sino en prácticamente cualquier profesión y ámbito.

Ocurre cuando, por salir del paso, y por tratar de cumplir con el trabajo asignado, se busca la solución más a mano que podría funcionar.

Muchas veces la solución aplicada acaba siendo una barbaridad, un disparate o un sinsentido, sin que el autor sea consciente de que lo que está evidenciando es una falta de conocimiento, de tiempo o de cuidado que genera un producto falto de calidad.

¿Quién no ha oído, o dicho, dentro del ámbito de su trabajo aquello de “algún día tengo que recoger todas las barbaridades que me encuentro en el día a día”?

6.2 MEDIDAS DE CALIDAD FORMALES E INFORMALES.

Durante los estudios formales de Ingeniería, existen asignaturas dedicadas a que los alumnos (futuros ingenieros) comprendan cuál es el concepto de calidad y cómo medirla y asegurarla.

En ingenierías tradicionales la calidad se puede medir aplicando una serie de métricas: número de piezas defectuosas, umbrales de estrés o presión de las mismas, etc…

En el caso del sotfware, también se utilizan variables que permiten medir la calidad del software utilizado: Número de líneas de código por componente, acoplamiento del diseño, complejidad ciclomática de los

algoritmos, impacto de memoria bajo determinadas circunstancias, tiempos medios de respuesta, etc…

Se trata de medidas de calidad "formales", que nos permiten medir y comparar el resultado de un proceso de desarrollo software.

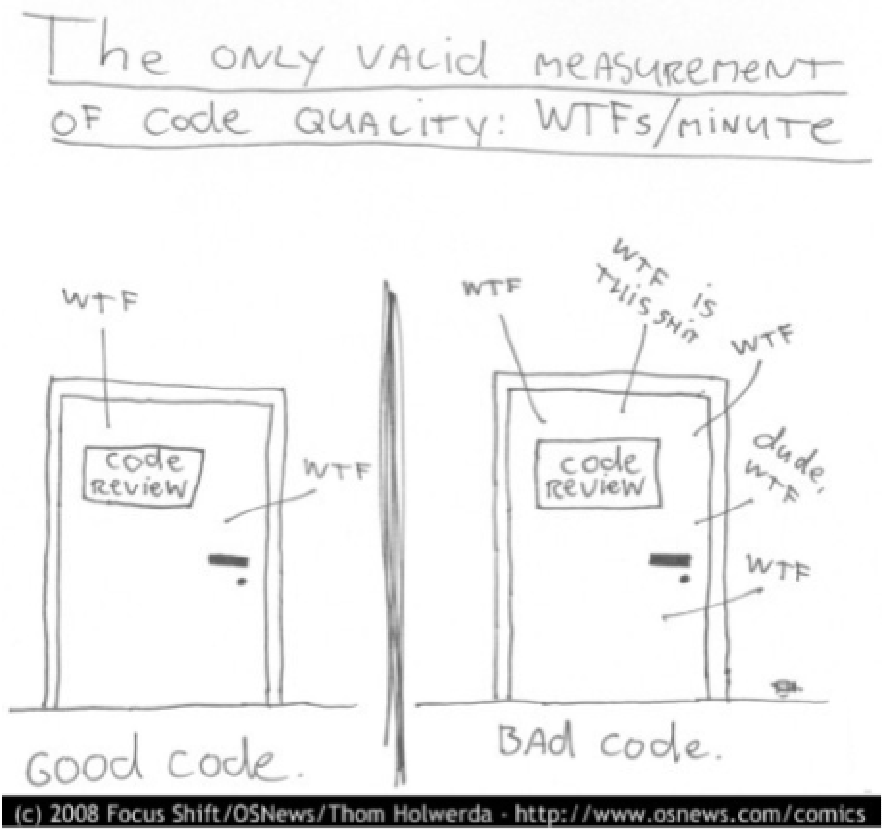
Existe, sin embargo, otro tipo de calidad: la calidad que percibimos cuando intentamos leer y seguir el código que forma parte de la aplicación. Es una medida de la calidad no menos importante que todas aquellas medidas formales, porque en el proceso de desarrollo es importante recordar que pasaremos más tiempo leyendo código ya escrito que escribiendo código nuevo.

Por tanto, la facilidad que tenemos para leer, seguir y entender el código que hemos escrito (o que hemos heredado y tenemos que mantener) debe ser tenida en cuenta como otro factor más de calidad del proceso de desarrollo.

Muchas de las medidas de calidad formales que utilizamos tratan precisamente de identificar no sólo la eficiencia del algoritmo, sino también cómo de fácil debiera ser entender y seguir el código a otro desarrollador.

6.3 LA ÚNICA MEDIDA VÁLIDA DE CALIDAD DE CÓDIGO: WTF/MIN

El concepto de la "calidad global del código" lo podemos redefinir en base a la viñeta de Tom Howeldar que titulaba "The only valid measurement of code quality is WTF/Min".



http://www.osnews.com/story/19266/WTFs_m

En esta viñeta, Thom Holwerda, recoge el concepto de medir la calidad de un desarrollo por el número de “disparates” que encontramos al leer el código.

El proceso para medir el código debería pasar por encerrar en una habitación a un desarrollador distinto al que escribió el código original,

pidiendo que revise todo el código de la aplicación, mientras nosotros esperamos tranquilamente fuera de la habitación.

Mientras dure el proceso de revisión de código, el supervisor que está fuera de la habitación, irá contando el número de exclamaciones que se oyen a medida que el proceso de revisión del código avanza.

Al final de la revisión, el número de estas exclamaciones, nos dará una buena indicación de la calidad global del código y de la facilidad que tiene para ser seguido y entendido. Esta facilidad nos dará la idea de cómo de fácil es mantener y evolucionar el desarrollo realizado.

Naturalmente, ésta métrica del WTF/min es un valor algo subjetivo e informal, pero da una idea alternativa y adicional que permite que tengamos un objetivo divertido con el que mantener un código limpio y de calidad.

6.4 EJEMPLOS Y CONTRAEJEMPLOS PARA EVITAR WTFS.

En el proceso de desarrollo y revisión de código hay algunas áreas donde es fácil encontrar patrones y reglas que nos hagan exclamar “¿qué es esto?” a medida que leemos dicho código.

Revisamos alguno de estos puntos donde recurrentemente podemos fallar al tratar de escribir código de calidad.

6.4.1 COMENTARIOS.

Los comentarios al código son una herramienta muy potente, porque nos permite añadir anotaciones que no serán tenidas en cuenta por el compilador o por el entorno de ejecución, pero que pueden aportar información de utilidad para la persona que leerá el código y que tiene que mantenerlo o evolucionarlo.

Los comentarios pueden servir también para generar documentación de un API, de forma automática, a través de herramientas que analizan el código y dichos comentarios.

Sin embargo, los comentarios son también una de las herramientas que más fácilmente pierden su utilidad al no ser usadas correctamente:

- Comentarios que repiten exactamente lo que está escrito en el código. En este caso, la información es redundante, ya que la misma información se puede extraer directamente del código.

- Comentarios que han quedado obsoletos, indicando una cosa distinta a lo que dice el código. Este suele ser un caso en el que derivan los comentarios del punto 1, cuando el código se modifica y el desarrollador olvida modificar el comentario original.

- Comentarios de partes de código. Cuándo el comentario se utiliza para desactivar una parte de código y que no sea interpretado por el compilador la legibilidad del código se ve afectada.

- Comentario procedente de plantillas, generados directamente por el IDE, que acaban quedando como bloques sin ningún significado que alargan el código y no aportan ningún valor.

Este mal uso de los comentarios, puede acabar haciendo que un desarrollador o revisor, los considere inútiles y acabe por no leer ninguno de los comentarios por defecto.

Los comentarios deberían ser utilizados en los casos en que

- Describan la motivación del diseño o dejen información adicional al propio código, por ejemplo, cuando el código está implementando un 'workaround' a un mal comportamiento de la tecnología subyacente.

- Indiquen información sobre precondiciones, invariantes o post-condiciones. Por ejemplo, por qué es o no necesario hacer una determinada validación, basada en una precondición.

- Explican los casos de uso del fragmento de código o el fundamento del algoritmo.

- Deberán formar parte de la documentación de un API

6.4.2 EXCEPCIONES.

En muchos lenguajes de programación, existe el concepto de Excepción, como mecanismo para gestionar los casos de error.

Aunque no es muy habitual, es posible encontrar casos donde la excepción se utiliza para controlar el flujo del código o buscan efectos laterales al flujo normal del desarrollo.

Algunos ejemplos de mal uso de las excepciones los podemos encontrar en

- Romper la ejecución normal del flujo, finalizando bucles o provocando la salida de una pila de llamada de métodos, que no está ocasionada directamente por un error.

- Utiliza la creación de una excepción para acceder a la información sobre la pila de llamadas u otra información sobre el entorno de ejecución.

Este uso de las excepciones supone una mala práctica, porque rompen la línea natural del algoritmo y el seguimiento que podemos hacer del código.

La regla general siempre debe ser que las excepciones deben ser generadas sólo cuando se haya producido una situación de error, o un caso no contemplado en la casuística normal que puede gestionar el código.

Por otro lado, las excepciones sólo deben tratar de capturarlas cuando se tenga la posibilidad de gestionar la situación de error y tomar alguna acción correctiva.

Las excepciones no deben ser ignoradas nunca, porque pueden ocultar el motivo de un malfuncionamiento que luego no podamos depurar.

6.4.3 NOMBRADO

El nombre que ponemos de los elementos que forman parte de nuestra aplicación es muy importante y debemos siempre tratar de que sean descriptivos de:

- lo que hacen (en el caso de funciones o métodos),
- de lo que contienen (en el caso de variables o atributos),
- de lo que representan (en el caso de tipos o clases).

Siempre hay que recordar la máxima de que "Un fragmento de código se lee muchas más veces de las que se escribe". Por tanto, el esfuerzo que dediquemos para bautizar un tipo, una variable, o un método tendrá su rendimiento posterior en la rapidez con la que se lea y comprenda el código, lo que redundará, a su vez, en la facilidad del mantenimiento del mismo.

Es fácil no encontrar el mejor nombre a la primera, por lo que no debemos renunciar al renombrado de atributos o clases y a la refactorización de métodos cuando, al volver sobre código ya escrito, encontremos un nombre mejor.

6.4.4 CÓDIGO INNECESARIO.

Otro punto frecuente que reduce la calidad de los desarrollos es el código innecesario, redundante, duplicado o código muerto (el código que nunca llega a ejecutarse).

Al igual que los comentarios que suponían una réplica del código, el código innecesario supone una distracción efectiva a la hora de leer y entender qué hace un componente software.

Algunos casos que deben ser evitados porque provocan la aparición de código innecesario son:

- Tratamiento temprano de excepciones. Cuando las excepciones se capturan en puntos en los que no pueden ser tratadas con éxito, acaba generando código que simplemente relanza la excepción.

- El recubrimiento directo de APIs o de librerías de terceros. Si ya hay un API pública, o una librería común que ofrece una funcionalidad, puede utilizarse directamente.

- El intento de excesiva optimización del código. Dependiendo del entorno de ejecución, este intento de optimización de código puede ser, en realidad, contraproducente.

- La refactorización y simplificación progresiva del código, que puede provocar que código auxiliar inicial, haya dejado de ser necesario.

6.5 MEJORA DEL PROCESO DE DESARROLLO Y ASEGURAMIENTO DE LA CALIDAD DE CÓDIGO.

6.5.1 CUANDO SE MIRA LA CALIDAD SÓLO AL FINAL.

Es frecuente encontrarse con proyectos y procedimientos en el que las auditorías y medición para asegurar la calidad de código se realiza al final del proceso, cuando ya es tarde para tomar medidas correctivas o tomarlas implica un retraso en la puesta en producción y un impacto económico en el proyecto.

6.5.2 INTEGRACIÓN PROGRESIVA DE LA CALIDAD EN EL PROCESO

Para evitar que nuestro código sufra de falta de calidad, que no sea legible complique su facilidad para mantenerlo y evolucionarlo, es interesante incorporar en nuestro proceso de desarrollo, hábitos que permitan llegar al final de la construcción con la garantía de que la calidad es parte integral del proceso.

Sin embargo, ese objetivo deseable no es fácil de implementar de golpe. En esos casos, podemos plantear una serie de cambios que, progresivamente, vayan mejorando la forma en que aseguramos la calidad del software en nuestro proceso.

Estos pasos se pueden ir abordando poco a poco, consiguiendo una mejora progresiva de la calidad del código y de la forma en que se incorpora la misma en el proceso.

6.5.2.1 Evaluación intermedia de la calidad

El primer paso consiste en ir realizando la revisión del código y la comprobación de calidad en puntos intermedios de la planificación del desarrollo.

De esta forma, los informes y las conclusiones de calidad se pueden incorporar antes del final del desarrollo, se pueden corregir los defectos y patrones incorrectos antes de que estén demasiado extendidos por la base de código y se pueden incorporar recomendaciones para evitar que se sigan reproduciendo.

6.5.2.2 Automatización de los controles de calidad

Una vez que se han incorporado en la planificación los controles manuales de calidad, el siguiente paso natural es el automatizar la ejecución de estos tests de calidad.

Esta automatización, puede hacerse a través de:

- herramientas específicas como PMD, checkstyle, y reglas sobre el código.
- de procesos planificados con un cron y
- de la integración, a través de la automatización del proceso de construcción, de la ejecución de los tests de calidad.

6.5.2.3 Recogida de medidas de calidad a través de herramientas.

Una vez que tenemos automatizada la ejecución de los controles de calidad, es recomendable introducir en el ecosistema de desarrollo herramientas que permitan recoger los resultados de estas ejecuciones y mantener un histórico de las mismas.

De esta forma, tendremos un control de cómo ha ido evolucionando el desarrollo y la calidad del mismo a lo largo del tiempo.

Algunas herramientas muy útiles para este punto son Sonarqube y la posibilidad de integración que permite con Jenkins.

6.5.2.4 Elección de valores umbral sobre la calidad.

Una vez que hemos automatizado los controles de calidad y los resultados de estos se recogen con herramientas como Sonarqube, el siguiente paso es definir unos valores que nos aporten una idea "medible" de cómo evoluciona la calidad del código.

Para la elección de estos valores, podemos tomar medidas de calidad formales como las que hablamos al principio:

- Complejidad ciclomática
- Nivel de anidamiento de bloques de código
- Longitud máxima y media de métodos y clases
- Nivel de acoplamiento entre componentes.
- Uso de variables locales.
- Violaciones de reglas y patrones de código.
- Porcentaje de cobertura de código por los tests...
- …

La violación de determinadas medidas o la superación de determinados umbrales pueden definir criterios de aceptación o no de un entregable para ser desplegado en los entornos de producción del cliente.

6.5.2.5 Configuración de Alarmas cuando los valores umbral de calidad son superados.

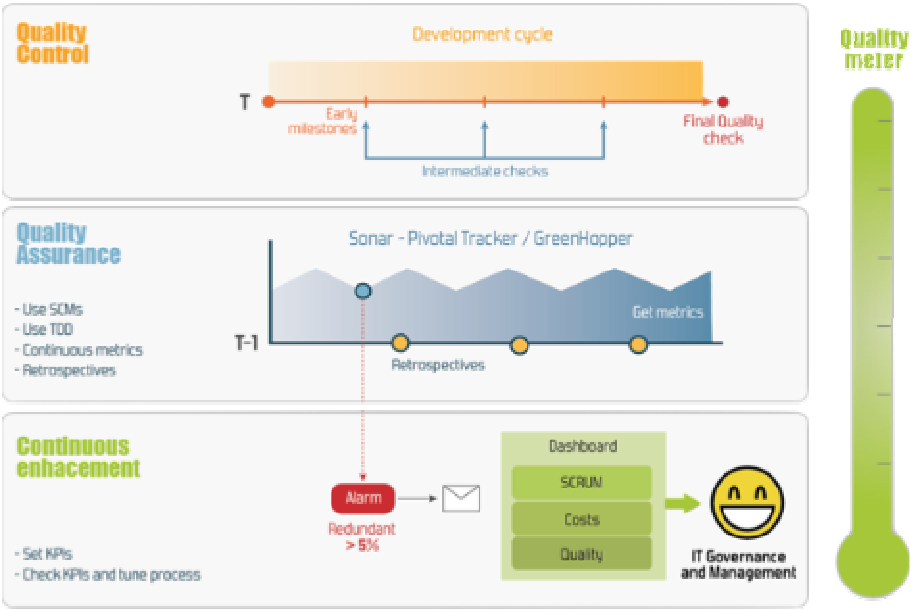
Con la definición de valores medibles y umbrales de calidad que no estamos dispuestos a que sean superados, es posible configurar alarmas automáticas que nos supongan una notificación a los desarrolladores y a los responsables del desarrollo de forma que el mero hecho de haber bajado los niveles recomendados o aceptables de calidad no pasen desapercibidos y puedan ser subsanados cuanto antes dentro del proceso de desarrollo.

6.5.2.6 Creación de un panel de control

Llegados a este punto, sólo queda recoger la información de todas estas métricas y valores medidos en una consola centralizada, incluso para todos los proyectos, de forma que toda la calidad de los procesos de desarrollo pueda ser revisada en un único sitio.

Este punto hace que la gestión del desarrollo y gobierno de proyectos sea mucho más sencilla, permitiendo tener una visión global de qué proyectos necesitan más esfuerzo y supervisión y cuáles están siguiendo un proceso más adecuado.

Una vez que completamos estos pasos, habremos conseguido mejorar nuestro proceso de desarrollo integrando de forma sencilla, progresiva y natural el aseguramiento de la calidad durante la construcción.


Quality Control
Development cycle
T
Early milestones
Final Quality check
Intermediate checks
Quality Assurance
- Use SCMs
- Use TDD
- Continuous metrics
- Retrospectives
Sonar - Pivotal Tracker / GreenHopper
T-1
Get metrics
Retrospectives
Continuous enhacement
- Set KPIs
- Check KPIs and tune process
Alarm
Redundant > 5%
Dashboard
SCRUM
Costs
Quality
IT Governance and Management
Quality meter

Capítulo

7

7. LA EXPERIENCIA DEL CORTE INGLÉS EN LA GESTIÓN DE LA CALIDAD SOFTWARE.

Autor: Jesús Hernando

"Cualquiera puede escribir código que un ordenador pueda entender. Los buenos programadores son aquellos que escriben código que los humanos puedan entender"

Martin Fowler

La evolución constante de las nuevas tecnologías, canales de comunicación y el constante cambio de los mercados, hacen que se multiplique la complejidad del desarrollo. Velocidad, inestabilidad, incertidumbre, unido a la necesidad de entrega de valor constante al Negocio, a la par que

disminuir el "Time to Market"[2], inducen a desarrollar software/productos que no alcanzan los mínimos de calidad esperado (Wishful thinking[3]).

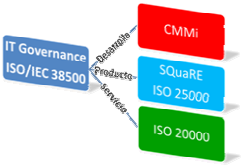
"Cuando encontré las respuestas me cambiaron las preguntas"

7.1 CALIDAD DEL SOFTWARE

Los procesos de Negocio de nuestras Organizaciones, están sustentados por *Productos Software*, siendo estos uno de los factores diferenciadores frente a sus competidores.

7.1.1 DIMENSIONES DE LA CALIDAD DEL SOFTWARE

Cuando nos referimos a **"Calidad Software",** subyacentemente nos referimos a cuatro tipos de calidad:

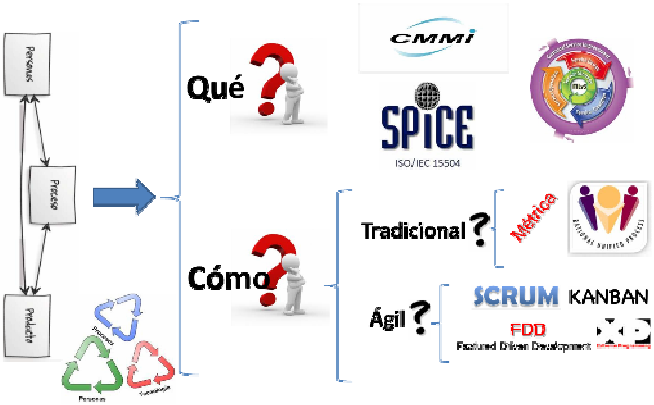


1. La del Proceso
2. La del Producto
3. La de las Personas/Equipos.

[2] *Voluntad de querer cumplir de cualquier manera un determinado objetivo por imposible que sea*
[3] *Software funcionando cuanto antes*

4. La del Servicio

Diferenciemos claramente:



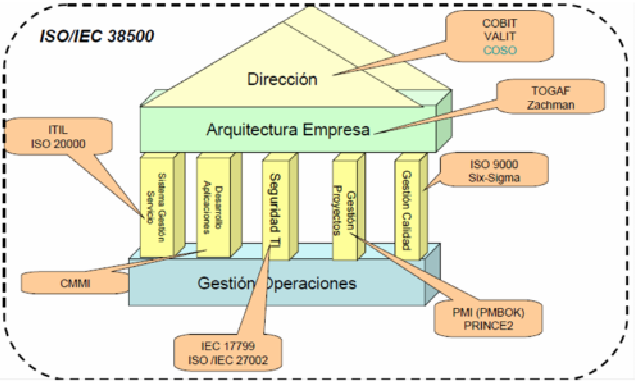
Confundir Calidad del Proceso y Producto, conlleva a falsas expectativas en el desarrollo del software.

Hay poca evidencia de que cumplir un modelo de procesos asegure la calidad del producto[4], es más, consideramos que estandarización de los procesos garantiza uniformidad en la salida, lo que puede incluso institucionalizar la creación de malos productos.

[4] *Kitchenham y Pfleeger, 1996*

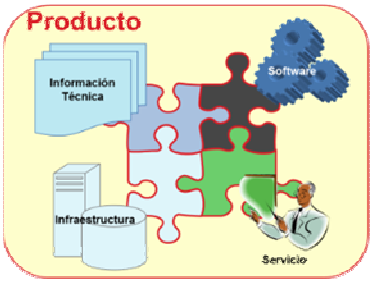
7.1.2 NORMATIVAS EN EL ÁMBITO EMPRESARIAL (TI)

Marco Normativo de referencia del Sector TI:



¿Cuál es la referencia de Calidad de Producto Software?

7.1.3 ¿QUÉ ES UN PRODUCTO?



Un Producto no es sólo la implementación software, escrito en un determinado lenguaje de programación, de un conjunto de funcionalidades, claras y explicitas, que permiten establecer un proceso de negocio en nuestras Compañías, con el fin de crear **Valor**.

Todo lo generado en la Cadena de Valor, en aras a la generación de un activo software, forma parte inherente del mismo. A este pool de asset lo denominamos **Producto**[5].

Por ende, cuando hablamos de Calidad del Software, no sólo nos estamos refiriendo al software en sí, nos referimos a la ***Calidad de todos los elementos que se construyen a lo largo del ciclo de vida de un proyecto***: ingeniería de requisitos, diseño, arquitectura, documentos de despliegue, desarrollo, plan de pruebas,… todo lo relativo al producto.

7.1.4 CALIDAD

La Calidad es la suma de todos aquellos aspectos o características de un producto o servicio, que influyen en su capacidad para satisfacer las necesidades de los usuarios. La satisfacción del usuario está determinada por la diferencia entre la calidad percibida y la calidad esperada, cuando éste hace uso de un producto o servicio.

Algunas definiciones relevantes de Calidad del Software:

- La concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares y procesos de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente[6].

[5] *Conjunto de funcionalidades altamente cohesionadas que dan soporte e un proceso de negocio. Implementadas en una o varias plataformas*
[6] *Roger S. Pressman (Software Engineering: A Practitioner's Approach. McGraw-Hill*)

- El conjunto de características de una entidad que le confieren su aptitud para satisfacer las necesidades expresadas y las implícitas (*ISO 8402 (UNE 66- 001- 92)*)

La calidad del software no es algo en lo que se empieza a pensar una vez que se ha generado el código. Según R Pressman[7] el Aseguramiento de la Calidad es una "**actividad de protección**" que se aplica a lo largo de todo el proceso de ingeniería software y engloba:

- Un enfoque de gestión de la calidad.
- Tecnología de ingeniería del software o del conocimiento efectivo (métodos y herramientas).
- Revisiones técnicas formales que se aplican durante cada paso de la ingeniería del software o del conocimiento. Apoyado en una estrategia de pruebas por niveles.
- Procesos de gestión de la configuración. El control de la documentación del software y de los cambios realizados.
- Un procedimiento que asegure un ajuste a los estándares de desarrollo del software (cuando sea posible).
- Mecanismos de medición y de información.

Así, se debe **garantizar** la calidad de cada uno de los subproductos del Desarrollo y con ello que los aplicativos **cumplen con los requisitos**

[7] *Software Engineering: A Practitioner's Approach. McGraw-Hill*

especificados por el Usuario y que **pueden ser puestos en operación**, para funcionar **de acuerdo a lo esperado**, de manera correcta y eficiente, **sin afectar los sistemas** que actualmente están **en operación.**

7.2 REFERENCIA EN LAS ORGANIZACIONES PARA EL CONTROL DE LA CALIDAD

La aparición de la nueva familia de normas ISO/IEC 25000 - **SQuaRE** [8], establece un marco de trabajo común para evaluar la calidad del producto software.

Guía el proceso en base a subcaracterísticas, denotadas en la siguiente imagen:

Es importante indicar que la familia incluye, al margen del modelo de la calidad del producto, la definición de un proceso para llevar a cabo la evaluación del producto software (Norma ISO/IEC 25040)

[8] *Software product Quality Requirements and Evaluation*

7.3 ÁMBITO EMPRESARIAL

El éxito no se logra sólo con cualidades especiales. Es sobre todo un trabajo de constancia, de método y de organización

(J.P Sergent)

7.3.1 PLAN DE ASEGURAMIENTO DE LA CALIDAD

La calidad es una cualidad esencial de cualquier producto generado por una organización, que va a ser usado por otros. Es por ello que una de las fases/actividades principales de la elaboración de un Producto, es el aseguramiento de la calidad: **Plan de Aseguramiento de la Calidad del Producto (SQAP)** [9]**.** Al igual que el conjunto de requisitos establecen las funcionalidades y los límites del Producto, el SQAP describe los umbrales de calidad asociados al Producto, y marca los axiomas/límites de aceptación para su traspaso a Servicio Continuo. Es obvio que estos, dependen en gran medida del objetivo final del Producto y el software del mismo: software de sistemas, tiempo real, gestión, ingeniería y científico, inteligencia artificial, etc. El software, debe ser entendido como **Producto y como Servicio**, y por ende es fundamental el aseguramiento de la generación de Productos de Calidad, previo paso a transición a Servicio continuo.

[9] *Plan de Aseguramiento de la Calidad del Producto*

7.3.2 MODELOS DE DESARROLLO ORIENTADOS A PRODUCTO: CALIDAD CONCERTADA

Así, debemos ir a un ***modelo orientado al producto no al proyecto***, basado en una ***calidad concertada***, que contemple pruebas de calidad en todo el ciclo de desarrollo.

Entendiendo por ***calidad concertada***, *el compromiso de entrega del desarrollo de un producto, con el cumplimiento estricto de lo descrito en el plan de aseguramiento de la calidad del mismo, SQAP, con las evidencias que se consideren necesarias, pasadas por exhaustivos planes de pruebas por parte del proveedor*.

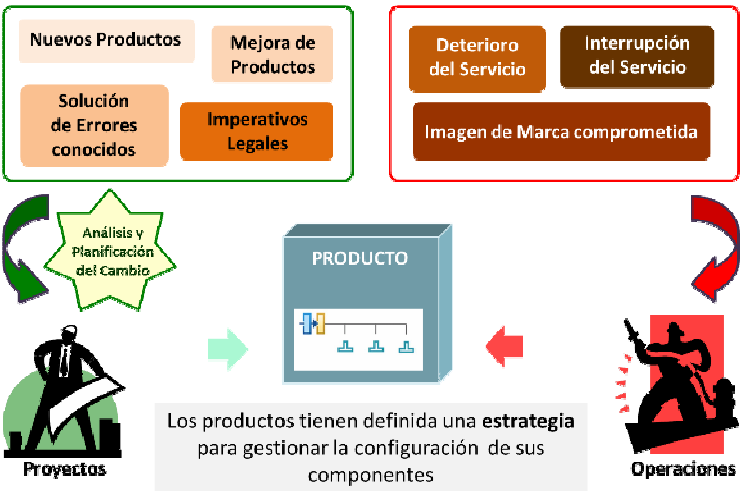
7.3.3 CATALOGACIÓN DE PRODUCTOS

Debemos catalogar nuestros activos de Productos y definir clara y concisamente el plan de aseguramiento de la calidad (SQAP) de cada uno de ellos. Este debe ser conocido y compartido con todos y cada uno de los participantes en el ciclo de vida de desarrollo, bien sea dentro o fuera de nuestra Organización.

- **Producto** es el dominio de funcionalidad que soporta un proceso de negocio (o parte de él) conforme a la estrategia de creación de valor del negocio
- Es decir, **producto** es la solución software que se entrega al usuario para satisfacer sus peticiones y que se obtiene mediante la ejecución del proyecto
- No es lo mismo **desarrollar productos** que **ejecutar proyectos**

Para que los **productos** puedan atender nuevas peticiones de negocio de forma **eficaz** es necesario gestionar la información relativa a:

- **Características del producto:** funcionalidades, reglas de negocio, …
- **Componentes del producto:** arquitectura, modelos, código, …
- **SQAP del producto:** nivel de calidad, casos de prueba, verificaciones, …



7.3.4 PLAN DE CERTIFICACIÓN

El **Plan de Certificación de un Producto** (aseguramiento de que el mismo cumple con lo establecido en su SQAP) debe estar alineada con cuatro pilares básicos de un entorno de desarrollo software: arquitecturas y tecnologías en las que se basan, herramientas de desarrollo y de apoyo al mismo, metodología a seguir, y *Plan de Aseguramiento de la Calidad* con el que se medirá la calidad de los proyectos y de los productos (cada conjunto de proyectos que den lugar a un nuevo producto o a un evolutivo de uno existente, tomará como referencia el SQAP).

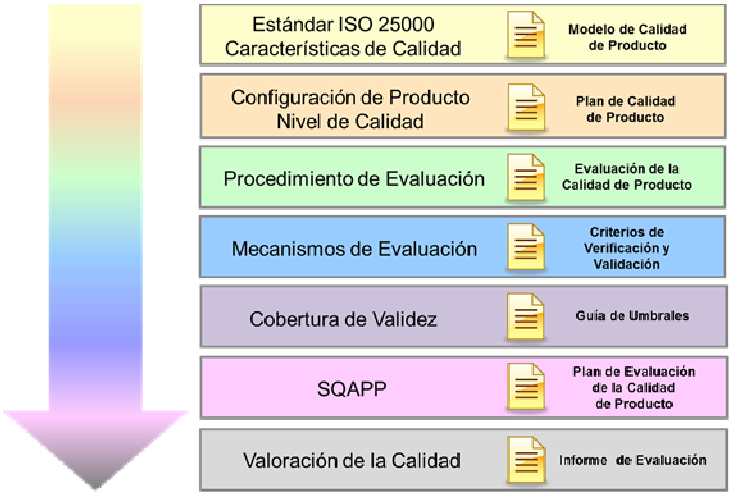
Para ello, y en base SQuaRE definir un proceso de ***V&V***[10] *(verificación y validación)*, sin hitos, sin barreras, que sea proactivo y en la medida de lo posible automatizado dentro de su cadena productiva.

La Calidad es inherente al Producto, forma parte inseparable del mismo, es un factor diferenciador para nuestras Organizaciones.

La Calidad no se negocia.

El control de la calidad del producto software, es aún más relevante, teniendo presente que las grandes organizaciones a día de hoy, implantan dentro de sus Ciclos de Vida de desarrollo, modelos de **outsourcing** y/o **outtasking.**

[10] ***Pruebas de verificación****: orientadas a garantizar la calidad de todos los Assets* ***No Software*** *generado en la vida del proyecto y que forman parte inseparable del Producto.*
Pruebas de validación*: orientadas a garantizar la calidad todos los Assets* ***Software*** *generado en la vida del proyecto y que forman parte inseparable del Producto*



Los productos tienen definido un **nivel de calidad** que determina:

- el **umbral** para las tareas de aseguramiento de la calidad
- las **características** de calidad que reúne el producto

7.3.5 GRUPO DE QUALITY ASSURANCE (QA)

Es imprescindible una gestión unificada de los distintos niveles de pruebas, de manera que todos los resultados se recojan en un informe ejecutivo que *certifique* la calidad global del Producto.

Por ende, en las Organizaciones, debería existir un **Grupo de Quality assurance** (**QA**), el cual, tomando como base el SQAP del Producto y SQuaRE, certifique la Calidad del Producto previo paso a Operaciones/Servicio Continuo.

El objetivo principal del Grupo de Quality Assurance, es **garantizar la Calidad del Producto Software** así como **los subproductos** necesarios para su evolución en la vida de este**.**

- Certificar la calidad de las Productos aplicando un Plan de aseguramiento de la calidad (SQAP).
- Entregar Productos estables y validadas.
- Controlar la calidad de los entregables de los equipos de desarrollo.
- Obtener informes y estadísticas de calidad para monitorizar el proceso.
- Asegurar:
    - Que el Producto software cumple con los requisitos especificados por el usuario.

- Que el Producto funcionará de acuerdo a lo esperado de manera correcta y eficiente.
- Que la operación del producto no afectará al resto de sistemas existentes.
- Que la evolución del producto a lo largo su vida será viable y sencilla garantizando la calidad de toda la información técnica necesaria para su evolución.

7.4 AUTOMATIZACIÓN DEL CICLO DE VIDA DE DESARROLLO

Se alcanza el éxito convirtiendo cada paso en una meta y cada meta en un paso

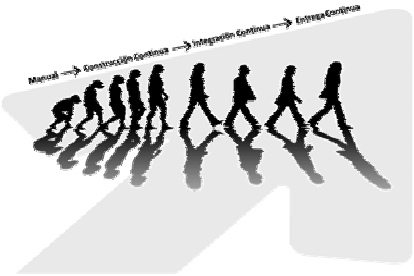
(C.C Cortéz)

En un futuro, a corto y medio plazo, teniendo en mente el estado presente y futuro de la Ingeniería de Software en las Organizaciones Empresariales, inducen a las compañías a disponer de plataformas *ALM* (*application lifecycle management)* que automatice y profesionalice, partes de sus actividades de desarrollo.

Las organizaciones deberían establecer el siguiente plan estratégico:

- Visión Producto frente a proyectos
- Diferenciación de modelos de Calidad de Proceso, Producto y Servicio
- Establecimiento del Proceso del SQAP y la familia SQuaRE
- Disponer de una plataforma de Continuous Integration
- Constitución del Grupo de QA: Certificación de Productos

- Continuous Delivery: reducción del time to market.



Todo ello, sustentado bajo el pensamiento Lean y la mejora Continua.

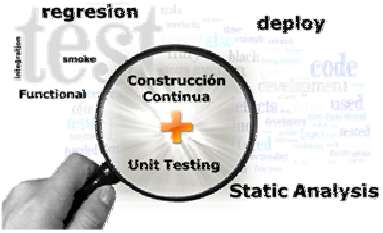
De manera paralela, las Organizaciones deben tener definido e implantado los procesos de Gestión del Cambio, Gestión del Despliegue y Gestión de la Configuración (ISO 20000) en aras a favorecer la transición a servicio continuo.

Para conseguir este fin, la comunicación, integración y colaboración entre los grupos de desarrollo de productos software y el entorno de las operaciones, transciende a los silos técnicos y requiere la medición continua y los objetivos de equipo: **DevOps**.

7.4.1 INTEGRACIÓN CONTINUA

Se necesite una Ingeniería de Software:

- Ingeniería Concurrente: control “continuo” del código
- Costes de no Calidad: análisis de la situación en etapas tempranas.
- Ejecución dinámica de pruebas
- Gestión del compilado, empaquetado y despliegue
- Gestión Integrada del Ciclo de Pruebas

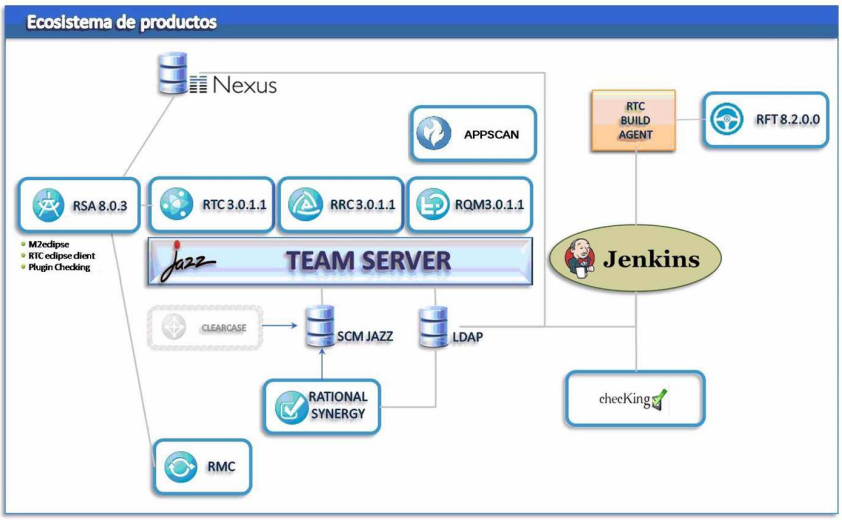


Todas estas actividades V&V (Validación y Verificación) deberían estar integradas dentro de un proceso proactivo, eliminando y/o minimizando los costes de no calidad, sin hitos, ni barreras, integrado en el Ciclo de Vida de desarrollo.

Así hablamos de automatización del Ciclo de Vida de Desarrollo (**Plataforma de Integración Continua (IC), Orientada a la Calidad del Software).**

7.4.1.1 Plataforma

Un ejemplo de ecosistema de productos, que nos puede ayudar al control proactivo de la Calidad de Producto Software:

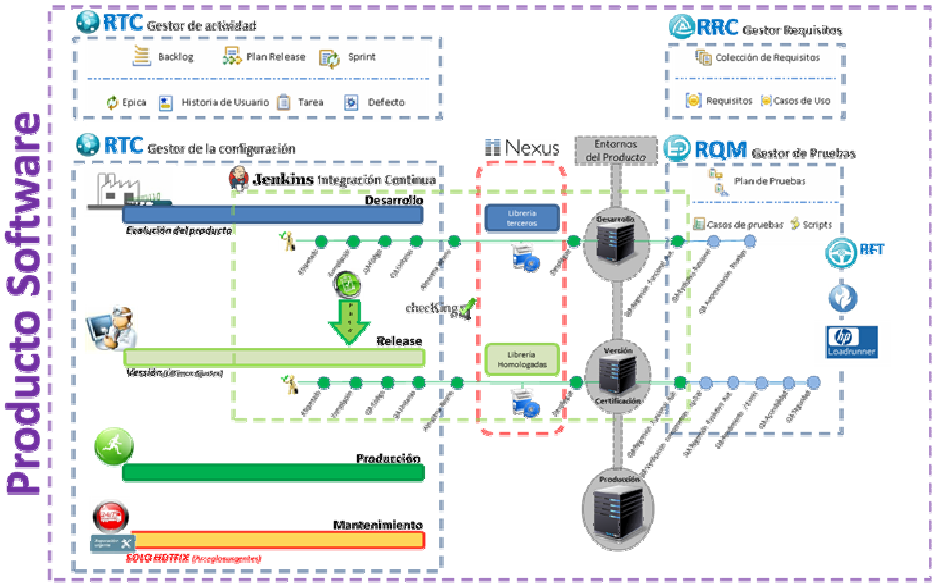


7.4.1.2 Uso de las herramientas en el modelo

La siguiente imagen, especifica las responsabilidades de los Productos del ecosistema en lo que se refiere a:

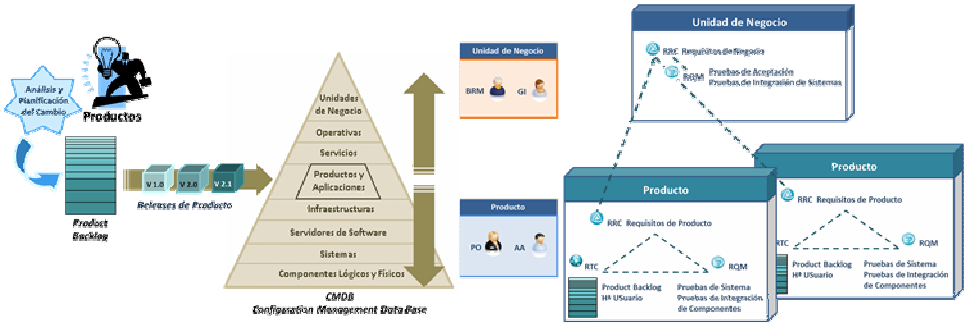
- Requisitos y Características de Producto
- Gestión de la actividad
- Gestión de Pruebas
- Gestión de la Configuración
- Gestión de Activos Software

Que constituyen la base del proceso de ingeniería de Software o cadena de valor:

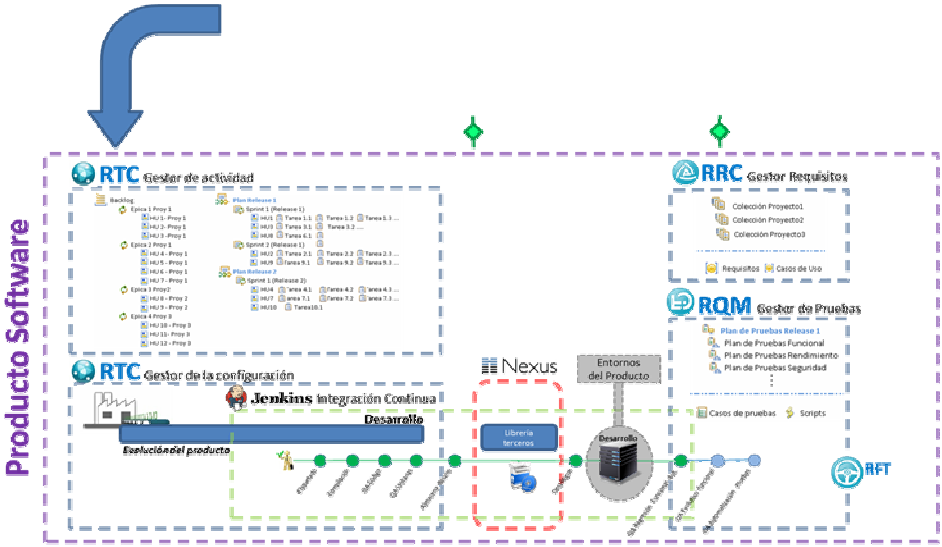


7.4.1.3 Visión producto

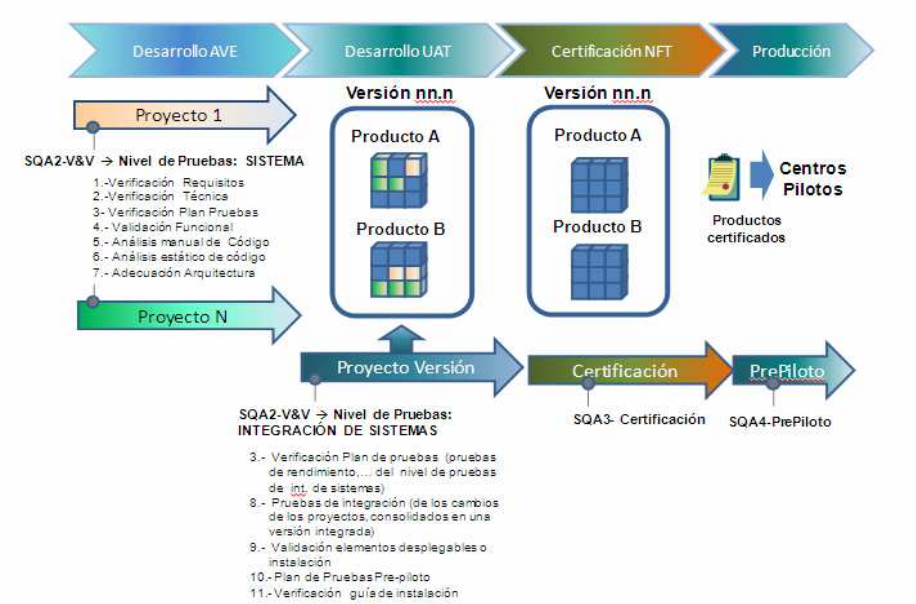
PROYECTOS TI: Conjuntos de **cambios** (*change requests*) sobre **productos TI** que tiene sentido implementar desde el punto de vista del Negocio o de Sistemas de Información.



RELEASES DE PRODUCTO: conjuntos de **cambios** entregados para atención de las peticiones/ solicitudes recibidas que configuran distintas **versiones** de los productos TI.



Los Productos catalogados son evolucionados por n proyectos que se engloban en un conjunto de releases, que dan paso a través de Gestión del Cambio y Gestión de Despliegue a un conjunto de servicios de certificación. El siguiente gráfico muestra la realización de dichos servicios de certificación en los proyectos de desarrollo y de integración de versión:

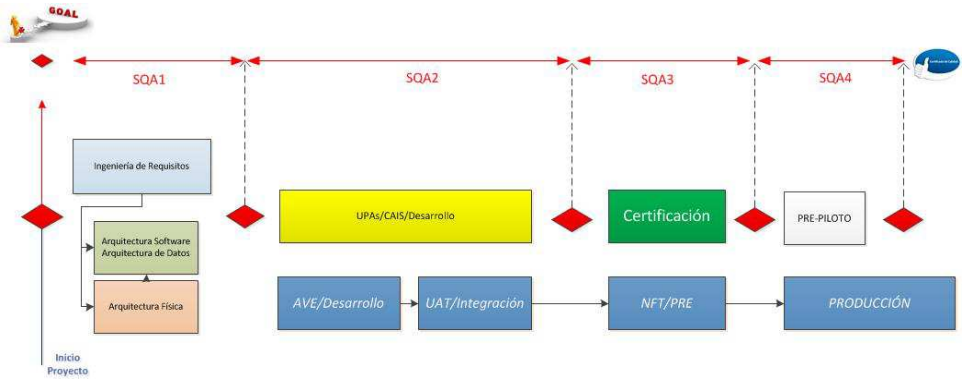


7.4.2 ENTREGA CONTINUA

En esta línea, la competitividad, calidad y el *Time to Market,* conducen a las organizaciones a disponer de un proceso de entrega de **valor** constante al Negocio *(Software funcionando cuanto antes !!!)* es decir, un proceso de **Continuous Delivery (CD)** apoyado en el proceso de Integración Continua, donde el control de la Calidad de manera automatizada cobra aún si cabe, más relevancia

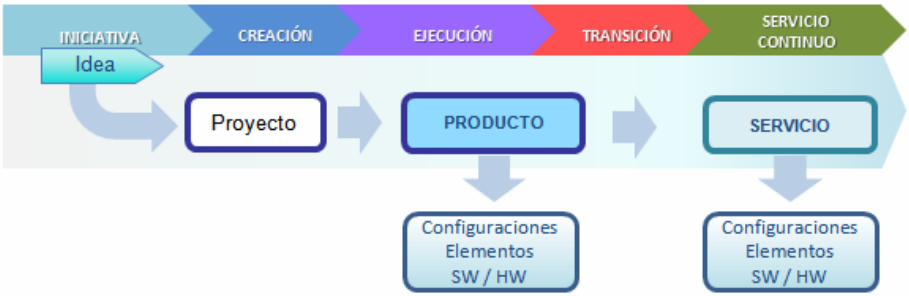
7.5 TECHNICAL QUALITY MANAGEMENT PRO-ACTIVO Y CONTINUO

El modelo de Certificación de Productos, bajo Calidad Concertada y tomando como base el SQAP, se realiza en base a una seria de servicios ejecutados a lo largo de la Cadena de Valor (SQAx):

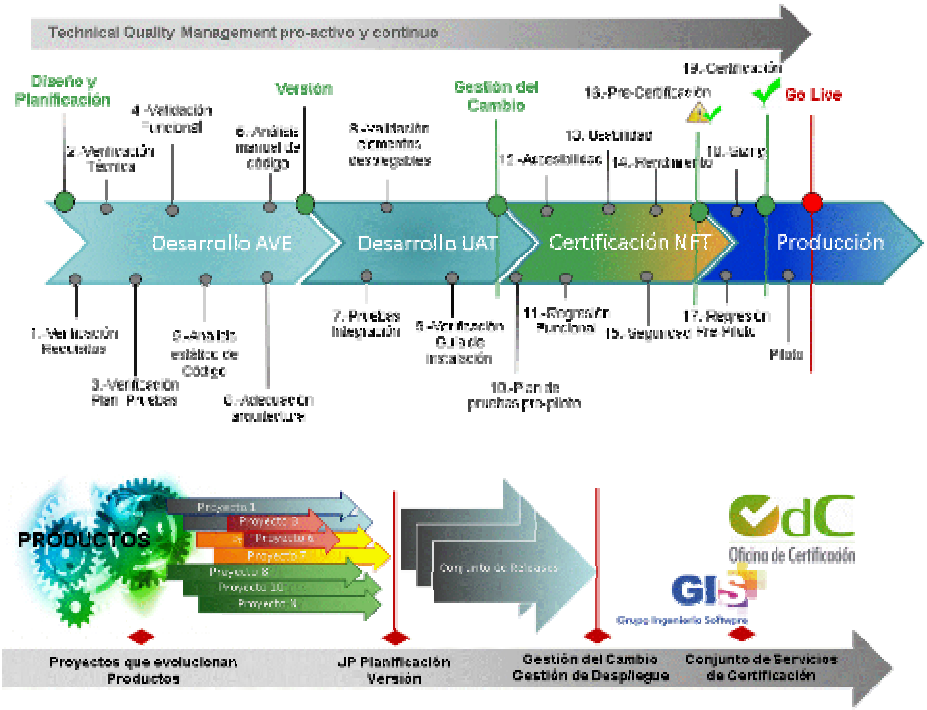


La Oficina de Certificación (OdC) u oficina de QA, de forma proactiva, actúa sobre las evidencias de aseguramiento de la calidad generadas durante la ejecución del proyecto con el fin de evaluar y certificar el nivel de calidad de los productos resultantes.

Realiza el control y evaluación de calidad de acuerdo al "Modelo de Calidad de Producto", que fundamenta la evaluación en una serie de características y subcaracterísticas de la calidad de producto (adecuación funcional, rendimiento, fiabilidad, etc.). La realización de dichas acciones hará posible que la OdC emita un informe concluyente de la calidad del producto, para la toma de decisión de la transición del producto/s a servicio.



El siguiente gráfico ilustra el conjunto de actividades realizadas a lo largo del CV para la Certificación:



7.6 REFERENCIAS

I Jornadas de Calidad de Producto Software
http://calidaddelproductosoftware.com/

http://www.javiergarzas.com/

IEEE http://www.ieee.org/index.html

PMI http://www.pmi.org/

ISO 25000 http://iso25000.com/

CMMi http://www.sei.cmu.edu/cmmi/

ITIL http://www.itil-officialsite.com/

Cobit 5 http://www.isaca.org/COBIT/Pages/default.aspx

Scrum Alliance http://members.scrumalliance.org/

Waterfall, Lean/Kanban, and Scrum
http://kenschwaber.wordpress.com/2010/06/10/waterfall-leankanban-and-scrum-2/

http://www.software-quality-assurance.org/index.htm

Capítulo

8

8. ¿QUÉ CALIDAD TIENE REALMENTE EL SOFTWARE? LA EXPERIENCIA DE EVALUAR LA CALIDAD DE 1000 PROYECTOS BAJO LA ISO 25000.

Autor: Ana María García

8.1 ¿POR QUÉ ANALIZAR LA CALIDAD DE 1000 PROYECTOS SOFTWARE?

Cada cierto tiempo solemos tener noticias de algún fiasco software derivado de la falta de preocupación por la calidad en cualquiera de sus dimensiones (producto, procesos o personas) pero… ¿Sabemos realmente cuál es el estado general de la calidad de los proyectos software? ¿Los programadores siguen buenas prácticas de programación? ¿Cuáles son las principales carencias de calidad que más se repiten entre los proyectos?

Profesionalmente, en lo que a calidad del producto software se refiere, nos solemos encontrar con que no se sabe cuál es el estado general de la calidad de los proyectos software.

Por otra parte, no existen umbrales de métricas de calidad recomendados y aceptados universalmente. Y si hay cosas de este tipo suelen estar desactualizadas.

Por ello, desde un tiempo a esta parte, han surgido distintas iniciativas orientadas específicamente a estudiar distintos aspectos de calidad de proyectos concretos.

Por ejemplo, desde 2006, Coverity lleva a cabo el proyecto Coverity Scan, junto con el departamento de Seguridad Interna de Estados Unidos, con el objetivo de ayudar a la comunidad de desarrollo open-source a mejorar la calidad del software que llevan a cabo.

Cada año, Coverity compara la calidad y seguridad de los proyectos open-source con la de proyectos comerciales. Su último informe (Coverity, 2013), obtenido con un muestreo de 300 proyectos open-source y 300 comerciales, concluyó que todavía los proyectos open-source tenían más densidad de defectos (defectos por cada 1000 líneas de código) que los proyectos comerciales de la muestra.

En 2013, (Almossawi, 2013b) analizó la mantenibilidad de Firefox, e hizo accesible esta información a través de su página web (Almossawi, 2013a). En ese mismo año, (Gómez, 2013) analizó la complejidad de las implementaciones TodoMVC.

Sin embargo, vemos que las iniciativas de este tipo son escasas y los muestreos de proyectos analizados, pequeños. Además son iniciativas que no están globalmente aceptadas, o son dependientes de herramientas comerciales concretas, como en el caso de Coverity.

Motivado por todo esto, se ha realizado este caso de estudio, analizando aspectos de calidad del software (concretamente la mantenibilidad del software).

Los objetivos principales han sido detectar cuáles son las carencias de calidad más comunes entre los proyectos software, y si existen relaciones entre distintas métricas de mantenibilidad.

Para llevarlo a cabo, se ha tomado como muestra los 1000 proyectos mejor valorados por los usuarios de GitHub.

GitHub, que fue creado en 2008, es un servicio de almacenamiento software que ha alcanzado una gran popularidad en estos últimos años. Con 3 millones de usuarios[11] y 11.8 millones de repositorios[12], GitHub es uno de los servicios de alojamiento de código open-source más utilizados. Por otra parte, proyectos tan famosos como el kernel de Linux, MongoDB y Neo4j están alojados en GitHub.

Además, muchos de estos proyectos y librerías se están utilizando cada vez más como base para otros desarrollos.

Esto hace que GitHub sea una muy buena fuente para obtener proyectos reales con los que estudiar los problemas que afectan a la calidad del código.

[11] A fecha de 27 de marzo de 2014, se realizó en el buscador de Github la consulta "followers:">=0"" (para encontrar los usuarios con ningún o más de 0 seguidores, es decir, todos los usuarios registrados) que devolvió 3475618 usuarios.
[12] Esta información se ha obtenido de https://github.com/features

8.2 EL CASO DE ESTUDIO

Este caso de estudio se ha diseñado de acuerdo a las guías de casos de estudio presentadas por (Brereton et al., 2008).

Cuando hablamos de calidad, nos referimos a calidad de caja blanca. El principal objetivo es inspeccionar el código fuente y obtener métricas de mantenibilidad relacionadas con él.

Para poder recoger todos los datos necesarios para llevar a cabo el estudio, se han utilizado varias herramientas:

- GitHub, donde están alojados los proyectos del experimento.
- Una librería que permite acceder a la API de GitHub a través de Java.
- Git, para clonar esos proyectos en disco y poder analizarlos posteriormente.
- PMD y Simian, herramientas que sirven para analizar el código de los proyectos y obtener informes de distintas métricas de calidad.
- Una serie de scripts que llevan a cabo todo el proceso.

Mientras que Simian es una herramienta que detecta el código duplicado para cualquier lenguaje de programación, PMD es exclusivo para Java.

Por otra parte, PMD obtiene más variedad de métricas, que se agrupan en conjuntos, como por ejemplo tamaño de código, comentarios, diseño etc.

Para el caso de estudio, se han elegido las reglas más representativas de cada conjunto de reglas, que influyen en la mantenibilidad del software. Éstas pueden verse en la Tabla 2.

Simian	
	Duplicated code
	Lines of code
Code size	**CyclomaticComplexity**
	TooManyMethods
Comments	**CommentSize**
	CommentRequired
	CommentContent
Coupling	**CouplingBetweenObjects**
	ExcessiveImports
	LooseCoupling
	LawOfDemeter

Design	UseUtilityClass
	SwitchDensity
	AvoidDeeplyNestedIfStmts
	AbstractClassWithoutAbstractMethod
	TooFewBranchesForASwitchStatement
	CompareObjectsWithEquals
Empty code	EmptyCatchBlock
	EmptyIfStmt
	EmptyWhileStmt
	EmptyTryBlock
	EmptyFinallyBlock
	EmptySwitchStatements
	EmptySynchronizedBlock
	EmptyStatementNotInLoop
	EmptyInitializer

	EmptyStatementBlock
	EmptyStaticInitializer
JUnit	JUnitStaticSuite
	JUnitSpelling
	JUnitAssertionsShouldIncludeMessage
	JUnitTestsShouldIncludeAssert
	TestClassWithoutTestCases
	UnnecessaryBooleanAssertion
	UseAssertEqualsInsteadOfAssertTrue
	UseAssertSameInsteadOfAssertTrue
	UseAssertNullInsteadOfAssertTrue
	SimplifyBooleanAssertion
	JUnitTestContainsTooManyAsserts
	UseAssertTrueInsteadOfAssertEquals
Unused code	UnusedPrivateField

	UnusedLocalVariable
	UnusedPrivateMethod
	UnusedFormalParameter
	UnusedModifier

Tabla 2. Conjuntos de reglas y reglas de PMD analizadas en el caso de estudio.

8.3 ¿CÓMO SE HAN ELEGIDO LOS PROYECTOS ANALIZADOS?

El criterio para la selección de los proyectos del caso de estudio ha sido que estuvieran escritos en Java (lenguaje para el que actualmente hay multitud de herramientas de análisis de código, que obtienen métricas de todo tipo), públicos (accesibles a todo el mundo) y alojados en GitHub.

GitHub, permite acceder a la información que almacena de muy diversas formas. Como soporta Git, podemos clonar los repositorios alojados en él con esa herramienta y utilizar todos los comandos que proporciona.

Por otra parte, existe un API para poder comunicar nuestras aplicaciones con GitHub. Además, hay disponibles librerías para acceder al API con distintos lenguajes de programación, entre ellos Java.

8.4 ANALIZANDO LOS RESULTADOS

El tamaño de los proyectos estudiados oscila entre 10 líneas de código hasta casi 2 millones de líneas de código, con una media de 35180 líneas de código. Más de la mitad de los proyectos (59.94%) tienen un tamaño de entre 10 y 10000 líneas de código.

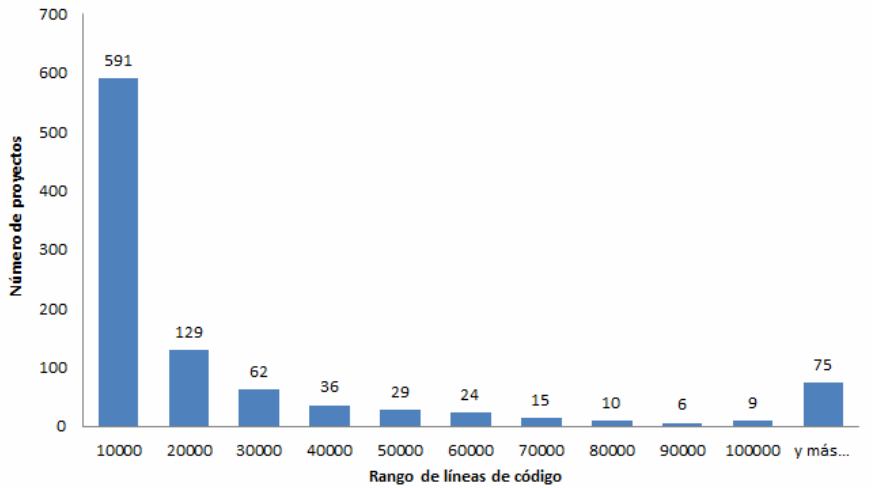


Figura 2. Distribución de los proyectos por su tamaño en líneas de código.

Veamos qué calidad tienen estos proyectos software.

8.4.1 CARENCIAS DE CALIDAD MÁS REPETIDAS ENTRE LOS PROYECTOS

Los resultados del caso de estudio reflejan que las reglas de calidad más infringidas entre los proyectos son las relacionadas con comentarios y acoplamiento.

Comentarios

En general, los proyectos de la muestra deberían mejorar la documentación de las APIs públicas (métodos y atributos públicos).

Este tema afecta por un lado a la mantenibilidad de los propios proyectos de GitHub, ya que si el código no se entiende bien y además está mal documentado, hará más difícil continuar su desarrollo.

Por otra parte, como los proyectos de GitHub se están utilizando cada vez más como base de otros desarrollos, si el código de estos proyectos está mal documentado, dificultará nuevamente el desarrollo de los nuevos proyectos.

Acoplamiento

Con respecto al acoplamiento, se ha encontrado un número muy alto de posibles violaciones de la Ley de Demeter.

La Ley de Demeter (LoD) (Lieberherr et al., 1987) es una regla de estilo para diseñar sistemas orientados a objetos. Esta regla se basa en que: “Solo hables con tus amigos”.

Según esta regla, un método de un objeto solo debe invocar los métodos de los siguientes objetos (Northeastern University, 1986):

1. Él mismo.
2. Los objetos que se pasan como parámetros del método.
3. Cualquier objeto que él crea.
4. Sus componentes directos.

Por ejemplo, no debería escribirse una cadena de mensajes de la forma objetoA.metodoDeA().metodoDeB().metodoDeC(); en otra clase distinta de la clase a.

Puede que pienses que algo así está bien, que tener muchas llamadas encadenadas de métodos es mucho más legible y ocupa menos líneas… Pero tiene sus problemas.

Cuando escribes código de este tipo, estás expuesto a los posibles cambios que puedan ocurrir en la clase del objetoA, del objetoB y del objetoC.

- ¿Qué pasa si el objetoA en el futuro cambia, y ya no necesita una referencia al objetoB?

- ¿Y si luego el objetoB ya no necesita una referencia al objetoC?

- Además, esta clase no es reutilizable, porque para usarla necesitas también las clases B y C.

JUnit

En cuanto a violaciones de JUnit, es muy frecuente que los proyectos de la muestra no utilicen mensajes en las aserciones de JUnit.

Una aserción (método assert) consiste en comprobar el resultado esperado del test con su resultado real. Suele ser una de las llamadas finales a la hora de crear una prueba unitaria en JUnit.

Es recomendable incluir un mensaje en las aserciones de JUnit, que se mostrará por pantalla en el caso de que la prueba falle.

Si no los incluimos nos costará más esfuerzo detectar por qué los casos de prueba no han pasado.

Código vacío

Otra de las carencias de calidad más destacadas es dejar bloques catch vacíos.

En este caso, muchos proyectos están capturando las excepciones del código, pero no las están tratando adecuadamente.

Esto puede ser un síntoma de querer silenciar los errores en lugar de detectar sus causas y solucionarlos.

8.4.2 RELACIONES ENTRE MÉTRICAS

Relación entre la complejidad ciclomática y las líneas de código.

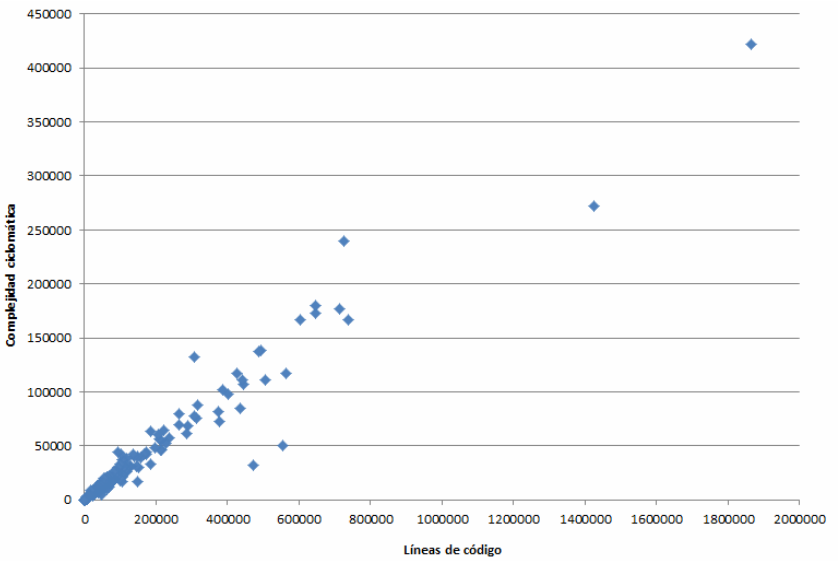


Figura 3. Relación entre la complejidad ciclomática y las líneas de código.

En este caso, existe una relación entre la complejidad ciclomática y las líneas de código. Esto indica que una gran cantidad de la complejidad ciclomática de la muestra proviene directamente del tamaño de los proyectos.

La complejidad ciclomática (McCabe, 1976) se basa en calcular el número de rutas linealmente independientes del código.

Por ejemplo, si el código fuente no contiene puntos en los que haya que decidir si tomar un camino u otro (sentencias if, swtich, bucles for etc.) la

complejidad ciclomática será 1, ya que solo hay un único camino en todo el código. En cambio, si por ejemplo hay un if con una única condición, la complejidad ciclomática será 2, ya que hay dos caminos en el código: el camino que se sigue cuando la condición es cierta, y el camino que se sigue cuando la condición es falsa.

La complejidad ciclomática se mide en los propios algoritmos, por lo que se puede calcular independientemente del lenguaje de programación que estemos tratando. Esta métrica permite apreciar la calidad del diseño software, de una manera rápida y con independencia del tamaño de la aplicación.

Normalmente, una complejidad ciclomática alta es un síntoma de un mal diseño del software.

Además, a medida que la complejidad ciclomática aumenta, el código es más difícil de entender (el programa se divide en varios caminos a tomar), con lo que la mantenibilidad se reduce.

También, se vuelve más difícil probar el software, ya que como el código tiene más caminos que seguir, aumenta el número de casos de prueba que habría que implementar para conseguir una cobertura de pruebas del 100%. Por ello, este código también es más propenso a contener errores.

Como se ha comentado, el algoritmo de complejidad ciclomática de McCabe se calcula con ciertos factores que crecen a medida que aumenta la cantidad de código, por lo que es normal que esta métrica esté relacionada con el tamaño del proyecto para la que está calculada.

Si un programador quisiera refactorizar el código, no tendría que fijarse solo si la complejidad ciclomática es muy alta. En su lugar, tendría que analizar por cuánto sobrepasa la complejidad ciclomática actual el valor esperado de complejidad para el número de líneas de código del módulo que se quiere refactorizar.

Relación entre el código duplicado y las líneas de código.

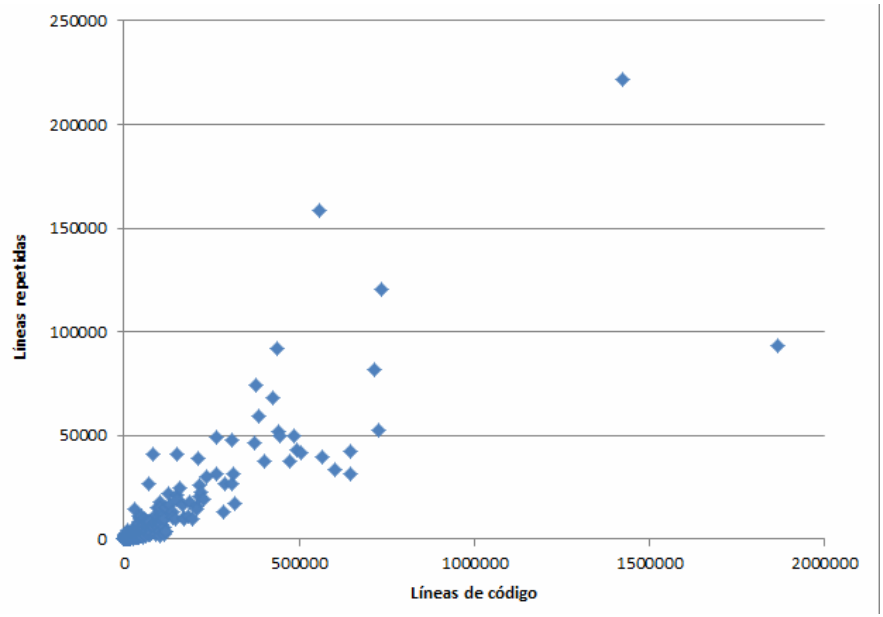


Figura 4. Relación entre las líneas de código y el código repetido.

Existe también una posible relación entre el número de líneas de código repetido y el tamaño del código fuente.

Sin embargo hay que recalcar que esto no implica que a mayor número de líneas de código mayor cantidad de código repetido, sino que, a medida

que el número de líneas de código aumenta, hay mayor probabilidad de tener más cantidad de código repetido.

8.4.3 RELACIÓN ENTRE LA COMPLEJIDAD CICLOMÁTICA Y EL CÓDIGO REPETIDO.

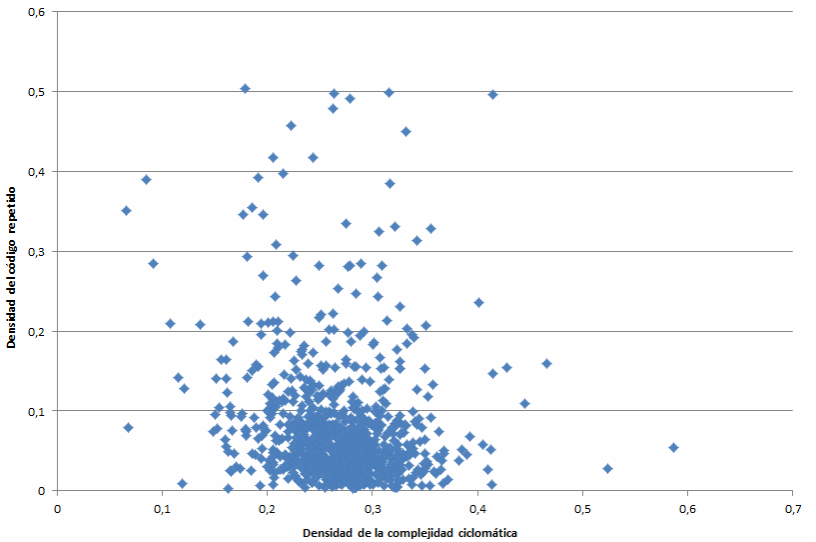


Figura 5. Relación entre la densidad de la complejidad ciclomática y la densidad de código repetido.

La densidad de la complejidad ciclomática, consiste en dividir la complejidad ciclomática entre el número de líneas de código. De la misma manera, la densidad de código repetido es la división del valor de código repetido entre el número de líneas de código.

Como vimos anteriormente, ya que como la complejidad ciclomática y las líneas de código tienen relación con las líneas de código del proyecto, debemos usar las densidades para compararlas entre sí.

Observando la gráfica, vemos que no hay relación entre las métricas de densidad de la complejidad ciclomática y densidad del código repetido. Por ello, un mayor valor de complejidad ciclomática no significa que aumente el código repetido.

En la muestra, las líneas de código repetidas, copiadas de un sitio y pegadas en otro, por lo general no son líneas que generan mucha complejidad ciclomática.

8.5 REFERENCIAS

Almossawi, A. (2013a). *Evolution of the Firefox codebase.* 2014. Disponible en: http://almossawi.com/firefox/

Almossawi, A. (2013b). *How maintainable is the Firefox codebase?* 2014. Disponible en: http://almossawi.com/firefox/prose/

Brereton, P., Kitchenham, B., Budgen, D., & Li, Z. (2008). Using a protocol template for case study planning. *Proceedings of Evaluation and Assessment in Software Engineering 2008,* , 1-8.

Coverity, I. (2013). *Annual Coverity scan report finds open source and proprietary software quality better than industry average for second consecutive year.* 2014. Disponible en: http://www.coverity.com/press-releases/annual-coverity-scan-report-finds-open-source-and-proprietary-software-quality-better-than-industry-average-for-second-consecutive-year/

Gómez, R. (2013). *How complex are TodoMVC implementations.* 2014. Disponible en: http://blog.coderstats.net/todomvc-complexity/

Lieberherr, K., IIolland, I., & Riel, A. (1987). Object-oriented programming. an objective sense of style.

McCabe, T. (1976). A complexity measure.

Northeastern University. (1986). *Object form of the law of demeter.*Disponible en: http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/object-formulation.html

Referencias

Ambler, S. (2008). *Acceleration: An agile productivity measure.* Fecha de consulta: 17 mayo 2012. Disponible en: https://www.ibm.com/developerworks/mydeveloperworks/blogs/ambler/entry/metric_acceleration?lang=en

Appleton, B. (2009). *Agile self-organizing teams.* Fecha de consulta: 17 enero 2014. Disponible en: http://bradapp.blogspot.com.es/2009/06/agile-self-organizing-teams.html

Austin, R. D. (2007). *CMM versus agile: Methodology wars in software Development* (Case study No. 9-607-084). Harvard, USA. Harvard Business School Press.

Beck, K. (1999). *Extreme programming explained: Embrace change*. USA. Addison-Wesley Professional.

Beck, K., et al. (2001). *Agile manifesto.* Fecha de consulta: 17 mayo 2012. Disponible en: http://agilemanifesto.org

Bran, S. (2009). Agile documentation, anyone? *IEEE Softw., 26*(6), 11-12.

Brooks. (1975). *The mythical man-month*. Addison-Wesley.

CMMI Product Team. (2010). In CMMI Product TeamEditors (Ed.), *CMMI for development version 1.3. improving processes for developing better products and services.* Pittsburg, IL, USA. Carnegie Mellon University.

Cockburn, A. (2002). *Agile software development*. Boston, MA, USA. Addison-Wesley Longman Publishing Co., Inc.

Cockburn, A. (2008). *Why I still use use cases.* Fecha de consulta: 17 mayo 2012. Disponible en: http://alistair.cockburn.us/Why+I+still+use+use+cases

Cohn, M. (2004). *User stories applied: For agile software development* Addison-Wesley.

Cohn, M. (2005). *Agile estimating and planning*. Upper Saddle River, NJ, USA. Prentice Hall PTR.

Cohn, M. (2009). *Bugs on the product backlog.* Fecha de consulta: 17 enero 2014. Disponible en: http://www.mountaingoatsoftware.com/blog/bugs-on-the-product-backlog

Doshi, H. (2010). *Template of task breakdown for a user story.* Fecha de consulta: 17 mayo 2012. Disponible en: http://www.practiceagile.com/2010/08/template-of-task-breakdown-for-user.html

Fowler, M. (2005). *The new methology.* Fecha de consulta: 17 mayo 2012. Disponible en: http://martinfowler.com/articles/newMethodology.html

Fowler, M. (2008). *Is design dead?* Fecha de consulta: 17 mayo 2012. Disponible en: http://martinfowler.com/articles/designDead.html

Garzás, J. (2010). *Desarrollo ágil o tradicional ¿existe el punto intermedio?.* Fecha de consulta: 17 mayo 2012. Disponible en: http://www.javiergarzas.com/2010/08/agiles-formales-e-hibridos.html

Garzás, J. (2011). *Dos razones por las que fabricar software no es lo mismo que fabricar coches o construir casas.* Fecha de consulta: 17 mayo 2012. Disponible en: http://www.javiergarzas.com/2011/02/diferencias-software-fabricacion-tradicional-1.html

Garzás, J., Irrazábal, E., & Santa Escolástica R. (2011, Guía práctica de supervivencia en una auditoría CMMI. *Boletín De La ETSII, Universidad Rey Juan Carlos, 002*, 1-33.

Garzás, J., & Paulk, M. (2012). Can scrum help to improve the project management process? A study of the relationships between scrum and project management process areas of CMMI-DEV 1.3. *Software Engineering Process Group,* Madrid. pp. 1-12.

Garzás, J. (2013). *Cómo sobrevivir... A la planificación de un proyecto ágil*. Madrid. 233gradosdeTI.

Glazer, H., Anderson, D., Anderson, D. J., Konrad, M., & Shrum, S. (2008). CMMI ® or agile : Why not embrace both! *IEEE Transactions on Geoscience and Remote Sensing, 33*(November), 48. Retrieved from http://www.sei.cmu.edu/library/abstracts/reports/08tn003.cfm

Haugen, N. C. (2006). An empirical study of using planning poker for user story estimation. Artículo presentado en *Agile Conference, 2006,* pp. 9.

Highsmith, J. (2001). *History: The agile manifesto.* Fecha de consulta: 17 mayo 2012. Disponible en: http://www.agilemanifesto.org/history.html

Hossain, E., Babar, M. A., & Hye-young Paik. (2009). Using scrum in global software development: A systematic literature review. Artículo presentado en *Fourth IEEE International Conference on Global Software Engineering, 2009.* pp. 175.

ISO. (2008). *ISO/IEC 12207:2008 systems and software engineering -- software life cycle processes.*

Jakobsen, C. R., & Johnson, K. A. (2008). Mature agile with a twist of CMMI. Artículo presentado en *Proceedings of the Agile 2008,* pp. 212-217. Disponible en: http://portal.acm.org/citation.cfm?id=1443221.1443497

Jakobsen, C. R., & Sutherland, J. (2009). Scrum and CMMI going from good to great. Artículo presentado en *Proceedings of the 2009 Agile Conference,* pp. 333-337.

Jeffries, R. (2001). *Essential XP: Card, conversation, confirmation.* Fecha de consulta: 17 mayo 2012. Disponible en: http://xprogramming.com/articles/expcardconversationconfirmation/

Kniberg, H. (2007). *Scrum and XP from the trenches*.

Kniberg, H. (2009). *Is your team cross-functional enough?* 2014. Disponible en: http://blog.crisp.se/2009/02/27/henrikkniberg/1235769840000

Kuphal, M. (2011). *Agile planning: My top five tips on decomposing user stories into tasks.*Disponible en: http://mkuphal.wordpress.com/2011/05/13/agile-planning-my-top-five-tips-on-decomposing-user-stories-into-tasks/

L. Kerth, N. (2001). Project retrospectives: A hand- book for team reviews.

Marcal, A., Furtado, F., & Belchior, A. (2007). Mapping CMMI project management process areas to SCRUM practices. Artículo presentado en *Proceedings of the 31st IEEE Software Engineering Workshop,* pp. 13-22.

McConnell, S. (2006). *Software estimation: Demystifying the black art*. Redmond, WA, USA. Microsoft Press.

Paulk, M. C. (2001). Extreme programming from a CMM perspective. *IEEE Software, 18*(6), 19-26.

Piattini, M., Manzano, J., Cervera, J., & Fernández, L. (2003). *Análisis y diseño de aplicaciones informáticas de gestión – una perspectiva de ingeniería del software* (2ª ed.). Madrid. Ra-Ma.

Pichler, R. (2010). *Agile product management with Scrum: Creating products that customers love*. Addison-Wesley Professional.

Potter, N., & Sakry, M. (2011). *Implementing Scrum (agile) and CMMI® together.* Fecha de consulta: 17 mayo 2012. Disponible en: http://www.agilejournal.com/articles/columns/column-articles/5794-implementing-scrum-agile-and-cmmi-together

Schwaber, K., & Sutherland, J. (2012). Software in 30 days.

Schwaber, K. (2004). *Agile project management with scrum* Microsoft Press.

Schwaber, K., & Sutherland, J. (2010). *Scrum guide.* Fecha de consulta: 17 mayo 2012. Disponible en: http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20-%20ES.pdf#view=fit

Schwaber, K., & Sutherland, J. (2013). *Scrum guide.* Fecha de consulta: 15 enero 2014. Disponible en: https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide-ES.pdf#zoom=100

Stacey, R. D. (2002). *Strategic management and organisational dynamics: The challenge of complexity* (3rd ed.) Prentice Hall.

Stellman, A. (2009). *Requirements 101: User stories vs use cases.* Fecha de consulta: 17 mayo 2012. Disponible en: http://www.stellman-greene.com/2009/05/03/requirements-101-user-stories-vs-use-cases/

Sutherland, J. (2001). Agile can scale: Inventing and reinventing Scrum in five companies. Cutter IT Journal *14*(12)

Sutherland, J. (2010). *Agile principles and values.* Fecha de consulta: 17 mayo 2012. Disponible en: http://msdn.microsoft.com/en-us/library/dd997578.aspx

Sutherland, J., Jakobsen, R., & Johnson, K. (2008). Scrum and CMMI level 5: The magic potion for code warriors. Artículo presentado en *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual,* pp. 466.

Sutherland, J., & Schwaber, K. (2011). *The scrum papers: Nut, bolts, and origins of an agile framework.* Fecha de consulta: 17 mayo 2012. Disponible en: http://jeffsutherland.com/ScrumPapers.pdf

Sutherland, J., & Johnson, K. (2010). *Using Scrum to avoid bad CMMI-DEV® implementation.* Fecha de consulta: 29 octubre 2010. Disponible en: http://aplndc.com/eventSlides/JeffSutherlandCMMI-DEV.pdf

Sutherland, J., Jakobsen, C. R., & Johnson, K. (2007). Scrum and CMMI level 5: The magic potion for code warriors. Artículo presentado en *AGILE 2007,* pp. 272-278.

VersionOne Inc. (2013). *State of agile survey 2013.* Fecha de consulta: 16 2014. Disponible en: http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf

Wake, B. (2003). *INVEST in good stories, and SMART tasks.* Fecha de consulta: 17 mayo 2012. Disponible en: http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

Las empresas en general, y en particular las TIC, incluidas las administraciones públicas, y por tanto sus profesionales, se enfrentan a un nuevo paradigma en los servicios que prestan a sus clientes o a los ciudadanos con la utilización de las últimas tecnologías.

La velocidad de aparición y de adopción de las nuevas tecnologías repercute en un aumento exponencial de las oportunidades de negocio, pero también de los riesgos en términos de seguridad, implantación y usabilidad de las mismas.

Las temáticas a abordar versarán sobre aspectos como ISO 9126, ESO 25000, Deuda técnica, Métricas, Integración continua, Agilidad, Auditorías, Externalización del Desarrollo, etc

Con esta iniciativa se propone un marco de reunión, debate y divulgación para investigadores, empresas, universidades y profesionales interesados en aumentar el valor que acercan a sus negocios, en base al control y optimización de la Calidad de los Productos que lo sustentan