SG #38 (Noviembre 2012 - Febrero 2013)

En esta edición de SG destacan los siguientes artículos:

- Testing de software ágil.
- Estudio de salarios 2012.
- Consideraciones para elegir a tus cofundadores.
- Crónica de una certificación fracasada de MoProSoft.
- Introducción a OpenStack.

Noticias SG38

Sección:

Noticias

Contacto Tractoras + PyMEs

Sección:

Reportaje

KUALI-BEH y ESSENCE

Sección:

Tejiendo nuestra red

Autor:

Hanna Oktaba

Lean y Desarrollo Ágil

Sección:

Mejora continua

Autor:

Luis Cuellar

Lo Que Viene

Sección:

Herramientas y Tecnologías

Bases de Datos En Memoria

Sección:

Tendencias en Software

Autor:
Luis Daniel Soto

Consideraciones para Elegir a Tus Cofundadores

Sección:
Emprendiendo
Autor:
Celeste North

Richard Stallman en México

Sección:
Entrevista

Marcando la Pauta para las Pruebas Ágiles

Sección:
Principal
Autor:
Berenice Ruiz

Pruebas y el Ciclo de Vida Ágil

Sección:
Principal
Autor:
Germán Domínguez

Agilizando las Pruebas de Desempeño

Sección:
Principal
Autor:
Federico Toledo

Test Driven Development

Sección:
Principal
Autor:
Alfredo Chavez

Pruebas Continuas

Sección:
Principal
Autor:
Christian Ramírez

Estudio de Salarios SG 2012

Sección:
Principal
Autor:
SG Software Guru

Aplicando Criterios Ágiles a la Calidad

Sección:
Código Innovare

Autor:
Jordana Villegas Sosa

Gustavo y Einstein

Sección:
Agilidad
Autor:
Masa K. Maeda

Conceptos de Diseño Patrones, Tácticas y Frameworks

Sección:
Arquitectura
Autor:
Humberto Cervantes

Crónica de una Certificación Fracasada de MoProSoft

Sección:
Procesos
Autor:
Alfredo Lozada Carrillo

El Extraño caso del Líder Jekyll y Mr. Hyde

Sección:
Gestión de Proyectos
Autor:
Efraín Cordero

El Testing de la Experiencia de Usuario

Sección:
Tecno-lógico
Autor:
Mauricio Angulo

Programación en la Escuela ¿Para qué?

Sección:
Programar es un Estilo de Vida
Autor:
Gunnar Wolf

De la Luna a Marte sin Cambiar de Equipo de Trabajo

Sección:
Carrera
Autor:
Francisco Estrada Salinas

OpenStack

Sección:
Infraestructura
Autor:
Por Pedro Galván y Juan Cáceres

El Gigante Invisible

Sección:
Columna invitada
Autor:
Pedro Soldado

Gadgets - Noviembre 2012

Sección:
Gadgets

Noticias SG38

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Noticias

SoftMty 2012

El pasado 15 y 16 de Octubre se llevó a cabo SoftMty 2012, el evento insignia del Consejo de Software de Nuevo León. SoftMty reunió a ejecutivos de TI con el propósito de conocer tendencias y estrategias para lidiar con las principales necesidades de las áreas modernas de TI. Entre los temas que se abordaron estuvieron: análisis del presupuesto de las áreas de TI corporativas, cómputo empresarial en los corporativos, y manejo de contratos de outsourcing. SoftMty se dió el lujo de reunir a conferencistas internacionales de las firmas de análisis e investigación más prestigiadas del mundo como Gartner, Forrester y Cutter Consortium. Adicionalmente participaron como panelistas los CIOs de organizaciones tales como Comercial Mexicana, Coppel, CostCo, Gas Natural, Alestra, entre otros.

Hackathon por la Transparencia

A pesar de que ya se han realizado eventos en nuestro país para impulsar "datos abiertos", lo relevante del hackathon realizado el pasado 19 y 20 de octubre es que fue una iniciativa de la misma Secretaría de Función Pública (SFP), a la que se unieron otras organizaciones como INFOTEC, Software Guru, Mapdata, Social TIC y Revista Política Digital. La app ganadora fue un servicio para localizar centros de salud cercanos. El segundo lugar se lo llevó la versión móvil del sitio "Tu Gobierno en Mapas" y el tercer lugar se lo llevó una aplicación para evitar que pases por lugares con alto índice delictivo.

Wowzapp México

Con una participación de más 17,000 desarrolladores alrededor del mundo, Microsoft organizó el pasado 9 y 10 de noviembre el hackathon global WOWZAPP 2012 para crear aplicaciones de Windows 8. Software Guru colaboró para convocar y organizar la sede de Wowzapp en Ciudad de México, donde se reunió a más de 140 desarrolladores. Sin duda, Microsoft y SG sentaron precedente en nuestro país.

Gartner CIO & IT Executive Summit 2012

Del 9 al 11 de octubre pasado, Centro Banamex Ciudad de México recibió a más de 300 personas especialistas, profesionales y estrategas de las Tecnologías de la Información en América Latina que durante dos días y medio accionaron sus habilidades de business networking, además de recibir insights, tips de estrategia e innovación así como tendencias en voz de los mejores analistas de Gartner que enfocaron su experiencia en cuatro tracks especializados: el CIO y el ejecutivo de TI, innovación de la información, aplicaciones empresariales e infraestructura y operaciones. *Crédito: 360° Contenidos on Demand / Susana Tamayo*

Talento Software

Por primera vez se realizó el Congreso "Talento Software" en la ciudad de Tehuacán, Puebla organizado por la empresa Link colectivo el 15 y 16 de noviembre. Talento Software contó con conferencistas nacionales e internacionales que expusieron tendencias y prácticas para que los profesionistas de software aprovechen las oportunidades que brinda nuestra industria. Como parte del evento se realizó también una Feria de Reclutamiento de Talentos, la cual dió la oportunidad a estudiantes y profesionistas de participar como aspirantes con empresas reclutadoras nacionales e internacionales.

Contacto Tractoras + PyMEs

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Reportaje

Gracias al apoyo de la Secretaría de Economía, a través del programa Fondo PyME, el pasado mes de octubre Software Guru llevó a cabo la segunda edición del evento Contacto Empresas Tractoras+PyMEs.

El evento reunió a grandes empresas en busca de productos y servicios de TI (empresas Tractoras), con empresas pequeñas y medianas (PyMEs) especializadas en dichos servicios, interesadas en iniciar una relación comercial ya sea como proveedores o como aliados.

Gracias a la coordinación de esfuerzos el evento logró:

- La participación de 32 empresas Tractoras
- 350 PyMEs participantes
- Más de 900 encuentros de negocio
- Más de 500 asistentes

Durante el evento se llevaron a cabo encuentros presenciales entre empresas Tractoras y PyMEs, así como encuentros virtuales con empresas Tractoras localizadas en diferentes puntos del país.

Entre la demanda de las empresas Tractoras encontramos: prueba de aplicaciones, desarrollo móvil, soporte técnico, capacitación, consultoría, aplicaciones propietarias, seguridad en TI, PaaS, IaaS, SaaS, cloud computing, y los que no pueden faltar desarrollo a la medida y outsourcing. También se buscaron aliados para integración de aplicaciones, fábrica de software, venta de hardware y software, implementación de BI e implementación de Big Data.

Entre los lenguajes y plataformas más comunes solicitados podemos mencionar: .Net, PHP, HTML, CSS, Javascript, C++, Ajax, Python, JAVA, SQL, ORACLE, VWARE, CITRIX, TAMDEN B24, Androide y iOS.

Agradecemos a las empresas Tractoras por su valiosa participación en este evento: AXA, Buzón E, Centro Escolar Cedros, Coca-Cola FEMSA, CompuSoluciones, COSTCO, Dell, EBP Software, ERP Software, Estafeta, Excelerate Systems, Farmacias del Ahorro, GOBTEC, Grupo IAMSA, Grupo La Europea,

Hildebrando, HP, IBM, ITKMAS, Logiti, Microsoft, N&S Trantor, Palacio de Hierro, Praxis, Renault, RIM, SAFENET, SAP, Secretaría del Trabajo y Previsión Social, T-Systems, Turrent Cameron y Ultrasist.

Agradecemos a ProMéxico, México First, y a la Cámara Nacional de la Industria de Transformación (Canacintra), por su participación como expositores durante el evento. De igual manera agredecemos a: AISAC, AMITI, CITI Tabasco, ClusterTIM, Csoftmty, CLUSTEC, e InteQsoft, por su apoyo en la difusión del evento.

SG continuará apoyando el desarrollo y sinergia entre empresas de TI, facilitando foros de encuentro y organizando diferentes actividades. Espera noticias.

KUALI-BEH y ESSENCE

Autor:

Hanna Oktaba

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Tejiendo nuestra red

En SG #36 les conté cómo llegamos a competir con Ivar Jacobson y sus aliados en una convocatoria del Object Management Group (OMG) que buscaba propuestas para "A Foundation for the Agile Creation and Enactment of Software Engineering Methods (FACESEM)". En febrero de este año enviamos a nombre de la UNAM (Miguel Ehécatl Morales Trujillo, Magdalena Dávila Muñoz y yo) la propuesta bajo el nombre de KUALI-BEH Software Project Common Concepts, mientras que el grupo de Jacobson presentó la propuesta de ESSENCE.

En marzo de este año, presenté a mi "hija" menor KUALI-BEH (sus "hermanos" mayores se llaman MoProSoft, Competisoft e ISO/IEC 29110-5-1-2 Perfil básico) en una reunión técnica de OMG. A su vez Jacobson presentó a su "hijo" ESSENCE (los "hijos" mayores de Ivar son muchos, los más destacados se llaman Casos de Uso, UML y Proceso Unificado). Desde que KUALI-BEH fue presentado nos dimos cuenta que a Ivar le pareció ser una competencia seria. Inmediatamente después de la reunión empezó a persuadirnos de que lo mejor para nuestro grupo era unirse a su propuesta y, a lo mejor, algunas pequeñas ideas de nuestra propuesta se agregarían a la suya; en otras palabras, pretendió desaparecer a KUALI-BEH. Esto no nos pareció justo, estábamos seguros de que nuestra propuesta tenía valor.

El 13 de agosto fue la siguiente fecha para la entrega ante OMG de las versiones mejoradas de las propuestas. A pesar de las presiones, decidimos mantener KUALI-BEH por separado. La fuerza para tomar esa decisión surgió del resultado del Taller Colaborativo en Métodos de Ingeniería de Software en el cual comprobamos que las ideas de KUALI-BEH son relativamente fáciles de asimilar. Además, recibimos una retroalimentación con 93 sugerencias, las cuales se analizaron, y 64 de ellas fueron consideradas para incorporarlas en la nueva

versión. Agradecemos a DGTIC-UNAM, Magnabyte y JPE Consultores por su valiosa participación en este experimento entre la academia y la industria.

Cuando entregamos la versión 1.1 de KUALI-BEH en agosto, la insistencia de Jacobson para que nos fusionáramos volvió y se incrementó. En las negociaciones entre ambas partes intervino el Dr. Carlos Mario Zapata Jaramillo, profesor de la Universidad Nacional de Colombia y actual presidente del capítulo latinoamericano del SEMAT. Carlos Mario analizó ambas propuestas y propuso ideas para la fusión que preservaban las aportaciones de KUALI-BEH. Sus ideas fueron tomadas en cuenta por ambas partes y se llegó al acuerdo unos días antes de la segunda presentación ante el comité técnico de OMG el 12 de septiembre en Florida. En esta ocasión, por nuestra parte sólo pudo asistir Miguel, le tocaron las últimas negociaciones para armar y hacer una presentación conjunta. Así inició el noviazgo formal entre KUALI-BEH y ESSENCE. El presidente de la OMG Richard Soley y el propio Ivar Jacobson expresaron su júbilo.

Ahora nos tocan los preparativos de la "boda" prevista para el 12 de noviembre. Tenemos que integrar las dos propuestas en un sólo documento. Como en todos los preparativos de las bodas hay que conciliar los egos y los "usos y costumbres" de ambas partes. Nosotros estamos dispuestos a hacer todo para lograrlo y tratamos de ser flexibles para que esta fusión sea exitosa.

Cuando escribo esta columna falta un mes para la "boda". Si logramos integrarnos, el documento final llevará el apellido ESSENCE del "esposo", como en matrimonios polacos. Posteriormente será revisado y calificado por un Comité Evaluador. Curiosamente, se nos invitó a proponer candidatos para este comité y me imagino que lo mismo hará el grupo de Ivar. La decisión final de volver esta propuesta conjunta un estándar de OMG se tomará a través de la votación de los miembros nivel Platform de la OMG. El resultado se hará público el 10 de diciembre. Hasta este momento sabremos si va a haber "luna de miel".

Esta "boda", como todas, cuesta. La UNAM tiene que renovar en OMG su membresía platform (5,500 USD) para poder seguir participando y votar. La asistencia a las reuniones técnicas de OMG tampoco sale gratis. Por lo tanto vuelvo a invitar a todos mis lectores a que visiten el sitio http://www.kuali-kaans.mxpara conocer como pueden apoyar y patrocinar este proyecto y de paso descargar el documento de KUALI-BEH 1.1 y llegar al sitio de estándares ISO, que son gratuitos. Por cierto, de este sitio pueden también descargar ISO/IEC 29110-5-1-2 Basic profile y recién salido del horno ISO/IEC29110-5-1-1 Entry profile. Este último dirigido a grupos de desarrollo muy pequeños (1 a 6 personas) y startups. Pueden encontrar el documento de ESSENCE en el sitio de SEMAT.

Figura 1. Miguel e Ivar al sellarse el acuerdo.

Bio:

La Dra. Hanna Oktaba es profesora de la UNAM, miembro del IPRC, y directora técnica del proyecto COMPATISOFT.

Lean y Desarrollo Ágil

Autor:

Luis Cuellar

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Mejora continua

Recientemente me preguntaban si las metodologías ágiles son la respuesta Lean al desarrollo de proyectos. La idea es interesante; yo estoy convencido de que podemos utilizar herramientas de calidad en muchas partes, tanto de la administración como ingeniería del desarrollo de software, así que exploremos más esta idea.

La metodología Lean fue desarrollada por Toyota en los noventas, como una práctica de producción que considera como desperdicio el uso de recursos en actividades que no generen valor para el cliente, definiendo valor como todo aquello que el cliente esté dispuesto a pagar. Lean conlleva una serie de herramientas diseñadas para eliminar este desperdicio, entre ellas el mapeo de valor, las cinco S, Kanban (o proceso de jalar en lugar de empujar) y Poka Yoka (desarrollo a prueba de errores).

El modelo ágil de desarrollo de software se formalizó en el 2001 con el Manifiesto Ágil, aunque tiene raíces desde mucho antes. Este modelo busca la agilidad a través de valorar la interacción entre las personas, la colaboración y la flexibilidad al cambio. El manifiesto define doce principios enfocados a: satisfacción del cliente a través de desarrollo rápido; flexibilidad al cambio; entrega continua de software; entrega es la base del avance; flujo continuo a través de un paso constante; cooperación continua entre participantes; comunicación frente a frente; confianza en los participantes; búsqueda de excelencia; simplicidad; auto-organización; adaptación a circunstancias de cambio.

A primera vista, el manifiesto promueve todos los principios de Lean: establece un modelo de Kanban, donde se tiene un paso continuo; una estructura en donde el cliente rige cada iteración, que puede ajustarse continuamente; estipula también una reducción de documentación y la cercanía de los participantes para poder asegurar una mejor comunicación.

Lean se creó como una herramienta de manufactura, por lo que requiere ser adaptada para ser utilizada en servicios. La principal adaptación que sufre es la modificación de los elementos que se consideran desperdicio. A continuación explico como Ágil ataca los problemas de desperdicio.

Desconocimiento de expectativas del cliente. Ágil ataca este problema al generar entregables continuos, y al tomar como medida de avance el software funcionando. Esto establece en forma indirecta que la participación del cliente siempre está relacionada con avanzar.
Energía y duplicidad de trabajo. Aunque Ágil fomenta el generar trabajo a través de pares, esto no va contra este principio porque la idea detrás de esto es que cada individuo busque defectos desde un punto de vista diferente, minimizando así el re-trabajo.
Errores de comunicación. Ágil fomenta la comunicación verbal como una forma eficiente de obtener claridad.

Inventario incorrecto. A diferencia de metodologías que miden el avance de un proyecto en base a actividades realizadas, en Ágil el avance se mide en términos de "software funcionando".
Errores en el servicio. Lean fomenta un retroalimentación continua y la detección de errores lo antes posible a través de revisión y retroalimentación continua.
Potencial humano no utilizado. Lean fomenta la participación de todos en todas las etapas de desarrollo y la distribución de roles en forma grupal.

En base a esta rápida comparación, definitivamente Ágil se puede considerar como una forma Lean para desarrollar software y que por lo tanto acarrea sus beneficios.

Tips para la adopción

Un inconveniente de Ágil es que al ser tan flexible, las implementaciones pueden variar considerablemente, permitiendo mucho uso inapropiado del modelo. Por ejemplo, recordemos que Lean considera valor agregado todo aquello que está dispuesto a pagar el cliente, por lo que si el cliente paga por tener documentación de su sistema, eso es parte de los entregables.

Otra situación común es que los equipos decidan eliminar actividades como reuniones periódicas de retroalimentación o programación en pares, las cuales tienen una razón de ser; eliminarlas sin cuestionar cómo sustituimos las prácticas equivale a remover acciones de valor agregado del modelo.

Por otro lado, no debemos ser tan rígidos, ya que seguramente nos encontraremos con situaciones especiales donde ya sea por tamaño de proyecto o dispersión geográfica, es posible sea necesario aplicar prácticas de metodologías no tan ágiles.

En resumen, definitivamente las metodologías ágiles son una excelente solución al desarrollo de software pero como todo en esta vida tenemos que cuestionar la razón de las actividades que vamos a llevar a cabo, y más importante, cuestionar la razón de las actividades que decidimos dejar de implementar

Bio:

Luis R. Cuellar es director de calidad a nivel mundial de Softtek. Es reconocido por la ASQ como Certified Quality Manager,Certified Software Engineer y Six Sigma Black Belt. @lcuellar

Lo Que Viene

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Herramientas y Tecnologías

Icenium

Olvídate del IDE, lo de hoy es el ICE

Icenium, denominado el primer Integrated Cloud Environment (ICE), es un ambiente de desarrollo móvil cross-plataform que combina el poder y flexibilidad de la nube con la conveniencia de un ambiente de desarrollo local. ¿Qué significa esto de combinar la nube con local? En el lado del cliente, Icenium expone un ambiente de desarrollo que puede ser una aplicación desktop (Mac o Windows) o un cliente web accesible desde cualquier navegador. Por otro lado, en la nube, Icenium expone los SDKs de Android e iOS. Es así que desde tu navegador puedes editar el código de tu app, simular su ejecución y mandar a compilar aplicaciones nativas (iOS y Android) que se compilan en la nube y que puedes descargar fácilmente cargar en tu teléfono simplemente direccionando a un QR code que Icenium te muestra en pantalla. Entonces, con Icenium no necesitas una Mac ni XCode para crear una aplicación para iOS. La programación en Icenium se hace usando HTML5, CSS y Javascript. Icenium utiliza las tecnologías Kendo UI y Apache Cordova (antes PhoneGap) para implementar el comportamiento de las apps y brindar acceso a capacidades como la cámara, acelerómetro y GPS.

Más info en http://icenium.com

—----------

RethinkDB

NoSQL sencillo y sin limitaciones

RethinkDB es una base de datos tipo NoSQL que a grandes rasgos pretende brindar una simplicidad comparable a la de MongoDB pero sin sus limitaciones.

Entre sus capacidades están:

- Modelo de datos JSON
- Lenguaje de búsquedas intuitivo pero poderoso que soporta opciones complejas como table joins, subqueries, group by.
- Map/reduce al estilo Hadoop.
- Interfaz de administración amigable

- Escalabilidad horizontal a un gran número de nodos.
- Búsquedas paralelizadas automáticamente
- Análisis de datos sin candados (lock-free) gracias a concurrencia tipo MVCC
  Más información en http://rethinkdb.com

—----------

Azure HDInsight

Hadoop para las masas

Como habíamos comentado anteriormente, Microsoft ha estado trabajando en ofrecer su versión de Hadoop para brindar capacidades de análisis de Big Data. HDInsight es la distribución Hadoop de Microsoft 100% compatible con Apache. HDInsight habilita a las organizaciones a analizar datos no estructurados al mismo tiempo que ofrece conectividad con las herramientas de Business Intelligence más populares. Aplicando una estrategia "muy Microsoft", con HDInsight la empresa de las ventanas busca llevar el análisis de big data a un espectro amplio de usuarios, quienes podrán realizarlo desde herramientas familiares como por ejemplo Excel, PowerPivot, y Powerview. Así es, Hadoop desde Excel. HDInsight está disponible en dos modalides: como un software instalable en Windows Server, o como un servicio en Windows Azure.

Bases de Datos En Memoria

Autor:

Luis Daniel Soto

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Tendencias en Software

Los elementos fundamentales en el diseño de computadoras están cambiando, en particular el costo y aplicación de la memoria RAM. Esto nos da la oportunidad de reinventar la plataforma e infraestructura tecnológica.

La actual generación de sistemas de bases de datos relacional (RDBMS) está optimizada para almacenamiento en disco duro. Las bases de datos "en memoria" (*in-memory*) son un tema naciente, pero que cobrará gran popularidad durante los próximos años.

Factores clave

Velocidad. La ventaja principal de almacenar todo en memoria RAM es la velocidad. La velocidad de acceso a un disco duro ronda los 5 milisegundos mientras que en el RAM es de 80 nanosegundos, es decir una diferencia de cerca de 100,000 veces. Aún utilizando discos de estado sólido y memoria FLASH no-volátil, que es 100 veces más rápida que los discos duros tradicionales, estaríamos 1,000 veces más lentos que usando RAM.

Durabilidad. Pensar en la memoria como único almacenamiento genera la gran pregunta de qué es lo que sucede en caso de pérdida de poder. Para resolver esto, cada determinado tiempo (pocos minutos), las páginas de memoria RAM se escriben en almacenamiento no-volátil. Adicionalmente, las "transacciones" no se consideran completas sin que se haga permanente una bitácora (*log*) de la operación realizada. Es así que en caso de falla, se recupera la página más recientemente almacenada y se vuelven a aplicar las transacciones desde la bitácora.

OLAP En Memoria

Hoy las bases de datos se pueden optimizar para "procesamiento de transacciones" (OLTP) con almacenamiento en renglones o "procesamiento analítico" (OLAP) con almacenamiento columnar.

Las bases de datos transaccionales no son apropiadas para efectuar análisis, requieren índices. Una variación del modelo relacional son las estructuras multidimensionales que organizan datos expresando las relaciones entre los mismos. OLAP permite responder preguntas en 0.1% del tiempo requerido en ambientes transaccionales. La compresión es la máxima en este modelo, permitiendo almacenar "en memoria" grandes tablas, frecuente con un radio 1:10.

Hasta el día de hoy no existe ninguna estructura que permita indistintamente ser eficiente tanto en OLAP como OLTP —aunque algunos fabricantes lo estén publicitando así—. Lo mejor que se puede hacer hasta ahora en las bases de datos más modernas es especificar a nivel de tabla si se quiere almacenamiento en renglones o columnas.

Utilizar bases de datos "en memoria" permite reducir dramáticamente la cantidad de cubos pre-calculados a construir, porque es virtualmente instantáneo calcular cualquier consulta. Gracias a esto, podemos vislumbrar la eliminación completa de los cubos de información.

Un sistema "100% puro" en memoria puede ser diseñado para no depender absolutamente de las estructuras necesarias para operar con discos duros. Pero debido a la explosión de datos la mayoría de los sistemas serán "optimizados" para "en memoria" y se conectarán con bodegas de datos tradicionales o sistemas archivos distribuidos (v.gr. HDFS de hadoop) con información histórica.

La nueva inteligencia de negocio

Las nuevas herramientas de autoservicio en BI permiten que cualquier usuario pueda descubrir y analizar información empresarial. La expectativa es que el análisis de información se haga en tiempo real.

La clave para habilitar análisis instantáneo en un sistema en operación es reducir dramáticamente el tiempo requerido para transferir información OLTP a su forma OLAP. Hasta recientemente, este era un proceso muy tardado denominado "Extract, Transform, Load" (ETL). Pero en las bases de datos "en memoria" es posible sincronizar ambas estructuras con replicación basada en bitácoras.

Procesamiento complejo de eventos

Otro escenario de uso donde las bases de datos en memoria brindan una solución ideal, es en sistemas donde se reciben cantidades masivas de datos pero solo se requiere almacenar los cambios en éstos. Un ejemplo es monitorear millones de sensores de temperatura pero solo registrar aquellos en ciertos rangos, o con cierta variación. Conforme vaya aumentando la cantidad de dispositivos y sensores conectados al Internet que generan datos continuamente —lo que llaman el "Internet de las cosas"—, este será un escenario cada vez más común.

OLTP En Memoria

Con la versión 2012 de SQL Server, Microsoft liberó una tecnología llamada xVelocity, que provee capacidades "en memoria" para bases de datos OLAP. En la próxima versión de SQL Server, nombre clave Hekaton, xVelocity también se podrá utilizar en bases de datos OLTP. Esta tecnología consiste en "convertir" una tabla de SQL Server en una nueva tabla "en memoria" y en "compilar" a código máquina algunos procedimientos almacenados. El beneficio fundamental es que esto no requiere rediseñar la base de datos o aplicaciones, mientras que otras iniciativas de industria requieren una nueva arquitectura.

Esto acelerará dramáticamente el desempeño de los sistemas transaccionales. Como ejemplo, un sistema de venta de boletos que hoy requiere de 5 servidores para atender todas las peticiones en un tiempo aceptable, podrá ser reducido a uno solo. Esto simplifica dramáticamente realizar operaciones como cambio de precio de un producto, que se tiene que propagar de manera muy compleja en granjas de servidores. O permite obtener la escala de sistemas enormes en sistemas medianos.

Más allá de las mejoras en escalabilidad, xVelocity for OLTP permite vislumbrar escenarios totalmente nuevos en el mundo de base de datos. Algunos ejemplos:

- "Procesamiento" de streams de datos. En el servicio de búsqueda Bing, la información se pre-procesa para ser colapsada y posteriormente almacenada; un "big data" puro sería demasiado costoso.
- Cache. Una base de datos en memoria se utilizará para absorber stream de datos. Piense en un desastre natural y miles de llamadas telefónicas que generan rastros digitales
- Baja latencia. Por ejemplo, el sistema del NASDAQ requiere una latencia menor a 0.5 milisegundos. Debido a la eliminación de contenciones en la tabla, esta métrica se puede cumplir sin soluciones especializadas con hardware disponible comercialmente.
- Lectura de datos en escala masiva. Actualmente requerido por soluciones analíticas avanzadas o sistemas de juegos en línea.

Conclusión

Una estrategia completa "en memoria" deberá cubrir todos los tipos de carga (OLTP, OLAP, Procesamiento de eventos), estar disponible para PC, servidor o granjas de servidores. También debe poder ser adquirido en forma de software, como servicio o como appliance. Bienvenido a la nueva era de analíticos de grandes datos donde se acortan los ciclos para contestar a las interrogantes del negocio.

Bio:

Luis Daniel Soto Maldonado (@luisdans) labora en la división de negocio de servidores y herramientas de Microsoft Corp.

Consideraciones para Elegir a Tus Cofundadores

Autor:

Celeste North

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Emprendiendo

Según Paul Graham, uno de los inversionistas más exitosos de Silicon Valley, dos causas comunes para el fracaso de un startup están relacionadas con los fundadores del proyecto: ser un fundador único (es decir no tener otros cofundadores) y peleas entre los cofundadores. ¿Cómo podemos evitar caer en estos errores? A continuación comparto algunas consideraciones.

El equipo correcto es más importante que el producto o el mercado. Para muchas personas pensar en con quién iniciar un startup puede tener poca importancia, sin embargo es uno de los pilares de cualquier proyecto. Tanto inversionistas como aceleradoras te dirán que a fin de cuentas, en quien se invierte es en las

personas. Siempre será más sencillo ajustar tu estrategia, idea o modelo de negocio que encontrar nuevos cofundadores.

Conocerse desde antes es básico, tener historia juntos es mejor. En un startup se comparten momentos de incertidumbre y alegría por igual. Saber cómo reaccionan los demás ante el estrés, el miedo, la desesperación o el cansancio es imprescindible para saber si podrán trabajar juntos. Si las diferencias de actitud o formas de pensar son irreconciliables entre ustedes, se estarán peleando de manera constante, convirtiéndose en un problema mayor que los obstáculos mismos del proyecto.

Objetivos y estilo de vida alineados. Una de las causas más comunes de conflicto entre fundadores (y fracaso del startup) es no tener objetivos de vida similares. Iniciar un startup requiere mucho trabajo, esto significa trabajar una gran cantidad de horas, tener menos dinero para gastar y dedicar gran cantidad de tu espacio mental a pensar en el proyecto. Si alguno de los fundadores tiene un estilo de vida que le demanda tiempo o ingresos de manera muy desigual para los otros fundadores, es muy posible que se generen problemas.

Complementen sus habilidades y áreas de experiencia. Independientemente de la cantidad de cofundadores, lo más importante es que se complementen en sus áreas de trabajo. En mi experiencia, lo mejor es un equipo constituido por un programador, un diseñador y una persona de negocios; de esta manera avanzan en paralelo en las tres partes fundamentales del producto.

Es importante tener confianza, pero también hay que documentar. Es muy posible que durante el camino alguien deje el equipo, por lo que es importante documentar todo. Con documentar me refiero a: tener acceso a todos los archivos, servidores y recursos de negocio generados; llevar alguna documentación (aunque sea informal) de procesos y tecnologías; y por último firmar entre ustedes un documento donde se establezca qué pasará si alguno de los fundadores deja el equipo ¿Tendrá derecho a un porcentaje de la empresa? ¿Qué procedimientos se harán para la entrega?

Encuentren la forma de divertirse juntos. Busquen tiempo para platicar, para pasar tiempo de esparcimiento juntos y recordar por qué son el mejor equipo para llevar a cabo el proyecto. Celebren las pequeñas victorias.

Espero que consideres estos puntos al momento de llevar a cabo un nuevo proyecto. Te facilitarán el camino para tener un startup que avanza, aprende y construye rápidamente. Aunque como todos los consejos bien intencionados, siempre es más fácil ofrecerlos que seguirlos.

Bio:

Celeste North (@celestenorth) es fundadora de NuFlick, plataforma de distribución de cine independiente y festivales de cine on demand enfocado en el mercado Latinoamericano. Organiza Founder Friday, un evento para fomentar el emprendimiento entre mujeres en la Ciudad de México. Es colaboradora editorial sobre temas de innovación y emprendimiento en Opinno, Emprendela y Software Guru.

Richard Stallman en México

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Entrevista

Richard Stallman, líder fundador de la Free Software Fundation y mundialmente conocido por el establecimiento de un marco de referencia moral, político y legal para el movimiento del software libre, estuvo en México durante el pasado mes de octubre para dar una "gira de conferencias". Durante el Encuentro Nacional de Software Libre (ENLi) en Puebla tuvimos oportunidad de asistir a su conferencia y estar presentes durante la ronda de preguntas y respuestas. Compartimos aquí un breve resumen.

Como era de esperarse, su arribo al auditorio fue bastante original, ya que entró diciendo "traigo pegantinas, tomen una", así que todo el auditorio se abalanzó sobre las "pegantinas" (estampas) de logos de su fundación y frases de software libre. Cuando ya tenía a casi todo el auditorio de pie cerca de él, entonces sacó otras estampas y pines diciendo "traigo más cosas, pero éstas las vendo".

Ya una vez terminada la venta matutina, comenzó su conferencia. El tema central de la conferencia fue la riqueza que brinda el conocimiento y cómo es un derecho para todos los seres humanos tener acceso a este sin restricciones. Recalcó que la importancia del software libre no es en relación al precio, sino a la disposición del conocimiento y a la disposición del código para que el software pueda mejorarse. "Las clases son espacios para compartir conocimiento y es en donde se debe de fomentar la libertad, es por ello que las universidades deben de enseñar con software libre", comentó.

Entrevista

¿Qué sugieres que hagan los estudiantes para convencer a sus universidades de que enseñen con software libre?

Sugiero que se manifiesten, que exijan porque tienen el derecho de hacerlo. El software libre es un movimiento que necesita acción, y es responsabilidad de los estudiantes velar por sus libertades.

¿Has hablado con alguien del próximo gobierno de México sobre asuntos de libertad tecnológica?

Tuve reuniones con senadores y diputados. Les sugerí políticas del estado para migrar al software libre, que es lo que hace falta para recuperar la soberanía informática del país. Una dependencia pública hace sistemas de información para los ciudadanos, si pierde el control de éstos, está incumplimiendo con su responsabilidad. Si tú pierdes el control de tus sistemas, es una lástima para tí y diría "lo siento", pero para una dependencia pública es mucho peor, porque afecta a sus ciudadanos. Pero sobre todo, la educación debe migrar al software libre, y el estado tiene la responsabilidad de asegurarlo.

Hablando sobre seguridad y protección de datos, al querer el Gobierno más control sobre los datos de la población con frecuencia hay más riesgo de que estos sean comprometidos, ¿qué opinas sobre esto?

Los datos que el Estado tiene, los tiene que proteger. Pero la protección más importante es que no recojan tantos datos personales. Si los datos se recogen se abusará de ellos. Así que el Estado debe de recoger menos datos.

¿Como podemos tener libertad en el contexto de la telefonía móvil?

El problema principal de los sistemas móviles es que nos vigilan, y no sé cómo evitarlo, porque el sistema de telefonía es capaz de ubicar el teléfono hasta sin su cooperación activa; comparando el tiempo de llegada de la señal a varias torres puede ubicar el teléfono aunque el teléfono no haga nada para ayudar. Esto me parece insoportable. Aunque había un teléfono móvil con GNU/Linux casi totalmente libre, cuando pensaba si tuviera uno cómo lo usaría para limitar la vigilancia a un nivel aceptable, no pude desarrollar un protocolo de uso. Tendría que preguntarme cuán frecuentemente y en qué tipo de lugar conectaría mi teléfono a la red de telefonía para aceptar llamadas. No pude llegar a una solución aceptable.

Otros comentarios

Uno de los alumnos le dijo a Stallman que el software privativo también da ventajas competitivas a sus usuarios, puesto que también son herramientas de trabajo y que incluso con frecuencia, son mucho más amigables, rápidas y fáciles de utilizar. Ante tal argumento Stallman esbozó una sonrisa diciendo “la libertad exige sacrificio”.

Al ser cuestionado sobre cómo lograr remuneración económica desarrollando software libre Stallman dijo que con donativos de usuarios satisfechos o donativos de personas que valoren el esfuerzo. Agregó: “Dar es la naturaleza humana, dar se siente bien, y yo me baso en la naturaleza humana”.

En fin, hablar de Stallman es hablar de un personaje controversial. Es una voz que puede llegar al extremismo, pero acarrea un mensaje que no podemos ignorar.

Marcando la Pauta para las Pruebas Ágiles

Autor:

Berenice Ruiz

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Principal

Dada su naturaleza, la industria del software enfrenta el enorme reto de mantenerse al ritmo de las cambiantes necesidades del mercado, la competencia y la globalización. Esto hace que la brecha entre la liberación de los productos de software y su comercialización se reduzca cada vez más, marcando con ello una dinámica de “puesta en producción” muy acelerada.

Las metodologías de desarrollo ágil han surgido intentando generar sistemas con alta calidad y a la vez, con una reducción y simplificación de tareas. Bajo este enfoque, las pruebas de software han tomado un papel crucial, dada la necesidad de realizar pequeñas liberaciones "funcionalmente" estables, surgiendo así el testing ágil. Para entender éste, es importante comprender cómo es que las pruebas encajan a lo largo de todo el ciclo de vida del desarrollo de software (SDLC). La figura 1 muestra un diagrama de Scott Ambler [1] que a grandes rasgos expone el ciclo de vida de desarrollo de software ágil.

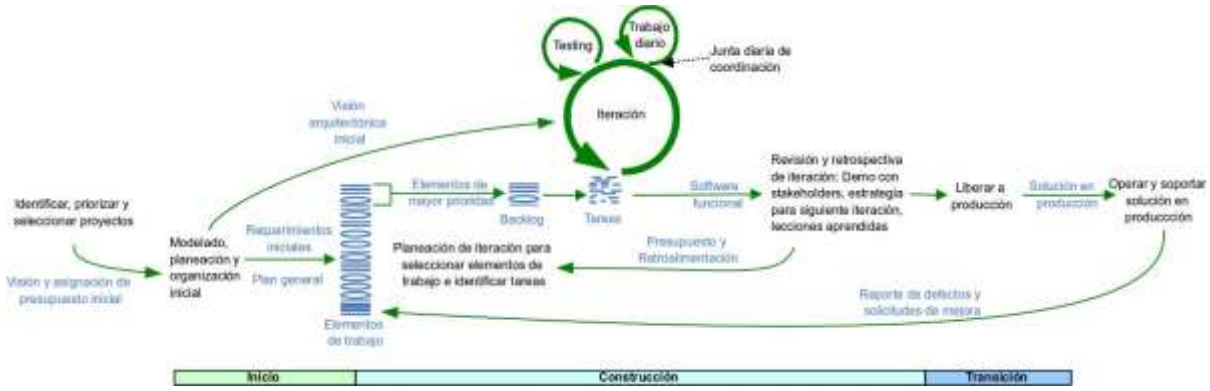


Figura 1. Ciclo de vida de desarrollo ágil

El testing ágil refiere un enfoque dinámico de las pruebas de software. Dicho enfoque supone que los requerimientos no son estables sino que se incrementan de forma continua o se encuentran en constante cambio, de acuerdo a las necesidades expresadas por el cliente. Asimismo, el cliente también juega un rol muy importante manteniendo una comunicación cercana y continua con el equipo de desarrollo; la obtención de sus necesidades (requerimientos) no se considera como una fase separada del desarrollo del software, sino una parte integral del mismo.

Mientras que en las metodologías tradicionales las actividades de prueba normalmente se pueden iniciar hasta que la especificación de requisitos se encuentre completa, en el caso del testing ágil dichas tareas quedan inmersas dentro de cada iteración.

Por citar otra de las diferencias importantes entre las metodologías tradicionales de pruebas y las ágiles, podemos destacar que mientras las primeras tienen como objetivo primordial la validación del producto desarrollado, en las ágiles tiene gran peso su utilización como medio para guiar el desarrollo del software, sustituyendo así la definición formal de requisitos.

A continuación se presentan algunas prácticas y actividades de prueba, tomando en cuenta el enfoque ágil.

Identificar aspectos a probar. Considerar las siguientes preguntas como guía para el desarrollo de los casos de prueba:

- ¿Qué necesidades del usuario debe resolver este producto?
- ¿Cuáles son las más críticas desde el punto de vista del usuario? (relación con pruebas de aceptación).
- ¿Cuál es el comportamiento esperado? ¿Cuál es la secuencia de acciones? (historias de usuario)
- ¿Hay alguna dependencia especial en el sistema?

- ¿Existen requerimientos no funcionales? ¿Cuáles?
- ¿Cuáles son las limitaciones del software/hardware respecto a características, funciones, datos, tiempo, etc.?
- ¿Las descripciones son lo suficientemente completas para decidir cómo diseñar, implementar y probar cada requisito y el sistema en sí como un todo?
- ¿Qué problemas y riesgos pudieran estar asociados con estos requisitos?

Definir el alcance de pruebas. Éste tiene que encontrarse perfectamente delimitado por cada uno de los entregables solicitados por el cliente en cada iteración. Puede identificarse respondiendo a las preguntas:

- ¿Qué necesidades del cliente van a ser incluídas en esta liberación?
- ¿Cuáles van a excluirse de las pruebas en este producto específico?
- ¿Qué es lo nuevo en esta liberación con respecto a otras?
- ¿Qué ha cambiado o se ha actualizado/corregido para este producto?

Identificar las métricas de pruebas. Elegir las métricas que se van a utilizar para dar seguimiento a los objetivos asociados a cada liberación.

Definir los niveles de pruebas que se aplicarán. Por ejemplo: unitarias, de integración, de aceptación.

Definir los tipos de prueba que se realizarán. Por ejemplo: funcionales, de carga, stress, seguridad, etcétera.

Definir estrategias de pruebas.

- Identificar las técnicas utilizar. Ejemplos: Pruebas exploratorias, Pruebas basadas en Riesgos, Pruebas automatizadas, etc.
- Identificar las herramientas de ejecución y de administración de pruebas usar, buscando principalmente aquellas cuyas plantillas de registro de defectos (por ejemplo), sean lo más simples y concisas posible, evitando trabajo redundante y exhaustivo; en el caso de herramientas de automatización, dependerán del lenguaje.
- Realizar la selección de los datos de pruebas.
- Definir cómo se va a preparar y configurar el ambiente de pruebas.

Definir el punto de terminación de las pruebas.

- ¿Cuándo continuar o detener las pruebas antes de entregar el sistema al cliente?
- ¿Qué criterios de evaluación deben cubrirse?
- ¿Qué criterios finales de aceptación deberán satisfacerse?

Pruebas de aceptación y ATDD

Las diversas metodologías ágiles (Scrum, eXtreme Programming, etcétera) de algún modo han promovido la aplicación de las pruebas, dado que éstas son indispensables para la liberación exitosa (con calidad) de cada entregable; con ello también se han diversificado las estrategias para abordar dichos proyectos. Es así que han ganado popularidad estrategias como el Desarrollo Dirigido por Pruebas (TDD -Test Driven Development) y el Desarrollo Dirigido por Pruebas de Aceptación (ATDD - Aceptance Test Driven Develepment).

Sobre este último enfoque, ATDD, algunos aspectos clave a mencionar son los siguientes:

- Toma como principal referencia al usuario final, al cliente.

- Utiliza historias de usuario como requisitos a los que se asocian pruebas de aceptación (escenarios).
- Las pruebas de aceptación dirigen el diseño/desarrollo del sistema, ver Figura 2.
- Requieren entrar tempranamente en detalles de implementación e instanciación de datos de pruebas

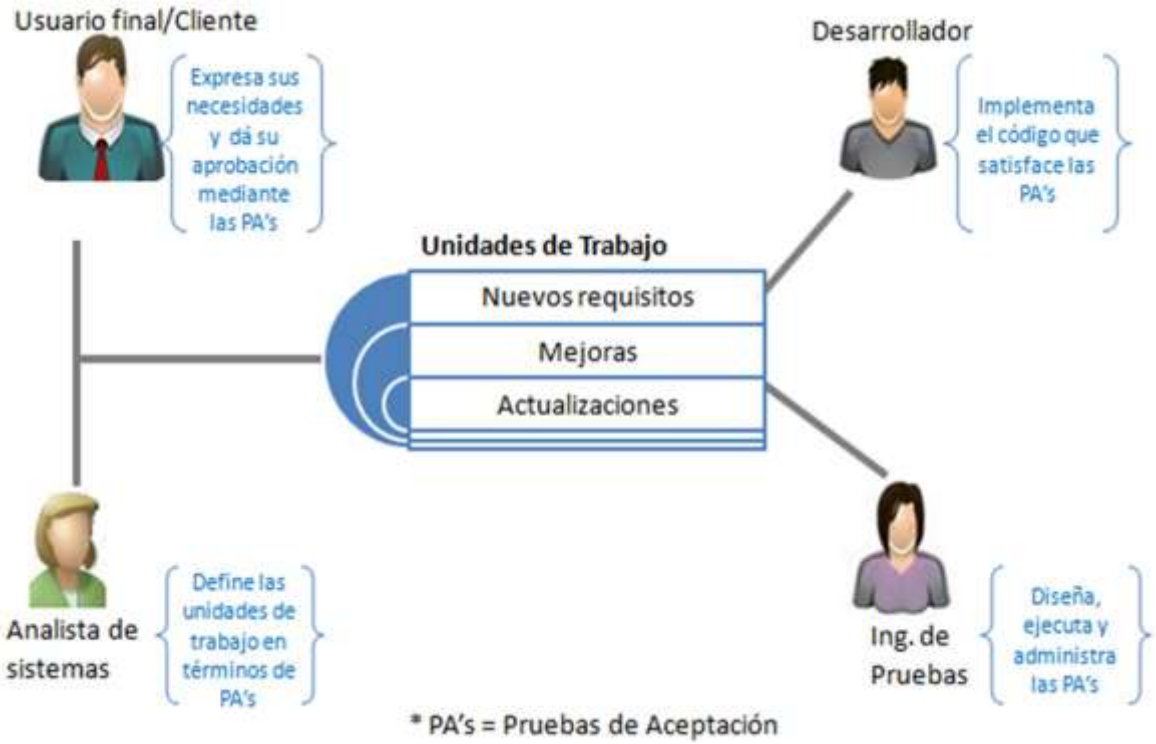


Figura 2. Proceso general de Desarrollo Dirigido por Pruebas de Aceptación.

De acuerdo al diccionario de la IEEE, cuando hablamos de pruebas de aceptación nos referimos a "aquellas pruebas formales con respecto a las necesidades, requerimientos y procesos de negocio del usuario, conducidas para determinar ya sea si un sistema satisface o no los criterios de aceptación y habilita al usuario, cliente o entidad autorizada para determinar si el sistema se acepta o no" [2].

Como ya decíamos, en ATDD cada necesidad del usuario final se representa a través de historias de usuario (al igual que en TDD, donde la base son las pruebas unitarias), y a su vez cada historia de usuario tiene asociados ciertos criterios de aceptación. Para cada criterio se deben diseñar los respectivos escenarios/pruebas de aceptación que deberán aprobarse; una vez que éstas pruebas han sido diseñadas, los desarrolladores comenzarán a programar el código fuente que permitirá satisfacerlas; por su lado los desarrolladores definirán las pruebas unitarias que evaluarán el cumplimiento de los requisitos (expresados en términos de criterios de aceptación).

Vale la pena reiterar que las Pruebas de Aceptación son sólo el medio para probar la funcionalidad del sistema, pero ATDD representa todo el proceso de desarrollo cuya finalidad es que el sistema se construya de

acuerdo a la funcionalidad expresada por el usuario. Por eso es que las pruebas de aceptación se definen en lenguaje natural, de modo que puedan ser comprendidas por los usuarios o analista de negocios del cliente.

Al igual que la definición de cualquier escenario de prueba, se plantea que contenga como mínimo: Título, pre-requisitos / precondiciones, descripción/pasos, criterios de aceptación/resultado esperado, tipo de usuario.

Automatización

Una vez que hemos diseñado todas las pruebas de aceptación que cubrirán todos los criterios que serán avalados por el cliente, podemos proceder a su automatización en alguna herramienta de pruebas para tal objetivo, teniendo posibilidad de elegir entre varias en el mercado, desde algunas gratuitas con sus respectivas limitaciones, hasta aquellas más completas y/o complejas que se ofrecen para compra o en la nube, soportando diversas tecnologías, protocolos, sistemas operativos: Selenium, soapUI, TOSCA testsuite, HP Quick Test Pro, SilkTest, IBM Rational Functional Tester, SOATest, TestPartner, Visual Studio Test Profesional, etc.

La automatización en sí ciertamente nos ayudará a agilizar la ejecución de dichas pruebas de aceptación, sin embargo es de considerar también el mantenimiento que dichos scripts de prueba requerirán, máxime cuando de antemano sabemos que los requerimientos cambiarán continuamente, de ahí que el tema de la automatización de pruebas juega un papel crucial en un marco de trabajo Ágil.

En mi particular punto de vista, se debería hacer una análisis de los costos asociados con el mantenimiento de código que implícitamente se tendrá con la automatización y evaluar a conciencia qué partes de la aplicación convendrá probar bajo esta técnica y cuáles no, pues quizás en otros casos pueda resultar más efectiva alguna otra técnica como las mismas pruebas exploratorias manuales, realizadas por supuesto por ingenieros de pruebas con la mayor pericia o experiencia posible.

Perfil del tester

El aspecto humano no debe dejar de analizarse. El personal de pruebas definitivamente debe contar con las siguientes características:

- Excelentes habilidades de comunicación (de vital importancia por toda la interacción que se da entre los miembros de los equipos).
- Pasión y entusiasmo por las pruebas.
- Muy fuertes habilidades de trabajo en equipo, no individualismo.
- Programación en lenguajes de scripting.
- Habilidad para trabajar bajo presión (constantes entregas rápidas).
- Liderazgo.

Si bien es cierto, las habilidades anteriormente señaladas, no sólo debieran estar presentes en testers de proyectos de pruebas ágiles, sino en cualquier ingeniero de pruebas. Sin embargo, para este tipo de proyectos podríamos casi decir que son “las imperdonables”, dada la dinámica con la que se suele trabajar.

Conclusión

En mi opinión, no buscaría decidir cuál metodología es mejor, sino más bien buscaría identificar claramente cuándo aplicar alguna y cuándo otra, dado que los proyectos son cambiantes, son diferentes en muchos aspectos, entre ellos en tamaño, en complejidad, etc. Toda esa serie de elementos nos permitirán definir si las metodologías ágiles son aplicables o no en cada caso. Creo también que puede incluso presentarse un híbrido de metodologías ágiles y convencionales, teniendo por supuesto muy claro cómo sacar mayor ventaja a los elementos que una y otra ofrecen.

Bibliografía:

[2] [IEEE 610] IEEE Std 610-1991, IEEE Computer Dictionary – Compilation of IEEE Standard Computer Glossaries, IEEE.

Bio:

Sandra Berenice Ruiz Eguino es Directora de Operaciones de e-Quallity, además ha participado como Consultora Senior en proyectos de mejora de organizaciones de Prueba de Software. Cuenta con certificación internacional en Pruebas por el ASTQB. A lo largo de su trayectoria profesional ha actuado también como Ingeniero de Pruebas Senior, Líder de Proyectos, Administradora de Proyectos nacionales e internacionales, analista y desarrolladora. Ha sido profesora de la Universidad Autónoma de Guadalajara (UAG), donde realizó sus estudios de Maestría en Ciencias Computacionales.

Pruebas y el Ciclo de Vida Ágil

Autor:

Germán Domínguez

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Principal

Aplicar los principios de los métodos ágiles al ciclo de vida de las pruebas permite mantener y mejorar su velocidad, la documentación, reutilización de casos de prueba, así como también la capacidad de repetir y automatizar las mismas.

Por otro lado, las mejores prácticas en el ciclo de vida del desarrollo de las aplicaciones indican que el ciclo de vida de las pruebas debe iniciar en el momento que los requerimientos han sido finalizados, esto permite que las pruebas sean basadas en las solicitudes iniciales del usuario/cliente y no ejecutadas sobre aquellas funcionalidades que el equipo de desarrollo generó, porque no necesariamente podría ser lo mismo.

Este inicio temprano del ciclo de vida de las pruebas sirve para aplicar uno de los principios de administración de la calidad: "empezar a probar temprano y probar más". En consecuencia, tener más resultados de pruebas permitirá maximizar la probabilidad de encontrar errores.

Las pruebas en el ciclo de vida ágil

Dado que las pruebas deben de formar parte integral del ciclo de vida del desarrollo de las aplicaciones, el flujo de proceso ideal debería ser el siguiente:

1. Requerimientos. Conforme los requerimientos, o solicitudes de cambio, van siendo definidos, revisados y autorizados, cada uno de ellos sirve como entrada para definir los casos de prueba con los que estos requerimientos serán cubiertos y probados.

2. Planeación y diseño. Con base en la planeación de las actividades de construcción y codificación de las aplicaciones, que a su vez ha tenido como entrada aquellos requerimientos adecuadamente priorizados y asignados a la iteración actual, es posible para el equipo de pruebas ir planeando e iniciar el proceso de aprovisionamiento del ambiente de pruebas. De igual manera, la planeación inicial de las pruebas, permitirá identificar aquellos casos, previamente creados y/o utilizados, que sean similares y que puedan ser reutilizados. Lo más importante, es que el equipo de pruebas no necesite "descubrir el agua tibia" con cada prueba que se requiera, y únicamente dedicar el tiempo a crear los pasos necesarios para validar las deltas que existen entre la aplicación anterior con la aplicación futura basada en las modificaciones que se estén creando en la iteración actual.

3. Ejecución. Conforme se implementan las piezas funcionales de código, los casos de prueba son ejecutados contra éstas, validando que el requerimiento original esté cubierto e implementado. A partir de los resultados de las pruebas se generan los protocolos pertinentes ya sea de aceptación de las piezas o la documentación de las solicitudes de mejora en forma de reportes de defectos o cambios.

4. Retroalimentación del proceso. Con el paso anterior, se generan nuevas entradas a la planeación de actividades de construcción y codificación, creando un macro-ciclo a través de las diferentes disciplinas del ciclo de vida del desarrollo de las aplicaciones.

Agilizando las pruebas

Un factor fundamental para mantener la agilidad de las diversas actividades de las pruebas, está basado en el ejercicio amplio y abierto de la colaboración entre los diversos equipos. La colaboración propuesta debe permitir el intercambio de la documentación intrínseca que se va generando durante el proceso de madurez de los artefactos e interacciones entre los diferentes actores del proceso de desarrollo de las aplicaciones. De esta manera, se satisface uno de los principios de las metodologías ágiles, "documentación mínima necesaria".

Para aceptar el "cambio como una constante", es necesario poder identificar y analizar fácilmente el impacto de los cambios a través de todo el ciclo de vida del desarrollo de las aplicaciones. Esto se logra conociendo: las actividades de construcción y codificación que están incluidas en una iteración, los casos de prueba que

validan cada uno de los requerimientos, y los resultados de las pruebas ya que afectan la planificación de las actividades en la iteración actual y futuras.

Para poder hacer más sencillos estos procesos complejos que involucran equipos, prácticas y disciplinas diversas, es necesario poseer herramientas y/o repositorios donde la información, documentación y resultados estén disponibles para todos los equipos.

Hablar de pruebas en el contexto de desarrollo ágil me recuerda la premisa que plantea Eric S. Raymond en su ensayo “The Cathedral and the Bazaar”:

Bio:

Germán Domínguez (germand@mx1.ibm.com) es Client Technical Professional para IBM Rational en México desde el 2008. Tiene 17 años de experiencia en desarrollo de aplicaciones de negocios, consultoría de procesos y lenguajes de programación. Actualmente apoya a diversos clientes con estrategias de modernización empresarial, application lifecycle management, arquitectura empresarial y herramientas de desarrollo.

Agilizando las Pruebas de Desempeño

Autor:

Federico Toledo

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Principal

Al realizar pruebas de desempeño (performance), debemos entregar resultados lo antes posible, de manera que minimicemos el costo de estas y permitir que se implementen las mejoras lo antes posible. En este artículo se presenta una estrategia para realizar las pruebas de performance de una manera ágil, a través de llevar a cabo ciertas acciones de manera temprana dentro del ciclo de desarrollo de un aplicación dada y con la colaboración del equipo de desarrollo.

El enfoque tradicional

Existen distintos tipos de pruebas de desempeño: load test (test de carga), donde la idea es simular la realidad a la que estará expuesto el sistema cuando esté en producción; stress test, donde se va más allá de la carga esperada para ver dónde se “truena” el sistema; endurance (resistencia), para ver cómo se desempeña el sistema luego de una carga duradera por un período largo de tiempo, etcétera.

Cualquiera que sea el tipo de prueba de rendimiento a realizar, requiera una preparación que lleva un esfuerzo no despreciable.

1. Es necesario hacer un análisis y diseño de la carga que se quiere simular. Esto incluye: duración de la simulación, casos de prueba, cantidades de usuarios que los ejecutarán, cuántas veces por hora, ramp-up (velocidad y cadencia con la que ingresan usuarios virtuales al inicio de una prueba), etcétera.
2. Luego de diseñado el "escenario de carga" pasamos a preparar la simulación con la(s) herramienta(s) de simulación de carga. A esta tarea se le denomina automatización o robotización y es en cierta forma una tarea de programación. Por más que estemos utilizando herramientas de última generación, esto involucra tiempo que separa más el día de inicio del proyecto del día en que se comienzan a reportar datos sobre el desempeño del sistema a nuestro cliente.
3. Por último, comenzamos con la ejecución de las pruebas. Esto lo hacemos con toda la simulación preparada y las herramientas de monitoreo configuradas para obtener los datos de desempeño de los distintos elementos que conforman la infraestructura del sistema bajo pruebas. Esta etapa es un ciclo de ejecución, análisis y ajuste, que tiene como finalidad lograr mejoras en el sistema con los datos que se obtienen luego de cada ejecución.

A medida que se adquiere experiencia con las herramientas, los tiempos de automatización van mejorando. Pero aun así, ésta sigue siendo una de las partes más pesadas de cada proyecto. Generalmente se intenta acotar la cantidad de casos de prueba que se incluyen en un escenario de carga para mantenerse en un margen aceptable, intentando no incluir más de 15 casos de prueba (varía dependiendo del proyecto, la complejidad del sistema y los recursos y tiempo disponibles).

El enfoque tradicional de pruebas de desempeño plantea un proceso en cascada, en donde no se comienza a ejecutar pruebas hasta que se hayan superado las etapas previas de diseño e implementación de las pruebas automatizadas. Una vez que esto está listo el proceso de ejecución inicia, y a partir de ese momento el cliente comienza a recibir resultados (ver figura 1). Es decir que desde que el cliente firmó el contrato hasta que recibe un primer resultado pueden pasar un par de semanas. Aunque parezca poco, esto es demasiado para las dimensiones de un proyecto de performance, que generalmente no dura más de uno o dos meses.

Figura 1. El enfoque tradicional de pruebas de desempeño

Es necesario agilizar este proceso tradicional para brindar resultados lo antes posible. Así el equipo de desarrollo podrá realizar mejoras cuanto antes, y estaremos verificando prontamente la validez de nuestras pruebas. Si tenemos algún error en el análisis o diseño de las pruebas y nos enteramos de esto hasta el final, eso será muy costoso de corregir; de la misma forma que un bug es más costoso para un equipo de desarrollo cuando se detecta en producción.

Agilizando el proceso

Veamos entonces algunas ideas de cómo agilizar este proceso.

Entregar resultados desde que estamos automatizando.
Aunque el foco de la automatización no es encontrar errores o mejoras al sistema, si estamos atentos podemos detectar posibilidades de mejora. Esto es posible dado que la automatización implica trabajar a nivel de protocolo, y se termina conociendo y analizando bastante el tráfico de comunicación entre el cliente y el servidor.

Por ejemplo, en el caso de un sistema Web, es posible encontrar posibilidades de mejora tales como:

- Manejo de cookies.
- Problemas con variables enviadas en parámetros.
- Problemas de seguridad.
- HTML innecesario.
- Recursos inexistentes (imágenes, css, js, etc.).
- Redirecciones innecesarias.

Es cierto que el equipo de desarrollo debería detectar estos errores previamente; sin embargo la realidad muestra que en la mayoría de los proyectos se encuentran muchos de estos incidentes. Por eso, al realizar estas pruebas estamos entregando resultados de valor que mejorarán el rendimiento del sistema.

Pipeline entre automatización y ejecución.
Para poder comenzar a automatizar pruebas para usarlas en pruebas de desempeño no es necesario esperar a contar con todo el conjunto de pruebas armado. En cuanto tengamos los primeros casos de prueba podemos comenzar a automatizarlos y aplicarlos (ver figura 2).

Figura 2. Flujo ágil de pruebas de desempeño

Este pipeline nos da mayor agilidad para obtener resultados lo antes posible en la automatización de pruebas, aprovechando ejecutar pruebas iniciales con cada nuevo caso de prueba que se automatice. También nos permite analizar cómo se comporta cada caso de prueba en específico. Estas pruebas son de gran valor porque permiten encontrar problemas de concurrencia, bloqueos de acceso a tablas, falta de índices, utilización de caché, tráfico, etcétera.

Para cuando completemos la automatización de todos los casos de prueba, el equipo de desarrollo ya estará trabajado en las primeras mejoras que les fuimos reportando, habiendo así solucionado los primeros problemas que hubiesen aparecido al ejecutar pruebas. Otra ventaja es que las pruebas con un solo caso de prueba son más fáciles de ejecutar que las pruebas con 15 casos de prueba, pues es más fácil preparar los datos, analizar los resultados, e incluso analizar cómo mejorar el sistema.

Otra técnica que se utiliza es ejecutar una prueba en la que haya sólo un usuario ejecutando un caso de prueba, pues de esa forma se obtendría el mejor tiempo de respuesta que se puede llegar a tener para ese caso de prueba, pues toda la infraestructura está dedicada en exclusiva para ese usuario. Esta prueba debería

ejecutarse al menos con 15 datos distintos a modo de tener datos con cierta validez estadística, y no tener tiempos afectados por la inicialización de estructuras de datos, de cachés, compilación de servlets, etcétera, que nos hagan llegar a conclusiones equivocadas. Con esto podemos ver cuál es el tiempo de respuesta que tiene el caso de prueba funcionando "solo", para luego comparar estos tiempos con los de la prueba en concurrencia con otros casos y ver cómo impactan unos sobre otros. Lo que suele pasar es que algunos casos de prueba bloqueen tablas que otros utilizan y de esa forma repercutan en los tiempos de respuesta al estar en concurrencia. Esta comparación no la podríamos hacer tan fácil si ejecutamos directamente la carga completa.

Considerar el rendimiento desde el desarrollo.
Generalmente los desarrolladores no consideran el rendimiento desde el comienzo de un proyecto. Con un poco de esfuerzo extra se podrían solucionar problemas en forma temprana, y hablamos de problemas graves: arquitectura, mecanismos de conexión, estructuras de datos, algoritmos. Como equipo de testing, siempre felicitamos a los desarrolladores que hacen una prueba en la cual simplemente instancian varios procesos (digamos unos 30, por poner un número, pero dependerá de cada sistema y de la infraestructura disponible para ejecutar la prueba), en el cual se ejecuta la funcionalidad que nos interesa con cierta variedad de datos. Esta práctica nos puede anticipar una enorme cantidad de problemas, que serán mucho más difíciles de resolver si los dejamos para una última instancia. Si sabemos que vamos a ejecutar pruebas de performance al final del desarrollo, tenemos una tranquilidad muy grande. Pero tampoco es bueno confiarse en eso y esperar hasta ese momento para trabajar en el rendimiento del sistema. Podremos mejorar el desempeño de todo el equipo y ahorrar costos del proyecto si tenemos en cuenta estos aspectos en forma temprana, desde el desarrollo.

Conclusión

Sin duda las pruebas de performance son muy importantes y nos dan una tranquilidad muy grande a la hora de poner un sistema en producción. El problema es seguir un proceso en cascada para ponerlo en práctica: es necesario agilizar las pruebas de performance. Mientras más temprano obtengamos resultados, más eficientes seremos en nuestro objetivo de garantizar el rendimiento (performance) de un sistema.

Bio:

Federico Toledo (@fltoledo) es ingeniero en Computación por la Universidad de la República de Uruguay y sustenta la maestría de Informática por la universidad de Castilla-La Mancha en España. Actualmente se encuentra realizando sus estudios de doctorado enfocados a la especialización en el diseño de pruebas automáticas para sistemas de información y es socio co-fundador de Abstracta (www.abstractaconsulting.com), empresa con presencia en México y Uruguay, que se dedica a servicios y desarrollo de productos para testing.

Test Driven Development

Autor:

Alfredo Chavez

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Principal

Durante los últimos ocho años he estado practicando esporádicamente la disciplina del Desarrollo Dirigido por Pruebas (TDD), lo cual no ha sido un proceso fácil ni rápido. El propósito de este artículo es allanar un poco el camino para aquellos que estén considerando aprender TDD y posiblemente utilizarlo en su trabajo o en proyectos personales.

TDD es un tema bastante amplio, por lo que en este artículo me concentraré exclusivamente en el tema de las pruebas unitarias.

¿Qué es TDD?

TDD es una criatura extraña; es simple de definir, pero su definición parece ir en contra del sentido común; es sencilla de explicar, pero difícil de llevar a cabo.

TDD es una disciplina que promueve el desarrollo de software con altos niveles de calidad, simplicidad de diseño y productividad del programador, mediante la utilización de una amplia gama de tipos de pruebas automáticas a lo largo de todo el ciclo de vida del software. El principio fundamental es que las pruebas se escriben antes que el software de producción y estas constituyen la especificación objetiva del mismo.

La primera parte de la definición suena todo miel sobre hojuelas. ¿Quién no quiere software confiable, bien diseñado y producido rápidamente?. Sin embargo, todo esto no viene gratuitamente; la palabra clave aquí es disciplina.

D*isciplina: Doctrina, instrucción de una persona, especialmente en lo moral. Observancia de las leyes y ordenamientos de la profesión o instituto.*

Esto nos lleva a la conclusión de que si TDD es en efecto una disciplina, entonces no es algo que aplicamos "según nos vayamos sintiendo", sino que es algo que debe formar parte integral de nuestra profesión o arte.

Las pruebas primero

La segunda parte de la definición de TDD viene con el primer "¡¿qué demonios?!" para muchos: Las pruebas se deben escribir antes que el software mismo.

Para entender que esto no es algo del otro mundo, recordemos cuando aprendimos a programar. Probablemente aprendimos con un lenguaje interpretado y normalmente comenzamos con cosas simples como por ejemplo, sumar 2 y 3:

```
>>> 2 + 3
5
```

Intuitivamente pensamos "debe dar cinco", incluso antes de oprimir la tecla Enter; y normalmente funciona; Si no, entonces hay algo definitivamente mal con el lenguaje o nuestro entendimiento del mismo. Posteriormente pasamos a cosas más complejas y/o sofisticadas, como por ejemplo:

```
>>> def sum(a, b):
...   return a + b
...
>>> sum(2, 3)
5
```

Cada que vemos aparecer en la pantalla el resultado que esperamos, aumenta nuestra autoconfianza. Esto nos motiva a seguir aprendiendo, a seguir programando. Este podría tal vez ser el ejemplo más básico de TDD.

Una vez que tomamos mayor confianza, comenzamos a escribir cantidades cada vez mayores de código entre una comprobación y la siguiente del resultado. Como creemos que nuestro código "está bien", comenzamos a "optimizar el tiempo" escribiendo más y más código de un jalón. Al poco tiempo, nos olvidamos de estas primeras experiencias, incluso tachándolas como "cosas de novatos".

Llegamos al presente y nos encontramos a nosotros mismos tratando de aprender TDD. Nos conseguimos una copia de JUnit, NUnit, o el framework de moda para nuestro lenguaje de elección y comenzamos a seguir un tutorial que encontramos por ahí. A partir de aquí, estamos en la parte sencilla de nuestra curva de aprendizaje. Comenzamos a producir grandes cantidades de pruebas y no tardamos en sentirnos cómodos con el proceso.

Conforme comenzamos a escribir pruebas para proyectos más complejos nos topamos con varios obstáculos en el camino:

- Las pruebas se tornan difíciles de escribir, por lo que sentimos una desaceleración importante.
- Las pruebas tardan en ejecutarse, por lo que nos volvemos renuentes a ejecutarlas frecuentemente.
- Los cambios aparentemente sin importancia en el código provocan que un montón de pruebas fallen. Mantenerlas en forma y funcionando se vuelve complejo y consume tiempo.

Es en este punto donde la mayoría de las personas y equipos se da por vencido con TDD. Estamos en la parte más pronunciada de nuestra curva de aprendizaje. Estamos produciendo pruebas y obteniendo valor de las mismas, pero el esfuerzo para escribir/mantener estas mismas parece desproporcionado.

Al igual que con cualquier otra habilidad que vale la pena adquirir, si en lugar de rendirnos seguimos adelante, eventualmente aprenderemos a cruzar a la parte de nuestra gráfica donde la pendiente de la curva se invierte y comenzamos a escribir pruebas más efectivas con un menor esfuerzo y a cosechar los beneficios de nuestra perseverancia.

Aprender a escribir bien y de mantener las pruebas toma tiempo y práctica. En el resto de este artículo comparto algunos tips para ayudar a acelerar un poco este proceso.

Las reglas de TDD

Robert C. Martin, una de las máximas autoridades en TDD, describe el proceso en base a tres simples reglas:

1. No está permitido escribir ningún código de producción sin tener una prueba que falle.
2. No está permitido escribir más código de prueba que el necesario para fallar (y no compilar es fallar).
3. No está permitido escribir más código de producción que el necesario para pasar su prueba unitaria.

Esto significa que antes de poder escribir cualquier código, debemos pensar en una prueba apropiada para él. Pero por la regla número dos, ¡tampoco podemos escribir mucho de dicha prueba! En realidad, debemos detenernos en el momento en que la prueba falla al compilar o falla un assert y comenzar a escribir código de producción. Pero por la regla número tres, tan pronto como la prueba pasa (o compila, según el caso), debemos dejar de escribir código y continuar escribiendo la prueba unitaria o pasar a la siguiente prueba.

Esto se entenderá mejor con un pequeño ejemplo. Vamos a crear un código en Python que cuando le indiquemos un número entero, nos regrese un arreglo con sus factores primos.

Escribimos suficiente de nuestra primera prueba para que falle

```
from unittest import main, TestCase


class TestPrimeFactors(TestCase):

    def testPrimesOf0(self):

        self.assertEquals([], factorsOf(0))


if __name__ == '__main__':

    main()
```

Al correr este código se ejecuta la prueba “testPrimesOf0” y obtenemos un error "NameError: global name 'factorsOf' is not defined". Esta es nuestra señal para detenernos y agregar a nuestro código la definición de factorsOf:

```
def factorsOf(n):
    return []
```

Ahora ya pasa nuestra prueba. Así que podemos continuar escribiendo código de prueba. Reemplazamos nuestro caso de prueba para incluir el caso del 1.

```
def testPrimesOf0to1(self):
    self.assertEquals([], factorsOf(0))
    self.assertEquals([], factorsOf(1))
```

Funciona bien. Ahora agreguemos una prueba para el 2, el cual es un número primo y por lo tanto debe regresar un arreglo con su mismo valor.

```
def testPrimesOf2(self):
   self.assertEquals([2], factorsOf(2))
```

Como era de esperarse, la prueba falla, así que es hora de escribir código. Cambiamos la implementación de nuestra función factorsOf por:

```
def factorsOf(n):
   if n > 1:
      return [n]
   return []
```

Con esto, la prueba es válida. Intentemos con otro número primo, el 3.

```
def testPrimesOf2to3(self):
   self.assertEquals([2], factorsOf(2))
   self.assertEquals([3], factorsOf(3))
```

También es válida. Así que agregamos una nueva prueba, donde alimentamos un número con factores.

```
def testPrimesOf2to4(self):
      self.assertEquals([2], factorsOf(2))
      self.assertEquals([3], factorsOf(3))
      self.assertEquals([2,2], factorsOf(4))
```

Obtenemos un error con el mensaje: “AssertionError: Lists differ: [2, 2] != [4]”, ya que nuestra función factorsOf no está lista para arrojar el resultado esperado. Hora de modificar el código:

```
def factorsOf(n):

   result, factor = [], 2

   while n > 1:

      while n % factor == 0:

         result.append(factor)

         n /= factor

      factor += 1

   return result
```

Con esto ya pasamos todas las pruebas, así que ya tenemos listo nuestro código.

[Nota del editor: El código de este ejemplo está disponible en github en http://swgu.ru/38r3 y si consultas su historial de revisiones verás los mismos pasos que seguimos aquí.]

Obviamente he resumido el proceso un poco debido a limitaciones de espacio, pero creo que el proceso es claro. Como habrán visto, en ningún momento escribimos mucho código de una sola vez. ¡Y de eso se trata precisamente! Es mucho muy similar al proceso de aprendizaje previamente descrito, cuando probábamos nuestro código interactivamente en el intérprete. La retroalimentación constante nos motiva a seguir adelante

con confianza y determinación, ya que sabemos que en todo momento nuestro sistema está funcionando. Incluso si introducimos un bug por error, podemos resolverlo con unos cuantos Ctrl-Z. Y eso es algo valioso.

Es común que para resolver un problema de programación nos basemos en código extraído de ejemplos u otros sistemas. Un ajuste aquí, otro allá hasta que aparentemente funciona. El problema es que no estamos seguros de lo que hicimos. Si el código falla después, no tenemos mucha idea de por qué. Al seguir de forma disciplinada TDD, entendemos si lo que estamos haciendo funciona o no, y por qué.

Escribiendo pruebas unitarias efectivas

En su libro The Art of Unit Testing [1], Roy Osherove dice que las buenas pruebas tienen tres propiedades comunes: son legibles, confiables y fáciles de mantener. Una cuarta propiedad que yo agregaría es "rapidez". A continuación explico estas propiedades.

Legibilidad
Una prueba legible es aquella que revela su propósito o razón de ser de forma clara; básicamente, qué es lo que la prueba ha de demostrar. Una parte importante de ésto consiste simplemente en darle un nombre apropiado. Si está probando una pila, por ejemplo, entonces no llamemos nuestras pruebas testStack_01, testStack_02, etcétera; este tipo de nombre son inútiles. Debemos elegir nombres que reflejen el comportamiento útil observable que el código debiera exhibir. Por ejemplo, testElementosGuardadosSonRegresadosEnOrdenInverso es un nombre que describe un comportamiento observable de las pilas: los elementos colocados al principio son los últimos en ser devueltos.

Es conveniente considerar que los nombres de las pruebas forman parte de la documentación del comportamiento de la Unidad de Código Bajo Prueba. Cuando llega el momento de implementar una nueva clase, a menudo encuentro útil comenzar con una lista inicial de las pruebas que quiero escribir (no siempre lo hago, pero a veces resulta indispensable).

Cuando una prueba lleva el nombre de una conducta observable, esta debe reflejar únicamente dicho aspecto del código. Es aceptable tener más de un assert dentro de una prueba, siempre que estos se refieran a una sola cosa, generalmente a un solo objeto.

Encontrar el justo equilibrio entre tener el código de inicialización dentro de las pruebas, en una fábrica o en un método setup dedicado, es también un elemento importante de la legibilidad. Es importante reducir el volumen del código en las pruebas, pero también queremos que sea evidente lo que la prueba está haciendo. Es fácil caer en la trampa de ocultar muchos detalles en los métodos de inicialización o de fábrica, por lo que un lector tiene que buscar estos métodos para poder entender la prueba. El principio DRY, a veces se encuentra firmemente grabado en la consciencia de los buenos programadores. Sin embargo, es perfectamente aceptable tener un poco más de redundancia, mientras que el propósito se mantenga claro.

Esto último no quiere decir que podemos ignorar las reglas y escribir nuestras pruebas de forma descuidada. Nuestras pruebas son parte esencial de nuestro código. Son tan importantes como el código de producción (o de acuerdo con Robert C. Martin, son aún más importantes). Por lo tanto es necesario poner tanto esmero en su manufactura como el que pondríamos en la demo que haremos la próxima semana frente al cliente.

Confiabilidad
Una prueba confiable es la que falla o pasa de forma determinista. Las pruebas que dependen si la computadora está configurada correctamente, o cualquier otro tipo de variables externas, no son confiables,

porque no es posible saber si una falla significa que el equipo no está configurado correctamente, o si el código contiene errores.

Estas pruebas que dependen de variables externas son en realidad pruebas de integración, y se deben poner en un proyecto por separado, junto con alguna documentación sobre la forma de ponerse en marcha. Esto es deseable, ya que este tipo de pruebas normalmente se ejecutan mucho más lentamente que las pruebas unitarias típicas, por lo que al estar separadas, no impedirán que ejecutemos nuestras pruebas unitarias tan frecuentemente como deseemos/necesitemos. Una variable externa es cualquier cosa sobre la que no tenemos control directo: el sistema de archivos, bases de datos, el tiempo, el código de terceros, etc.

En algunas ocasiones especiales, es imposible evitar el tener una prueba indeterminable sin importar cuanto nos esforcemos. Martin Fowler y otros recomiendan en primer lugar, aislar estas pruebas. Lo último que queremos es acostumbrarnos a ver fallar pruebas en nuestra suite. Una barra roja para nosotros siempre debe ser una señal de alarma. No importa que podamos reconocer la prueba por su nombre. El punto de usar pruebas automáticas es precisamente no tener que inspeccionar visualmente los resultados para darlos o no por buenos. Si esto sucede, ¡podemos pasar por alto un fallo real sin notarlo! Otro punto es el analizar si una aproximación probabilística es útil en estos casos. Si los resultados de la prueba se encuentran acotados dentro de un margen de tolerancia, es posible eliminar la incertidumbre hasta un grado aceptable para nuestros propósitos.

Mantenibilidad
Una prueba fácil de mantener es aquella que no "se rompe" fácilmente cuando se le da mantenimiento. Un bajo acoplamiento es probablemente el factor más importante para la facilidad de mantenimiento. El uso de métodos de fábrica (Factory) nos permite desacoplar nuestras pruebas de los constructores de clase, que tienden sufrir cambios en sus listas de parámetros más a menudo que otros métodos.

Que las pruebas tengan buena legibilidad también ayuda al mantenimiento. Cuando se puede deducir a partir del nombre lo que la prueba está tratando de comprobar, se puede ver si en realidad el código hace lo que se pretende.

Rapidez
Una prueba unitaria efectiva debería ejecutarse en milisegundos. Una suite de pruebas puede llegar a contener cientos de pruebas, cada una enfocándose a un aspecto particular del código. Para minimizar la disrupción a nuestro flujo de trabajo, necesitamos que el conjunto de pruebas se ejecute en unos cuantos segundos. De no ser así, vamos a evitar ejecutarlas con la frecuencia necesaria, es decir, cada que hacemos un cambio; y entonces perdemos la confianza en nuestros cambios y en nosotros mismos. Regresamos al ritmo "tradicional" y finalmente puede llegar a parecernos "más fácil" abrir el depurador de nuestro IDE que dar un par de pasos hacia atrás hasta el punto en que todo aun funcionaba bien. Volver al ciclo tradicional de modificar/compilar/debuguear destruye la motivación que habíamos construido. Si probar un cambio de una sola linea nos lleva 5 minutos de revisión en el depurador, nuestra motivación cae por los suelos y se convierte en una excusa para alargar los tiempos de desarrollo casi infinitamente.

Un componente fundamental en la construcción de una suite de pruebas es la habilidad de construirla a partir de subconjuntos más pequeños y enfocados. Esto permite ejecutar únicamente las pruebas para la clase o el sub-sistema que estamos probando en este momento, lo cual disminuye el tiempo requerido para ejecutar las pruebas.

Tips para el aprendizaje

A continuación comparto algunos tips que te ayudarán para el aprendizaje de TDD.

- Escribe muchas pruebas, tantas como puedas. Familiarizate con el ritmo y las reglas de TDD.
- Comienza con algo sencillo (¡pero no te detengas ahí!).
- Cuando encuentres algo que no sabes como probar, apóyate en un compañero. Si no programas en parejas, consulta con un colega. Recolecta ideas de diversas fuentes.
- Sé persistente y no te rindas. Si quieres obtener los frutos, debes primero poner el trabajo duro.
- Conforme escribas más y más pruebas, comienza a organizarlas en suites y asegúrate que estas puedan ejecutarse de forma individual o colectiva, según sea necesario. ¡La organización también es una habilidad que hay que aprender!

Prácticas para el día a día

Es recomendable probar una unidad de código solo a través de su API pública (y en términos prácticos, "protegido" es efectivamente público). Al hacer esto, obtenemos un mejor aislamiento de los detalles específicos de la implementación.

Evita colocar lógica en el código de prueba (if-then, switch/case, etc). Donde hay lógica, hay la probabilidad de introducir bugs, ¡y definitivamente no queremos bugs en nuestras pruebas!

Evita calcular el valor esperado, ya que podríamos terminar duplicando el código de producción, incluyendo cualquier error que este pudiera tener. Preferiblemente, calcula el resultado esperado manualmente (y revisalo por lo menos un par de veces) y colócalo como una constante.

Evita compartir estado entre pruebas. Debe ser posible ejecutar las pruebas en cualquier orden o incluso, ejecutar una prueba dentro de otra prueba. Mantener las pruebas aisladas de las demás también es un factor indispensable para la confiabilidad y mantenibilidad de las mismas.

Ninguna cantidad de comentarios puede sustituir un código claro. Si una prueba se convierte en un desastre, reescríbela.

Si no es posible determinar lo que una prueba está haciendo, es probable que en realidad esté verificando múltiples cosas: hazla pedazos y convierte cada uno en su propia prueba individual.

Frecuentemente los errores en el código de pruebas se esconden en los métodos de inicialización. Mantener este código simple y compacto puede ser un gran paso para la mantenibilidad del código.

Una unidad de código puede necesitar operar en circunstancias de escenarios variables. Esto puede llevar a que el código de inicialización se convierta rápidamente en un desastre. Crea fixtures o incluso casos de prueba especializados para cada escenario.

Nunca escatimes en claridad. Si es necesario, convierte cada escenario en una clase de prueba individual.

Si al probar una parte de tu código parece que requieres tener la mitad o más del sistema presente, verifica el alcance de la misma. ¿Estás probando una sola cosa?

Si una parte del código es particularmente resistente a tus esfuerzos de probarla, voltea al código en busca de problemas en el diseño del mismo. Un código fácil de probar frecuentemente está débilmente acoplado con el resto del sistema, es altamente cohesivo y sigue los principios fundamentales del diseño de software.

Conclusión

A lo largo de este artículo he compartido algunos conceptos clave de TDD así como tips para llevarlo a cabo exitosamente. Mi último tip sería que no dejes de aprender: lee libros, revistas, blogs, etcétera. Los proyectos de código abierto también son una excelente fuente de aprendizaje.

Referencias

[1] R. Osherove. The Art of Unit Testing. Manning Publications, 2009.

Bio:

Alfredo Chavez (@alfredochv) comenzó a programar como hobby en 1989 con Basic y Pascal y profesionalmente desde 1993 con los lenguajes C y xBase. A principios de la década de 2000 descubre los métodos ágiles de desarrollo y desde entonces busca nuevas formas de aplicar sus prácticas en el trabajo del día a día. Se considera un autodidacta y un apasionado de la Programación Orientada a Objetos. Actualmente se interesa por enfoques que integren los principios y técnicas de la Programación Funcional y OO—algo así como una "Teoría de Campo Unificado de la Programación"

Pruebas Continuas

Autor:

Christian Ramírez

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Principal

Uno de los principales elementos del desarrollo ágil es la entrega continua y regular de valor. Por ello han surgido técnicas como la integración continua, la cual nos ayuda a tener un software compilable en todo momento. Pero no basta con que el software únicamente se pueda compilar, de hecho es el primer paso de toda una serie de elementos y características que debe cumplir el software.

Una práctica que ayuda a mitigar el nivel de riesgo introducido por los cambios creados con las modificaciones realizadas durante un sprint de desarrollo, es el ejecutar de manera constante, selectiva o completa las pruebas de software automatizadas que se tengan para las diferentes fases del ciclo de vida del software. Así pues, las pruebas unitarias serán las que más frecuentemente y de manera completa se ejecuten

y por otro lado, las pruebas encargadas de ejercitar el software para medir su desempeño serán las que menos se ejecuten.

Adaptándonos

Lo mencionado anteriormente conlleva un esfuerzo extra, ya que adicional a las labores de diseño y codificación, el equipo debe encontrar tiempo para ejecutar estas pruebas. Por ello la solución lógica es emplear el mismo modelo de continuidad empleado en la construcción del software, pero aplicado a las pruebas. Esta técnica se llama Pruebas Continuas (continuous testing) y consiste en adaptar a nuestro ciclo de vida fases donde se ejecuten las pruebas de software de los diferentes niveles de manera regular, con el fin de tener una retroalimentación acelerada para con nuestro equipo de desarrollo. De esta manera podemos ser más preventivos que reactivos.

Esto no es una labor sencilla, ya que involucra un cambio en el proceso. Primeramente es necesario tener una responsabilidad hacia todos los miembros del equipo sobre codificar y mantener las pruebas de software, También hay que seleccionar o construir las herramientas necesarias y adoptarlas. Finalmente hay que construir los mecanismos de visibilidad, es decir, la forma en la cual todos los miembros del equipo conocen y emplean la información generada de manera oportuna.

¿Por dónde iniciamos?

La mayoría de los desarrolladores y testers en nuestra región no están familiarizados con técnicas formales de diseño de pruebas y mucho menos con patrones de diseño de pruebas, por lo que es indispensable crear conciencia en el equipo de esta área de oportunidad para poder aprovecharla.

Para comenzar, lo primero que debe realizar el equipo es construir una prueba por cada pieza de software que se implemente. La prueba debe verificar que el software "hace lo que tiene que hacer", es decir, deberá revisar que en los casos "normales" del flujo de operación la pieza de software devuelve un resultado acorde al diseño, esto es lo que conocemos comúnmente como una prueba unitaria automatizada.

Principios y lineamientos

Estos son algunos principios que recomiendo tener en cuenta para el diseño de pruebas:

- Las pruebas de software deben reducir riesgo, no introducirlo.
- La ejecución de las pruebas de software debe ser sencilla.
- Las pruebas de software deben ser fáciles de mantener.

Adicional podemos ejecutar prácticas comunes del buen diseño de pruebas como por ejemplo:

- Cada prueba unitaria debe fallar solo por una razón. Es decir, cada prueba solo revisa una cosa a la vez. Es una muy mala práctica tener más de una verificación en una sola prueba.
- Sólo debe existir una prueba para una razón de fallo. Es así que no debe haber dos o más pruebas que fallen por el mismo motivo. Lo anterior ayuda a mantener una línea base de pruebas claramente definidas.
- Las pruebas deben ser completamente independientes de base de datos, de otros tests, de archivos, ui, etc.

Bajo estas premisas es muy sencillo realizar un proyecto de pruebas de software y que el mismo sea llevado a buen fin.

Organización

Una vez construidas nuestras pruebas de software es necesario agruparlas, con el objetivo de tener una clara separación de qué partes son las que afectamos. Por ejemplo, podríamos separarlas por funcionalidad del producto, por capas de la arquitectura o por cualquier otro elemento. El único requisito es que sea fácilmente trazable cualquier prueba, es decir que sepamos donde está el área de influencia de la misma. Con las pruebas ya separadas formamos "suites", cuyo objetivo es que puedan ser otro elemento de organización del proyecto, así al tener las pruebas segmentadas podemos ejecutarlas de manera selectiva.

Lo siguiente a contemplar es la selección y uso de un TestRunner, una herramienta que se encargará de ejecutar los tests y presentar los resultados. Esta herramienta deberá tener una interfaz por línea de comando debido a que será la pieza que servirá para comunicarse con el servidor de integración continua.

Ya con todas las herramientas necesarias, es preciso integrarlas de modo que no afecten nuestro ciclo de desarrollo. Básicamente se trata de introducir una serie de pasos más a nuestro actual flujo de trabajo, siempre de manera ordenada y coherente.

¿Gasto o inversión?

Es bien sabido que este tipo de pruebas requieren un esfuerzo adicional y requieren un alto grado de conocimiento de la herramienta que se empleará, pero es una inversión que muestra su valía una vez que los proyectos van creciendo y se vuelven más complejos. El papel de estas pruebas consiste en ejercitar todas las capas de la aplicación así como verificar que las piezas de software se comunican de manera correcta.

Uno de los elementos clave de la práctica es el definir el momento y la frecuencia con la cual ejecutaremos los distintos tipos de prueba. Para ello es de suma importancia considerar cómo afectaremos nuestro proceso de entrega. Primeramente todas las pruebas unitarias deben agregarse al proceso de compilación e integración continua, así con cada cambio realizado en nuestro sistema de gestión de la configuración, el servidor de integración continua deberá obtener la última versión del código del proyecto y de manera conjunta realizar la compilación de los componentes y ejecutar las pruebas unitarias. Es importante recalcar que deberá ser un servidor de integración continua quien ejecute esta acción y no los desarrolladores, con el fin de no afectar los tiempos de construcción ni consumir recursos innecesarios.

La ejecución de estas pruebas será el mecanismo con retroalimentación más temprana dentro de nuestro proceso de construcción, por lo que es muy importante cuidar elementos como la velocidad de ejecución y robustez de las pruebas.

¿Y los demás tipos de prueba?

Las pruebas unitarias no son el único tipo de prueba que se puede ejecutar dentro de un flujo continuo. Si bien es cierto que son las más baratas y efectivas, no son las únicas que deben ser tomadas en cuenta en un contexto completo. Por ejemplo, es sumamente complicado realizar una prueba de integración exclusivamente con tests unitarios ya que por su naturaleza son independientes, es en estas situaciones donde hay que considerar otras estrategias basadas en pruebas de caja negra.

Los otros tipos de prueba (funcionales, aceptación y desempeño) típicamente son costosas en tiempo o recursos de cómputo, por lo que no es recomendable integrarlas en su totalidad a nuestro flujo continuo de prueba.

Una buena técnica es hacer suites de pruebas dinámicas, es decir conjuntos de pruebas que se adapten a los cambios mediante un elemento de control. Por ejemplo, se puede implementar un mapa de la funcionalidad asociada a módulos y piezas de software en particular así como sus áreas directas de influencia. De esta forma es fácilmente rastrear qué tests ejecutar.

Asociando esto a un mecanismo de mensajes mediante la descripción del commit podemos indicar qué funcionalidad se afecta, de tal modo que al obtener los últimos cambios en nuestro código también obtenemos la información para armar dicha suite. Un ejemplo de mensaje sería el siguiente:

Número_de_issue Tipo_de_cambio Módulo_funcional prioridad

En el cual:

Número_de_issue: es el identificador de nuestro sistema de issues, por ejemplo el número de bug, el número de requerimiento, etcétera.
Tipo_de_cambio: indica de qué tipo es el código que estamos introduciendo, por ejemplo si es nueva funcionalidad, la solución a un bug, un cambio debido a una refactorización.
Módulo_funcional: es el identificador de alto nivel que nos dice la funcionalidad a la que afecta el cambio.
Prioridad: es una escala numérica que indica la relación entre nivel de riesgo del componente respecto a la frecuencia de uso.

Con lo anterior podemos reunir la información respecto a los tests a ejecutar, así como orden de ejecución de los mismos.

Adicional a lo antes mencionado es recomendable tener un mecanismo para ejecutar la totalidad de las pruebas. Aquí debemos ser muy cuidadosos con la frecuencia de ejecución, nuevamente debido al alto costo. Una buena práctica es ejecutarlas una vez a la semana, pudiendo ser los fines de semana ya que así al inicio de semana se tendría información más completa del estado real de la totalidad del proyecto.

La ejecución continua de las pruebas de desempeño es un asunto bastante debatible debido a que este tipo de prueba es aplicable generalmente a proyectos de software bastante maduros, por lo que su ejecución se debe programar acorde a la situación del proyecto.

Conclusión

La práctica de pruebas continuas enriquece y agrega valor a la integración continua. En este artículo he mostrado algunos aspectos clave. Te invito a conocer más sobre este fascinante tema.

Bio:

Christian Ramírez (@chrix2) es Ingeniero en computación por la UNAM y astrónomo aficionado. Actualmente es consultor de testing y dedica la mayor parte de su tiempo a ayudar a mejorar la práctica de testing en los proyectos donde participa. Es ponente regular en eventos internacionales sobre Software Testing y Agile. Es amante de python y ferviente creyente de que el conocimiento debe ser libre.

Estudio de Salarios SG 2012

Autor:

SG Software Guru

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Principal

SG periódicamente realiza un estudio para conocer el nivel de salarios y compensaciones que reciben los profesionistas de software en nuestra región, así como cuáles son los factores que más influyen en ésto.

Durante octubre de este año realizamos la 4ta edición de dicho estudio. En las páginas siguientes compartimos los resultados de éste.

Antes de iniciar con los detalles de sueldos y sus variaciones, les comentamos que los resultados aquí presentados se generaron en base a las respuestas provistas por los lectores de Software Guru a la encuesta de salarios que estuvo disponible en el sitio web de SG durante octubre del 2012. La encuesta fue contestada por 920 personas. Sin embargo, después de depurar datos, eliminar duplicados potenciales y dejar solamente a aquellas personas con empleo o actividad profesional formal nos quedamos con 813 respuestas. Las estadísticas presentadas en este reporte se generaron en base a ese total, 813. Sin más, lancémonos a los datos.

Salario

Los resultados del estudio arrojan que el salario mensual promedio de un profesionista de software en México es de $25,049 pesos mexicanos, es decir alrededor de 1,900 dólares (considerando un tipo de cambio de 13.20 pesos por dólar).

Este dato refleja un aumento del 8.6% respecto al salario promedio obtenido en la encuesta del 2010 ($23,052), lo cual nos hace pensar que en este periodo los salarios en nuestro sector aumentaron relativamente a la par de la inflación.

Otra estadística descriptiva de utilidad es la mediana, que en este caso es de $23,000 pesos. Entonces, de acuerdo a este resultado podemos estimar que el 50% de los profesionistas de

software de la región tienen un sueldo mayor a 23 mil pesos –también está la contraparte, que el 50% gana menos de 23 mil pesos, pero mantengamos la mira hacia arriba.

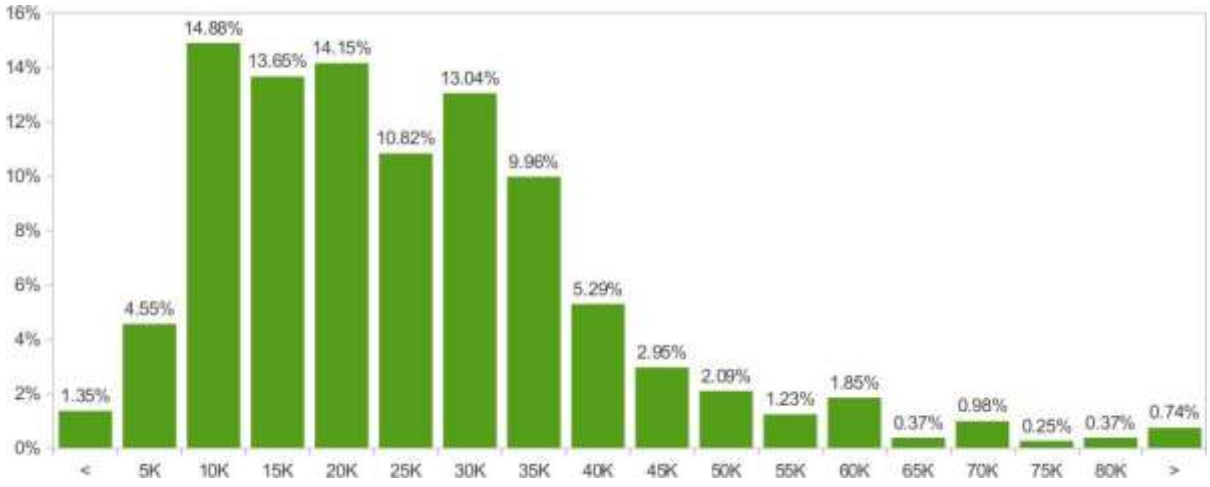


Figura 1. Histograma de frecuencias relativas para rangos de salario

La figura 1 muestra una gráfica que refleja la agrupación de salarios en rangos, indicando el porcentaje de personas que corresponde a cada rango. Esta gráfica nos muestra un par de revelaciones notables:

1. La agrupación con mayor frecuencia es la del salario de alrededor de 10 mil pesos mensuales.
2. Los 3 grupos con mayor frecuencia son los que corresponden a un salario mensual de 10 a 20 mil pesos. Este rango de los 10 a 20 mil pesos mensuales contiene a más del 40% de los encuestados.

Remuneración adicional

Aunque el principal componente de la remuneración laboral es el salario, sin duda otro elemento muy importante son las remuneraciones adicionales. Alrededor del 80% de los participantes reportaron recibir una compensación adicional económica en forma de bonos, aguinaldo, reparto de utilidades y otros. En promedio, la compensación extra recibida fue de $31,825 pesos, con una mediana de $15,000 pesos.

Esquema

La tabla 1 muestra los esquemas de compensación más populares entre los participantes, indicando el salario promedio así como el porcentaje de la población que representa cada grupo. Podemos ver que el esquema más popular es el de nómina.

Los resultados en años anteriores han arrojado que las personas empleadas bajo un esquema de honorarios perciben un salario mayor que las personas que están por nómina. En esta ocasión están prácticamente igual.

Por otro lado, a pesar de que en la encuesta no había una opción explícita para esquema mixto —cuando las personas perciben una parte de su compensación por nómina y otra parte por honorarios—, un número significativo de personas usaron la opción “Otro” y reportaron estar bajo dicho esquema. De haber estado como opción explícita, suponemos que esta cifra hubiera sido mucho mayor, ya que es un esquema que ha cobrado popularidad en los últimos años.

Esquema	Salario	Pct
Empleado de tiempo completo en nómina	$25,662.14	66.42%
Empleado de tiempo completo por honorarios	$25,506.06	19.80%
Soy socio/dueño de mi propia empresa	$31,063.27	6.03%
Independiente/Freelance	$19,444.44	3.32%
Esquema mixto	$26,713.20	2.46%
Becario	$9,021.64	1.72%
Medio tiempo	$8,000.00	0.25%

Tabla 1. Compensación mensual por esquema.

Variación

En términos de la variación que ha tenido el salario de cada profesionista de software durante los últimos 12 meses, el promedio calculado es de 10.9% con una mediana de 5%.

Tipo de organización

El tipo de organización a la que uno pertenece impacta el salario que se puede esperar, así como las prestaciones.

La tabla 2 muestra la descomposición por distintos tipos de organización. Tal como ha sucedido en ocasiones anteriores, los mejores salarios están en las empresas proveedoras de servicios. Aunque como sabemos, estas empresas típicamente son las que tienen las menores prestaciones y es común que paguen por honorarios o en esquemas mixtos con una parte en nómina y la otra por honorarios. Aunque el mayor porcentaje de profesionistas de TI todavía labora en áreas de sistemas de empresas de otros sectores, el porcentaje de profesionistas de TI que labora en empresas de outsourcing ha crecido continuamente desde la primera edición que realizamos esta encuesta. En la edición 2008 fue el 28%, en 2010 el 30% y ahora el 34%. Sin duda esto refleja como cada vez una mayor proporción del trabajo de TI es realizado por terceros.

Tipo organizacion	Salario	Pct
Proveedor de servicios de T.I. para terceros (outsourcing)	$29,490.16	33.95%
Distribuidor de software de terceros (canal)	$25,719.05	2.58%
Desarrollador de productos de software propios (ISV)	$24,652.36	15.62%
Área de sistemas/T.I. en una empresa o institución	$24,142.00	41.21%
Capacitación (empresa)	$15,457.75	0.98%
Institución educativa	$14,483.24	5.66%

Tabla 2. Salario por tipo de organización.

Geografía

La tabla 3 lista los estados de la República Mexicana donde se reportó el mayor salario promedio. Solamente hemos incluido los estados de donde se obtuvieron por lo menos 20 respuestas.

En años anteriores, Nuevo León y el Distrito Federal se han dividido la posición de liderazgo, sin embargo en esta ocasión es el estado de Querétaro el que reportó el mayor salario promedio, pasando del 3er lugar en 2010 al primer lugar en 2012.
Los estados de Campeche y Sonora también reportaron salarios promedio superiores a los 30 mil pesos (33 mil en Campeche y 31 mil en Sonora), pero decidimos no incluirlos en la tabla dado que la muestra de respuestas era menor a 20.

Fuera del DF y Estado de México, que tienen su propia inercia por la importancia del área metropolitana, el resto de los estados que reportan los salarios más altos son aquellos donde en los últimos años se han establecido más centros de desarrollo de software globales. Entonces, al parecer los centros de desarrollo de software de gran escala sí impactan los salarios locales de los profesionistas de software.

Estado	Salario	Pct
Querétaro	$32,668.61	4.18%
Nuevo León	$30,634.32	7.48%
Distrito Federal	$29,025.53	37.64%
Jalisco	$24,963.37	6.59%
Estado de México	$24,461.17	11.03%
Aguascalientes	$22,486.96	2.92%
Guanajuato	$19,040.36	2.79%
Puebla	$18,470.67	3.04%
Veracruz	$14,190.00	2.53%

Tabla 3. Salarios por entidad federativa.

Durante la encuesta obtuvimos respuestas de otros países fuera de México, los cuales nos arrojaron los siguientes salarios promedio:

1. Colombia: $1,470 dólares
2. Costa Rica: $2,600 dólares
3. Ecuador: $700 dólares
4. Estados Unidos: $5,500 dólares
5. Perú: $920 dólares

En cada uno de estos casos, la muestra de resultados es menor a 5, por lo que es demasiado pequeña como para ser confiable. Aún así, en la mayoría de los casos los resultados obtenidos parecen estar alineados con la realidad, con la excepción del caso de Costa Rica que parece un poco alto para la región.

Tipo de actividad

La tabla 4 muestra el salario promedio dependiendo del tipo de actividad realizada, así como el porcentaje de personas que indicaron realizar cada actividad. Cada persona podía escoger un máximo de 3 actividades, así que los porcentajes suman más de 100%.

Actividad	Salario	Pct
Preventa / Tech sales	$47,341.18	2.09%
Dirección de negocio	$42,826.56	10.58%
Venta y desarrollo de negocios	$37,140.32	3.81%
Mejora de procesos de software	$34,344.53	10.82%
Consultoría	$32,514.31	9.84%
Project Manager	$32,018.29	22.51%
Arquitectura de soluciones	$28,183.30	32.23%
Testing y QA	$26,252.22	9.72%
Análisis de requerimientos	$24,656.00	33.58%
Programación back-end	$23,943.01	33.95%
Admón de sistemas	$22,850.68	11.19%
Seguridad de sistemas	$21,940.18	3.44%
Programación front-end	$21,877.42	31.86%
Implantar sistemas empresariales	$21,730.57	8.49%
Análisis de datos	$20,964.14	16.73%
Capacitación	$19,938.67	7.75%
DBA	$18,098.76	19.19%
Diseño gráfico	$13,461.90	2.58%

Tabla 4. Salario por tipo de actividad.

La actividad que resultó con mejor compensación es la preventa, también conocida como tech sales. La preventa consiste en realizar la parte técnica para soportar la venta de un proyecto o solución. Esto incluye tareas como: ayudar a los clientes a verificar que un producto en cuestión resuelve sus necesidades de manera satisfactoria, realizar demos de productos, preparar propuestas de arquitectura para una solución y brindar capacitación.

Nos llama la atención que las personas que realizan testing aparecen con una mayor compensación que los programadores; este no era el caso hasta hace pocos años. Típicamente en organizaciones con poca madurez —donde no se le da mucha importancia al testing y en el mejor de los casos se hace de forma manual—, el testing se deja en manos de las personas con menor experiencia o nivel. En cambio, en organizaciones maduras, el testing es una actividad que requiere experiencia y especialización. Así que para nosotros es un signo de madurez de la industria local que los testers perciban una compensación mayor que los programadores.

Otro punto que resalta de este desglose es que las actividades relacionadas con gestión y análisis de datos están hacia el fondo de la lista. Esto nos hace creer que el nivel de análisis de datos realizado por las organizaciones locales es bastante básico. A nivel mundial esta es un área que está cobrando gran importancia, impulsada por la tendencia del Big Data. De hecho, en Estados Unidos una de las carreras con mayor futuro es la del "científico de datos". Así que esperamos que esta actividad aumente su valor significativamente hacia los próximos años.

A pesar de la importancia que ha cobrado el aspecto visual, especialmente en las aplicaciones móviles y web, podemos notar que el diseño gráfico sigue siendo la actividad con el menor salario promedio. A nuestros ojos,

el principal problema con el diseño gráfico de software es que hay una gran cantidad de personas que lo hacen (aunque la cantidad de personas que lo hagan bien sí es escasa).

Experiencia

La tabla 5 muestra el salario promedio en base a los años de experiencia. Que conforme se acumule mayor experiencia se pueda obtener un salario mayor no es ninguna sorpresa. Sin embargo, lo que nos llama la atención en este vista de los datos es el salto notable que hay entre el grupo con 1 a 3 años de experiencia y el siguiente grupo (3 a 5 años); estamos hablando de un aumento de más del 50%. Este umbral es el que típicamente diferencia a un desarrollador "novato" de otro que por lo menos ya tuvo sus "primeros tropezones". Es por ello que la mayoría de las ofertas de empleo que se pueden encontrar en bolsas de trabajo para desarrolladores de software piden "mínimo 3 años de experiencia". Así que si te encuentras en este grupo, mantén la calma y enfócate en aprender.

Experiencia	Salario	Pct
Más de 20 años	$41,103.16	9.10%
Más de 10 años	$31,445.19	26.08%
5 a 10 años	$26,736.39	29.89%
3 a 5 años	$19,491.91	16.24%
1 a 3 años	$12,713.35	16.73%
Menos de un año	$11,299.38	1.97%

Tabla 5. Salario por experiencia.

Género

Como podemos apreciar en la tabla 6, que nos muestra un desglose por género, las mujeres representan un 16% de la población de profesionistas de software, y en promedio perciben una compensación 30% menor que los hombres.

Género	Salario	Pct
Femenino	$20,350.52	16.56%
Masculino	$26,513.67	83.44%

Tabla 6. Salario por género.

Esta diferencia de 30% parece bastante grande, así que realizamos un análisis a mayor profundidad sobre este caso. Fue así que encontramos que en grupos que realizan el mismo tipo de actividad y con el mismo nivel de experiencia, el salario entre hombres y mujeres es similar, y que incluso en algunos casos el salario reportado por mujeres era mayor. Por ejemplo, el salario reportado para un programador front-end de sexo masculino con 3 a 5 años de experiencia es de $21,704 pesos mientras que el de una mujer es de $21,980. Un caso más dramático: un project manager hombre con 5 a 10 años de experiencia reporta un salario promedio de $26,237 mientras que el de una mujer es de $33,014.

Encontramos también que mientras que el 70% de los hombres tiene más de 5 años de experiencia, esta cifra se reduce al 50% en el caso de las mujeres. Esto nos indica que un porcentaje significativo de mujeres abandona su carrera como profesionistas de software antes de 5 años.

Por otro lado, encontramos que las mujeres están concentradas en actividades como el diseño o la capacitación, que están en el fondo de la tabla de ingresos por actividad. Posiblemente esto se deba a que estas son actividades que tienden a brindar mayor flexibilidad en cuestión de horarios y/o trabajo remoto, lo cual es conveniente para mujeres con hijos.

En conclusión, encontramos que en efecto el salario promedio percibido por una mujer en nuestra industria es significativamente menor que el que percibe un hombre; pero esto no necesariamente es porque se valore menos su trabajo, sino porque en general su carrera profesional no llega a los roles y niveles de experiencia mejor compensados.

Habilidades y conocimientos

Las tablas 7, 8 y 9 muestran un desglose del salario promedio dependiendo del expertise técnico en distintos lenguajes de programación, bases de datos y plataformas. Por su parte, la tabla 10 se basa en las certificaciones más populares.

En general, podemos ver que las tecnologías fuertemente usadas en el ámbito empresarial siguen obteniendo un salario mayor que aquellas usadas en las apps móviles y hacia el consumidor.

Adicionalmente hemos incluido una tabla específica (tabla 11) para analizar el impacto del dominio del idioma inglés, donde podemos notar claramente como afecta el salario percibido.

Lenguaje	Salario	pct
Cobol	$34,462.67	6.64%
Objective C	$31,853.33	3.69%
Ruby	$31,790.32	3.81%
Ensamblador	$31,003.85	3.20%
Python	$30,729.17	2.95%
C	$30,163.01	18.33%
Shell scripting	$30,018.52	6.64%
PL/SQL	$27,469.13	29.52%
Javascript	$27,370.60	31.12%
Java	$27,365.79	31.12%
C#	$27,345.03	25.58%
Visual Basic	$26,658.46	29.03%
Delphi	$25,931.11	5.54%
Actionscript	$23,246.82	4.06%
PHP	$23,204.34	19.80%

Tabla 7. Salario por lenguaje de programación

Base de datos	Salario	Pct
Informix	$35,839.47	4.67%
DB2	$33,002.25	7.50%
Oracle	$30,935.64	23.49%
Sybase	$28,625.81	3.81%
Mongo	$27,835.04	4.92%
Postgresql	$27,619.87	11.44%
SQL Server	$27,372.14	43.05%
MySQL	$24,455.52	27.55%

Tabla 8. Salario por base de datos

Plataforma	Salario	Pct
Mainframe	$33,117.65	4.18%
Java (JVM)	$28,737.56	24.97%
Web	$27,188.57	44.28%
.NET	$27,029.42	28.41%
iOS	$26,885.66	3.57%
Android	$24,109.16	5.41%
Air	$21,138.28	2.21%

Tabla 9. Salario por plataforma de especialización

Certificación	Salario	Pct
IBM Websphere	$57,000.00	0.98%
Enterprise Architect (SEI/IASA)	$47,125.50	0.98%
Project Manager (PMP)	$42,406.17	5.78%
Business Analyst	$33,475.15	2.46%
ITIL	$32,301.47	7.01%
Scrum Master	$31,922.21	3.44%
UML	$30,744.03	4.06%
Testing y QA (SQA, ISTQB)	$30,073.55	3.57%
Microsoft Expert	$28,474.22	6.03%
Java (Oracle/SpringSource)	$28,437.28	10.82%
SAP	$27,490.00	1.97%
Seguridad (CISSP, Ethical Hacker)	$26,124.82	2.09%
TSP/PSP	$25,593.75	1.97%
Microsoft Associate	$25,390.38	6.40%
Oracle DBA	$25,384.06	1.97%
Linux (LPI/RHCE)	$25,200.00	1.60%

Tabla 10. Salario por certificación obtenida

Nivel de Inglés	Salario	Pct
Avanzado	$34,528.37	19.31%
Intermedio	$24,757.68	59.16%
Principiante	$19,190.26	21.53%

Tabla 11. Salario por nivel de inglés.

Conclusión

A través de estas páginas hemos conocido cuál es la compensación que puede esperar un profesionista de software en nuestra región, y cuales son los principales factores que la impactan. Esperamos que esta información te sea de ayuda para evaluar mejor tu situación actual y planear mejor tu carrera.

A las empresas que emplean profesionistas de software, esperamos que esta información les sea de utilidad para calibrar sus tabuladores de compensación.

Durante el estudio de salarios se recopilaron muchos otros datos que no hemos analizado todavía. En próximos números de SG los presentaremos, y también los haremos disponibles por medio de nuestro sitio web.

Aplicando Criterios Ágiles a la Calidad

Autor:

Jordana Villegas Sosa

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Código Innovare

Hoy es la presentación del nuevo sistema de software. El cliente usuario y el cuerpo directivo estarán presentes. Llegó la hora, el cuerpo directivo se muestra interesado, el presentador realiza la demostración de acuerdo al “happy path” (escenario de uso ideal sin condiciones de excepción), pero no contemplaron que el ambiente en el cual se iba a presentar no era el mismo donde se desarrolló y no se había probado ahí. De pronto, a media presentación … ¡aparece una pantalla de error y el sistema se cierra repentinamente! Se escuchan murmullos y se hace sentir la molestia de los presentes, mientras el presentador trata de minimizar la falla y terminar la presentación de manera “normal”.

El escenario descrito anteriormente es demasiado común. La deficiencia de pruebas de software conlleva a: retrasos en la entrega del producto; fallas del software en la operación; desacreditación de posicionamiento del producto y la compañía responsable; costos no contemplados por las fallas del producto, provocando que el presupuesto del proyecto rebase lo contemplado; o en el peor de los escenarios puede ser que las fallas en el sistema provoquen la pérdida de vidas humanas.

No podemos seguir creyendo que es imposible comprobar la calidad del producto antes de que se está ejecutando o validando con el usuario. Es muy importante incluir como una actividad las pruebas de software

durante el desarrollo del producto, y no como un proceso de pruebas opcional o de trámite para después de ser liberado.

Por donde empezar

Ya nos queda más que claro que las pruebas de software son una necesidad que beneficia tanto a la imagen como a la economía y viabilidad de nuestra empresa, pero ahora, ¿Cómo podemos realizar esta planeación de manera eficiente o con procesos que podamos llamar ágiles? ¿Cómo poder tener un equipo multidisciplinario ágil? Es decir, es necesario que desde el punto de vista de la calidad y pruebas se tenga una mentalidad ágil.

Para ser ágil es necesario que todo el equipo sea consciente de la importancia que tienen las pruebas para alcanzar una calidad alta. Las actividades de prueba no deben ser una etapa al final del proyecto, sino que deben ser parte de todo el ciclo de desarrollo.

Existen distintos métodos y niveles de prueba de software que deben de ser contemplados en el plan de pruebas de cualquier proyecto de desarrollo de software. Entre los métodos de prueba más comunes están las pruebas estáticas, dinámicas, de caja negra, de caja blanca; cada uno con distintas técnicas y enfoques. Las actividades y diferentes técnicas que se utilizarán deben de planearse de manera eficiente dependiendo del contexto de cada proyecto.

Es necesario contar con un plan de pruebas que incluya la definición, preparación y ejecución de pruebas a lo largo de todo el ciclo de vida del proyecto. Debemos incorporar al personal responsable de las pruebas en las reuniones como el "sprint planning" (planificación de las tareas a realizar en la iteración) para que entienda qué se va a construir y bajo qué condiciones va a ser aceptado. Todo requerimiento debe estar asociado a un conjunto de pruebas de aceptación, de manera que en todo momento se sepa claramente si dicho requerimiento ha sido correctamente resuelto o no.

En las etapas tempranas podemos empezar con pruebas de exploración con las primeras funcionalidades que se vayan completando, como prototipos, que nos sirve para asegurar la retroalimentación de la iteración.

La persona responsable de la calidad de un producto de software debe mantenerse centrada en la visión global del proyecto, definiendo y ajustando la estrategia de prueba de manera que esté alineada a dicha visión. También debe contar con el valor y las bases para decirle al equipo de desarrollo cuando algo no es correcto.

Conclusión

Para hacer testing ágil es necesario que todo el equipo de trabajo esté consciente de la importancia que tienen las pruebas para alcanzar alta calidad en un producto de software. Si todo el equipo está preocupado por el desarrollo y generar versiones por cada modificación o parche al sistema, el testing no encaja en ese equipo. Establecer un esfuerzo de pruebas ágiles requiere compromiso y colaboración.

Referencias

[1] L. Crispin & J. Gregory, Agile Testing: A Practical Guide for Testers and Agile Teams, Addison-Wesley Professional, 2009.
[2] X. Albaladejo, "Planificación ágil vs planificación tradicional" http://swgu.ru/sg38r1

Bio:

Jordana Villegas Sosa (jordana.villegas@infotec.com.mx) es responsable de la especialidad de prueba de software y transferencia de conocimientos en INFOTEC. Es Ingeniero en Sistemas Computacionales por el Instituto Tecnológico de Acapulco, institución donde también fue instructora.

Gustavo y Einstein

Autor:

Masa K. Maeda

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Agilidad

Gustavo es un excelente profesional pero a pesar de ello perdió su trabajo como resultado de una estrategia de reestructuración, derivada de la económica actual de la empresa donde trabajaba.

Un día durante su periodo de búsqueda de empleo, Gustavo decidió bañar a su perro, llamado Einstein. Comenzó por alistar los materiales necesarios: manguera, cubeta, shampoo, toalla y cepillo. En un cierto instante durante la bañada, Einstein aprovechó un momento de distracción de Gustavo para escaparse (Einstein odia ser bañado) y correr por el jardín mientras Gustavo lo perseguía. Después de un par de minutos, Gustavo logró agarrar a Einstein para continuar bañándolo. Aunque Gustavo ya había lavado las patas de Einstein, tuvo que hacerlo de nuevo porque se ensuciaron durante la escapada. Al terminar, Gustavo enjuagó el área donde bañó a Einstein y regresó a su lugar las cosas que utilizó. Vale la pena mencionar que Gustavo planeó el baño de Einsten para no efectuarlo al mismo tiempo que un par de entrevistas telefónicas que tenía pendientes, además de ir a recoger a un amigo al aeropuerto.

La actividad de bañar a Einstein tiene una estructura igual al comportamiento y actividades dentro de proyectos. La actividad de valor agregado es bañar a Einstein y existen tres tipos de costos asociados a la actividad de valor agregado:

- **Costos de transacción**. Son los costos de la actividad de valor no agregado que se deben llevar a cabo antes y después. En el caso de Gustavo y Einstein, previamente conseguimos y alistamos los materiales (cubeta, shampoo, toalla) y después del baño los limpiamos y guardamos. En proyectos de software, las actividades previas incluyen adquirir equipo, preparar infraestructura, contratar personal, y las actividades posteriores pueden ser restaurar el equipo de cómputo, limpiar el ambiente de desarrollo, etcétera.
- **Costos de coordinación**. Son los costos asociados a actividades de gestión y logística (asegurar que las entrevistas no sean a la hora de bañar a Einstein o de ir a recoger al amigo al aeropuerto) tales como comunicación entre stakeholders, coordinar el involucramiento del cliente, adquisiciones, etcétera.

- **Costos de carga de falla**. Son los costos que resultan de eventos imprevistos de valor no agregado dentro de la actividad de valor agregado (Einstein sale corriendo por lo que hay que perseguirlo, tomarlo, y volver a lavar las patas) tales como fallas de equipo o equipo inadecuado, re-trabajo y otros.

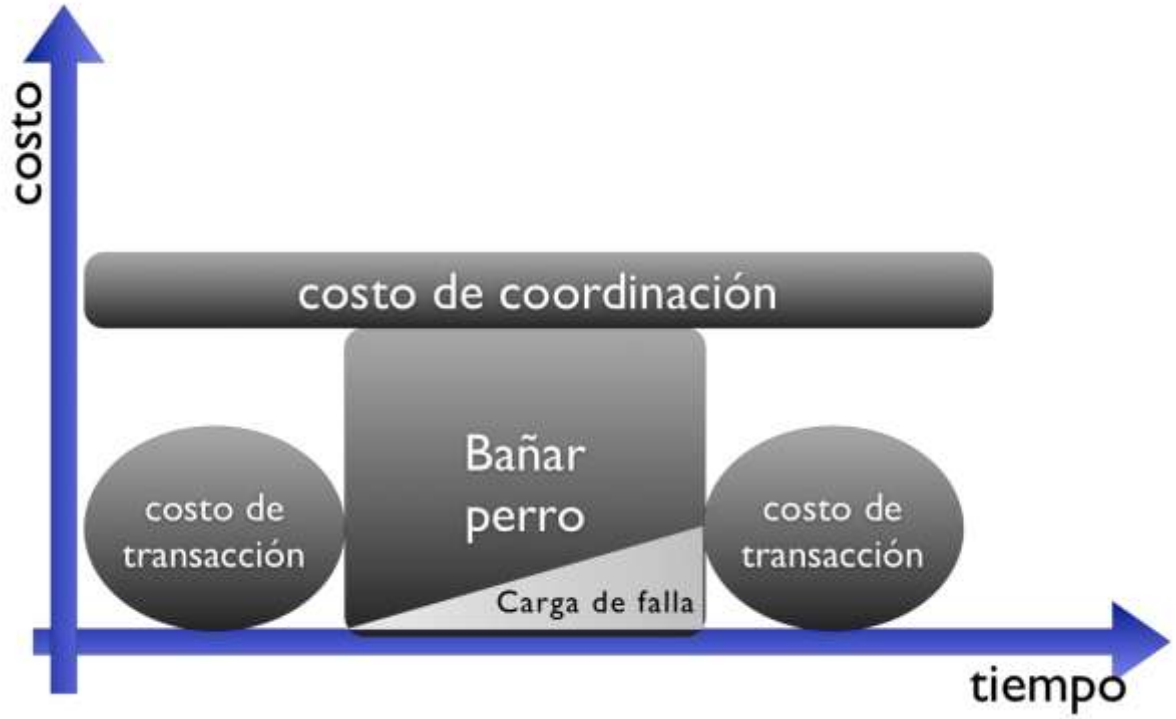


Figura 1. Ejemplo Costos de Coordinación

Sigamos con nuestro amigo Gustavo. Resulta que en la zona donde vive hay muchas personas con buena capacidad económica y que consienten mucho a sus perros. Esto le da la idea de iniciar su propio negocio de baño de perros, lo cual él cree que le permitirá tener ingresos suficientes para subsistir mientras obtiene un nuevo empleo. Gustavo creó su estrategia de promoción en línea y comenzó a obtener negocio, pero lo que nunca imaginó fue que su negocio prosperaría tanto. En poco tiempo estaba bañando 20 perros por día y la demanda continuaba creciendo.

Gustavo tenía problemas de escalamiento: entre más perros tenía que bañar, menos tiempo tenía para bañarlos. Comenzó a contratar gente en proporción al crecimiento del negocio, para entonces con un promedio de 50 perros diarios. El negocio en lugar de prosperar comenzó a sufrir y las ganancias comenzaron a reducirse.

Ese comportamiento negativo inesperado se debió a que al incrementar el monto de trabajo de valor agregado también se incrementan los costos de transacción, operación y carga de falla. Pero el comportamiento de crecimiento es distinto para cada uno de ellos:

Los costos de transacción crecen de manera casi lineal, por lo que no son particularmente difíciles de manejar, pero aún así su crecimiento afecta la economía del sistema.

Los costos de operación crecen de manera exponencial. Esto se debe a que entre mayor cantidad de trabajo a coordinar se requieren de más y más elementos de coordinación. Es por eso que grandes empresas y grandes proyectos tienden a tener muchos niveles organizacionales. Éste tipo de costo es el que más afecta la economía del sistema.

Los costos de carga de falla crecen en proporción mayor al del crecimiento del trabajo de valor agregado. Esto quiere decir que conforme la carga de trabajo crece, más y más tiempo es dedicado a re-trabajo y otras actividades correctivas, por lo que la economía del sistema baja significativamente.

Figura 2. Ejemplo de costos de coordinación con escalamiento.

Moraleja

Una característica fundamental de las metodologías ágiles y lean es el efectuar el trabajo en pedazos pequeños y entregar el valor generado lo antes posible sin sacrificar calidad. Es decir, los proyectos grandes se descomponen en partes pequeñas que se pueden implementar fácilmente. El resultado desde el punto de vista económico es: pequeños costos de transacción, operación y carga de fallas que son mucho más fáciles de manejar y eliminan en gran medida los problemas de escalamiento de proyectos medianos y grandes.

Bio:

El Dr. Masa K. Maeda es el CEO de Valueinnova basada en el Silicon Valley y con oficina satélite en Panamá. Masa es la persona de mayor influencia e impacto en adopción y entrenamiento empresarial de Agile y Lean en Ibero-América, además de ofrecer servicios también en Norte América, Europa, Asia y África. Su cartera de clientes incluye de Fortune 100 a PyMEs. Masa obtuvo el doctorado y maestría en Japón y ha radicado en USA por casi 18 años.

Conceptos de Diseño Patrones, Tácticas y Frameworks

Autor:

Humberto Cervantes

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Arquitectura

En la columna dedicada al diseño de la arquitectura (SG #29) se mencionó que el proceso de diseño involucra la toma de decisiones que dan lugar a las estructuras del sistema. Estos incluyen aspectos tales como los patrones de diseño, las tácticas y los frameworks. En esta columna revisaremos en más detalle estos conceptos y discutiremos la relación que existe entre ellos.

Patrones de diseño

El concepto de patrones de diseño se origina en la arquitectura de edificaciones y fue popularizado por el arquitecto Christopher Alexander y sus colegas en el libro “A Pattern Language: Towns, Buildings, Construction” [1]. Alexander presenta una colección de patrones que son soluciones conceptuales a problemas recurrentes de diseño que se aplican a distintas escalas. Los autores establecieron la estructura de un catálogo de patrones de diseño: cada patrón tiene un nombre que lo identifica; una descripción del problema o contexto; una descripción de la solución conceptual; una discusión de las implicaciones del patrón; y su relación con otros patrones.

El conjunto de nombres que identifican a los patrones dan lugar a un lenguaje de diseño. La idea es que con solo utilizar el nombre de un patrón, un diseñador hace referencia a la solución conceptual que representa el

patrón de forma rápida, sin tener que describir todos los detalles (esto es, si las demás personas hablan el mismo lenguaje de diseño).

El concepto promovido por Alexander fue retomado en el software y popularizado en el libro "Design Patterns: Elements of Reusable Object Oriented Software" [2]. Debido a que son cuatro autores, a este libro también se le conoce como el "Gang of Four" (GoF). En el GoF se documentan 23 patrones que son soluciones conceptuales a problemas recurrentes de diseño de software. El concepto de patrones de diseño resultó extremadamente popular y después del GoF han aparecido una gran cantidad de libros de patrones de diseño enfocados a temas específicos como pueden ser diseño de arquitecturas, creación de sistemas distribuidos, sistemas tolerantes a fallas, seguridad, integración de sistemas, etcétera. Existen cientos de patrones de diseño documentados y una dificultad ahora es elegir patrones específicos de entre la gran variedad.

Para ejemplificar el uso de patrones consideremos el caso de un sistema de subastas en línea en donde se tiene una gran cantidad de usuarios que están realizando ofertas de un producto desde su navegador. En todo momento, los usuarios deben poder ver el precio actual del artículo que les interesa y cuando alguno de los usuarios realiza una oferta, todos los demás usuarios deben de poder ver el último precio que se ofrece por el artículo

Este es un problema común y justamente el GoF documenta un patrón llamado "Observador" (Observer) cuya intención es "definir una dependencia entre objetos de uno a muchos de tal forma que cuando cambia el estado del objeto observado, todos sus dependientes sean notificados y actualizados de forma automática". La intención de este patrón es muy acorde al problema que intentamos resolver. La figura 1 muestra la estructura base de la solución propuesta por este patrón.

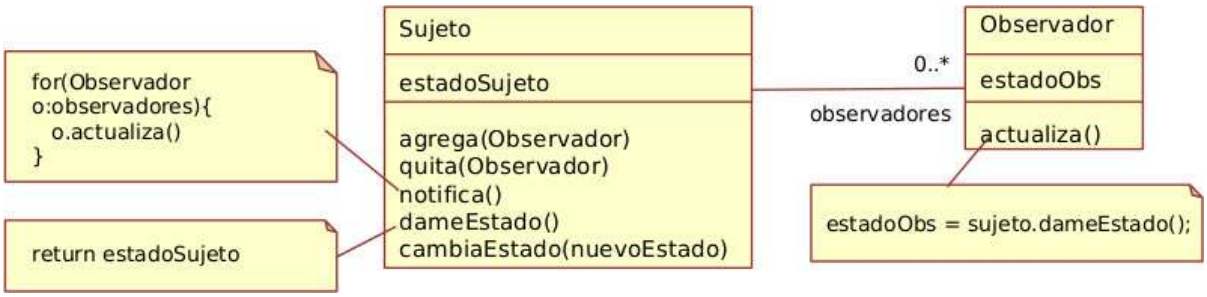


Figura 1. Estructura del patrón Observador

Como se puede apreciar, existen varios observadores que están interesados en conocer cambios en el estado del sujeto. Los observadores se pueden registrar como interesados en recibir notificaciones de cambios del estado del sujeto mediante el método agrega(). Cuando ocurre un cambio en el estado del sujeto, se invoca el método notifica() del mismo. Este método tiene como consecuencia que a cada uno de los observadores registrados se les envíe una notificación de cambio mediante la invocación al método actualiza(). Al recibir la notificación, los observadores solicitan el nuevo estado del sujeto y actualizan su estado local de acuerdo al nuevo estado del sujeto.

En el contexto del sistema de ejemplo, podemos suponer que la subasta es el sujeto, y los participantes son los observadores. Cuando un usuario envía una nueva oferta, cambiará el estado del sujeto y se enviará un aviso a los observadores para que actualicen su interfaz. De esta manera vemos como un patrón de diseño nos puede ayudar a resolver un problema de diseño de forma relativamente simple y sin que tengamos que "reinventar la rueda".

Tácticas

Las tácticas para el diseño de arquitecturas de software son un concepto que fue popularizado en el libro "Software Architecture in Practice" de Bass y sus colegas del Software Engineering Institute (SEI) [3]. Las tácticas son decisiones de diseño que influyen en el control de la respuesta de un atributo de calidad. A diferencia de los patrones de diseño, las tácticas son soluciones menos detalladas y están enfocadas a atributos de calidad específicos. Las tácticas son presentadas de forma jerárquica como se muestra en la figura 2. A la raíz del árbol se presenta la categoría de atributo de calidad y debajo de esta categoría se muestra una serie de "preocupaciones" (concerns). Finalmente, debajo de las preocupaciones se encuentran las tácticas específicas. La manera de entender esto es la siguiente: si se tiene que satisfacer el atributo de calidad de desempeño, una posible preocupación es influir sobre la demanda de recursos. Para lograrlo una posible opción es incrementar la eficiencia computacional y esto se puede llevar a cabo por ejemplo mediante algoritmos más eficientes.

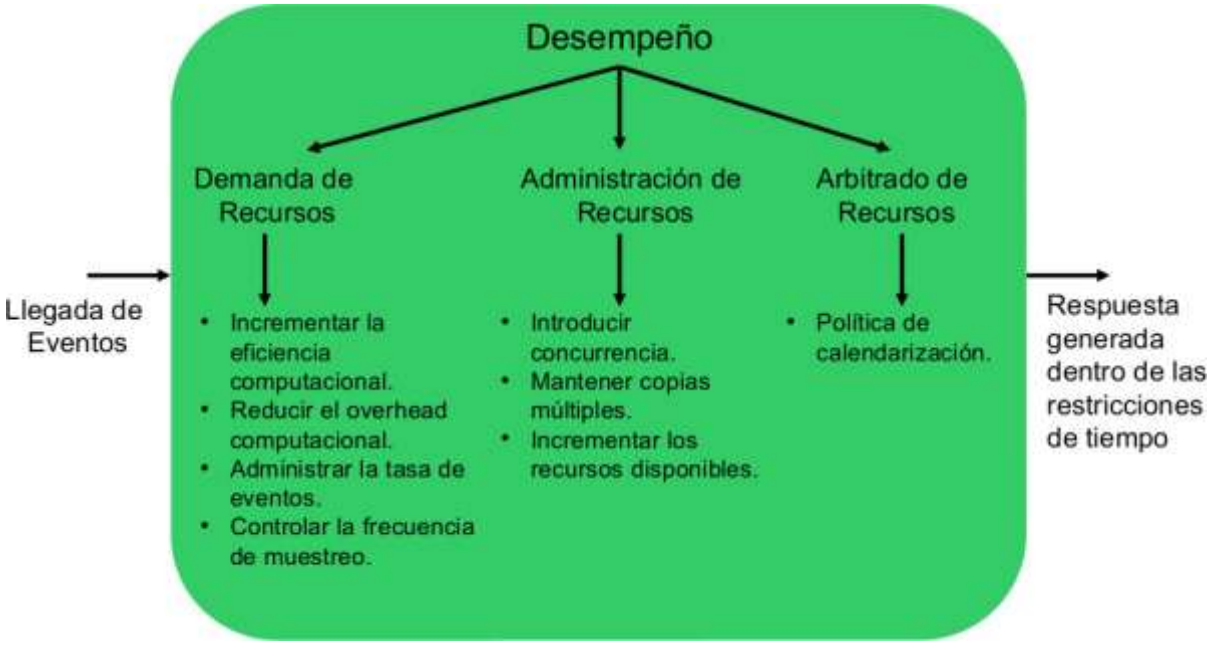


Figura 2: Tácticas de desempeño

Las tácticas se aplican en conjunto con los patrones durante el diseño de la arquitectura de un sistema. Para entenderlo, retomemos el ejemplo anterior y consideremos que existe un escenario de atributo de calidad de desempeño como el siguiente:

- "Un usuario envía una oferta en un momento en que hay otros 500 usuarios conectados, la oferta es procesada exitosamente y transcurren menos de 20 ms entre la recepción de la petición y el envío de las actualizaciones"

Para poder lograr un escenario como este, podemos referirnos al catálogo de tácticas. En este caso la preocupación principal es la administración de recursos, que en este caso son el procesador y los datos que corresponden al estado. Dos tácticas nos pueden apoyar para resolver este problema: la introducción de concurrencia y el mantenimiento de copias múltiples. De esta forma, en vez de procesar cada petición de forma secuencial y esperar a que terminen de ser notificados todos los programas cliente antes de procesar la siguiente oferta, podemos realizar este proceso de forma concurrente. Por otro lado, el mantenimiento de copias múltiples se puede lograr, por ejemplo, mediante la introducción de un cache de tal forma que cada que se tenga que notificar a los programas cliente no sea necesario tener que acceder a la base de datos para recuperar el estado.

De esta forma podemos ver cómo los patrones y las tácticas se combinan para resolver problemas de diseño permitiendo así estructurar el sistema y satisfacer los atributos de calidad.

Frameworks

Si bien los patrones y tácticas son fundamentales en la creación arquitecturas de software, estos conceptos resultan abstractos y en ocasiones no es claro cómo conectarlos con la tecnología que se usa para el desarrollo. Es así que otro elemento clave son los Frameworks. Estos son elementos reutilizables de software que proveen funcionalidades genéricas enfocadas a resolver cuestiones recurrentes. Existen frameworks para resolver distintos aspectos de una aplicación; por ejemplo, Java Server Faces (JSF) es un framework para crear interfaces de usuario en aplicaciones web. Existen muchos otros frameworks para resolver todo tipo de aspectos, como el manejo de XML, el acceso a bases de datos relacionales, manejo de concurrencia, etcétera.

Los frameworks típicamente se basan en patrones y tácticas. Un ejemplo de ello es el framework Swing para creación de interfaces de usuario en Java, que aplica patrones como Modelo-Vista-Controlador (MVC), Observador y Composite; e incorpora tácticas de modificabilidad y usabilidad tales como la generalización de módulos y la separación de elementos de interfaz de usuario.

Conclusión

El diseño de arquitecturas de software se basa en la toma de decisiones que involucran la selección de conceptos con el fin de satisfacer los requerimientos que influyen sobre la arquitectura. Dentro de estos conceptos se encuentran los patrones de diseño, las tácticas y los frameworks. Uno de los retos importantes del diseño es la selección oportuna y adecuada de dichos elementos, en particular porque existen cientos de patrones de diseño y una gran variedad de frameworks. Además de asegurarnos de que los conceptos de diseño seleccionados satisfagan los requerimientos que influyen sobre la arquitectura, debemos tomar en cuenta el impacto negativo que pudieran tener en la satisfacción de los atributos de calidad. Un ejemplo de ello es la introducción de un framework de persistencia que facilita el mapeo orientado objetos a relacional, y por lo tanto promueve la modificabilidad, pero posiblemente influye negativamente sobre el desempeño.

Referencias

[1] Alexander, Ishikawa & Silversteing, “A Pattern Language: Towns, Buildings, Construction”, Oxford University Press, 1977.

[2] Gamma, Helm, Johnson & Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional, 1994.

[3] Bass, Clements & Kazman, “Software Architecture in Practice, 2nd Edition”, Addison-Wesley Professional, 2003.

Bio:

El Dr. Humberto Cervantes es profesor-investigador en la UAM-Iztapalapa. Además de realizar docencia e investigación dentro de la academia en temas relacionados con arquitectura de software, realiza consultoría y tiene experiencia en la implantación de métodos de arquitectura dentro de la industria de desarrollo nacional. Ha recibido diversos cursos de especialización en el tema de arquitectura de software en el Software Engineering Institute, y está certificado como ATAM Evaluator y Software Architecture Professional por parte del mismo. www.humbertocervantes.net

Crónica de una Certificación Fracasada de MoProSoft

Autor:

Alfredo Lozada Carrillo

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Procesos

Este artículo describe mis experiencias y puntos de vista ante el intento de certificación de Moprosoft de una empresa en la que participé. El proyecto se gestionó desde la dirección general y fue presentado en una reunión en conjunto con representantes de la compañía consultora que nos apoyaría para la implantación. Como toda reunión de mejora, el proyecto se presentó con los honores merecidos. Siendo honestos, el posicionamiento en esos momentos era bueno y las perspectivas de crecimiento bien valían la pena invertir el tiempo necesario para lograr esta certificación, afianzando así el prestigio de la empresa.

El proyecto tenía un año para consolidarse y todos con el ánimo hasta el cielo comentábamos que lo lograríamos en seis meses.

Aproximadamente después de un mes de reuniones con el director y personal de la consultoría, tiempo en el que nosotros desconocíamos en realidad de qué se trataba el proceso de certificación, nos preguntábamos en qué momento iniciaríamos realmente, a la vez que el trabajo continuaba con su habitual rutina.

La siguiente reunión relacionada con la certificación, fue para presentar el calendario de capacitación que impartirían los consultores, programando sesiones diarias por 15 días. Una vez completada la capacitación y asignados los roles que tendríamos, la empresa consultora nos dejó a nuestra suerte. Fue así que el proyecto fue archivado sin seguimiento alguno por más de tres meses.

Llegamos a los seis meses que habíamos proyectado sin contar con un centímetro de avance real. Hasta ese momento comenzamos a retomarlo, realizando reuniones extraordinarias de trabajo, minutas y documentos a diestra y siniestra. El pánico y la desesperación comenzó a envolvernos, por una parte las asesorías eran cada día más esporádicas y como en toda empresa pequeña existían personas que debían cubrir tres y cuatro roles ya que no existía más personal. Hasta este punto fue cuando pudimos dimensionar las implicaciones que tendría el lograr implantar cada uno de los procesos que dicta la norma así como la dependencia entre los diferentes departamentos existentes. Es decir, necesitábamos comenzar a trabajar en equipo y después de casi siete meses perdidos en planes, capacitación y asignaciones, nos dimos cuenta que no existía esa cultura de trabajo, así que necesitábamos cambiarla con tan solo cinco meses restantes.

¿Y la base de conocimientos?

Para este tiempo nos percatamos de un elemento esencial para la evaluación y más importante para la implantación: hacía falta la herramienta que serviría como base de conocimientos. Ante esta necesidad y con el tiempo ya consumido era necesario buscar una herramienta ya elaborada para tal fin, con una característica importante: sin costo. La empresa consultora (como mago que saca un conejo de la chistera), promovió su herramienta ya probada y certificada para tal fin con la gran desventaja de que implicaba un costo. Al final decidimos utilizar un software open source llamado Owl Intranet Knowledgebase. La puesta a punto fue rápida así como la capacitación para su uso, sin embargo el uso adecuado y la generación de documentos aún no quedaba clara, así que se comenzó a ingresar todos los documentos que ya habíamos elaborado previamente.

Pre-evaluación y resultado

Con una pre-evaluación programada por la empresa consultora, el trabajo aumentó. Cada quien comenzó a subir sus documentos correspondientes; no importaba que proyecto presentaríamos ni la correspondencia entre unos y otros.

Como era de esperarse, el resultado de la pre-evaluación fue un rotundo fracaso: una pila de documentos aislados, ninguna correspondencia entre el plan de negocio y el plan de trabajo, proyectos fuera de tiempo, correos y documentos donde no coincidían las fechas de su elaboración y ninguna evidencia real correspondiente a la realización de los procesos ni evidencia de que la cultura organizacional había cambiado.

Después de más de un año de trabajo, el resultado se tradujo en un personal desgastado ante la presión del proyecto, problemas ante una empresa de consultoría que generó gastos de asesoría y capacitación, una base de conocimientos desechada y otra en proceso de fabricación, dos socios disgustados, dos gerentes despedidos y una empresa que continúa funcionando como negocio de pueblo: “mientras haya dinero en la caja y producto en los estantes, vamos bien”.

Lecciones aprendidas.

Después de este breve relato y de haber pasado por este proceso, me encuentro en condiciones de puntualizar las decisiones y acciones que a mi parecer fueron un error y al mismo tiempo señalar acciones que podíamos haber realizado para lograr un mejor resultado.

La Gestión. Como punto inicial puedo mencionar que el proyecto se gestionó desde la dirección general, pero creo que no se inició con el deseo de un cambio en metodología de trabajo. Dio la impresión de ser un proyecto similar a un trámite administrativo que se presenta documentación, se paga y se espera a la resolución. No existió un conocimiento de las implicaciones que llevaría el proceso ya que era necesario contemplar los cambios que se tendrían en la relación con los clientes, el esfuerzo para cumplir con los documentos de trabajo, las relaciones de los diferentes departamentos, la comunicación y dependencia entre departamentos y sobre todo la metodología de trabajo en equipo que nos permitiera confiar y comprender el trabajo de nuestros colaboradores.

Confianza sin realismo. Otro factor que afectó el proyecto fue el exceso de confianza. En plática de escritorio cualquier proyecto se ve fácil, sin embargo la verdad se presenta durante el trabajo y en este caso al tratarse de un cambio que nadie conocía no podíamos darnos el lujo de descuidarlo. Era necesario un mayor esfuerzo y seguimiento al proceso, pero ante la descripción de procesos que propone la norma el inicio lo debe marcar la Gestión de Negocio ya que a partir del plan de negocios es de donde comienzan los objetivos tanto del plan de desarrollo, trabajo, recursos y mantenimiento.

Expectativas de la consultoría. No puedo decir que el trabajo de la empresa consultora haya sido malo, sino que tal vez esperábamos demasiado. Es necesario reconocer que un consultor no es responsable de nuestro trabajo, pero sí es responsable de guiarnos y en esa línea, nosotros teníamos que presionar las revisiones de la consultoría y no esperar a que ellos nos solicitaran algo. Claro que también la consultoría tiene cierta responsabilidad ya que ante cualquier proyecto se tiene que comenzar con un estudio del cliente para saber qué terreno se está pisando y en este caso al reconocer que todos éramos totalmente neófitos en la materia, era necesaria una asesoría con mayor interacción entre los responsables de cada proceso.

Considerar la herramienta. A pesar de que la norma no especifica el uso de cierto producto o en particular, sí especifica ciertos requerimientos a cubrir que permitan tanto almacenar los documentos de trabajo, como generar, resguardar y consultar las evidencias de que se está siguiendo el proceso establecido. Es necesario decidir desde un inicio qué mecanismo se utilizará como base de conocimiento.

Trabajo en equipo. Es necesario fomentar el ambiente de trabajo en equipo, así como tener un compromiso serio ante la asignación de roles. Al asignar a una persona cierto rol es porque se le dará la oportunidad y ofrecerá la confianza para que realice las actividades correspondientes, y en consecuencia la persona que acepta uno o más roles acepta esa responsabilidad y ofrece realizar las acciones necesarias para cumplir con esa tarea. Esta relación es vital para el buen desempeño de todos los implicados ya que si no se brindan las facilidades para la realización de las tareas, o se aceptan más tareas de las que se está dispuesto a cumplir, el perjuicio se ve reflejado entre todos los niveles de cada proceso.

Comunicación. Un último factor que es necesario generar y lograr mantener es el de una comunicación efectiva, en donde los responsables de cada proceso de forma autónoma presenten los resultados, problemáticas y acciones de corrección en reuniones breves y con los implicados respectivos. Es desgastante asistir a reuniones a tratar temas de conflictos internos o personales recurrentes en donde no se toman las medidas de corrección respectivas. Estas reuniones deben estar planeadas con un orden y bajo un programa

previamente definido sin desvíos significativos y terminando con acuerdos formales que formarán parte de los resultados consecutivos.

Conclusión

Habiendo vivido esta experiencia, considero que para lograr un buen resultado ante un proceso de certificación, independientemente de contar con una empresa consultora, es recomendable contar con algún elemento que tenga la experiencia en el proceso de implantación o que haya laborado bajo esa metodología de trabajo. Pero si no se cuenta con este elemento que proporcione su experiencia se deberá de invertir en la preparación del mismo para que esté un paso delante de la empresa consultora, prevea las implicaciones, decida las acciones a tomar, resuelva conflictos y pueda coordinar todo el proceso de implantación.

Bio:

Alfredo Lozada Carrillo (ialozadac@gmail.com) es analista y desarrollador en Apesa Software y consultor externo para Grupo AUSA. Anteriormente fue jefe de Informática en la Dirección de Materiales Educativos en la SEP, gerente de desarrollo en Sistemas CASA y consultor en Siga Desarrollos.

El Extraño caso del Líder Jekyll y Mr. Hyde

Autor:

Efraín Cordero

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Gestión de Proyectos

Esta es la historia de un líder de proyecto llamado Jekyll, el cual alegremente —al menos al comienzo— lideraba un nuevo proyecto de desarrollo de software como tantos otros que existen, han existido y existirán.

Nuestro líder Jekyll era un caso típico de trayectoria profesional en nuestra industria. Después de estudiar una carrera afín al mundo de sistemas, inició como programador becario en alguna tecnología de moda para luego desempeñar durante algunos años diferentes roles de ingeniería de software en diferentes proyectos —a veces varios de manera simultánea—. En el transcurso tomó uno que otro curso, normalmente relacionado con lenguajes de programación o herramientas de desarrollo. Finalmente, en aras de su probada capacidad técnica y organización personal, su jefe tomó la decisión de convertirlo en "líder de proyecto" (entendiendo como tal lo que el rol significara para el jefe), y para "foguearlo" lo asignó a dirigir un nuevo proyecto.

Acostumbrado a afrontar nuevos retos con optimismo, Jekyll aceptó su nuevo rol poniéndose al frente del equipo de trabajo que le fue asignado, el cual estaba compuesto principalmente por recursos de poca experiencia junto con un par de "lobos de mar". El proyecto, según le explicó su jefe, había sido ganado "con

mucho esfuerzo", para lo cual había sido necesario realizar ajustes en las tarifas y recortes en los tiempos. Al preguntar quién había calculado los tiempos, recibió como respuesta que “el equipo de ventas” (sin más detalles de quiénes o cómo). Sin embargo, para tranquilizarlo su jefe añadió que tenía la ventaja de que el levantamiento de requerimientos ya estaba hecho (lo había hecho otra consultoría), y que la arquitectura base ya estaba definida (la había definido el cliente), por lo que “únicamente” le iba a tocar dirigir el desarrollo.

De manera previsora, nuestro Jekyll solicitó a sus "lobos de mar" asignados que le echaran un vistazo a la arquitectura base. Para su sorpresa, los lobos le manifestaron que, a pesar de contar con varios años de experiencia y un largo recorrido en desarrollo de sistemas, desconocían por completo las tecnologías que la componían. Dado que el resto del equipo estaba compuesto por programadores novatos, lo más que pudo obtener fue un par de "leí algo, alguna vez, en un sitio de Internet...". Por tanto, Jekyll acudió con su jefe para solicitar que se le asignara algún recurso con experiencia en la tecnología, a lo que recibió una negativa con el argumento de falta de personal y comentarios respecto a problemas con las tarifas. Sin embargo, el jefe recordó que Jekyll había participado como desarrollador en algo parecido en el pasado, por lo que aseveró que seguramente no tendría ningún problema para implementar el proyecto. Un poco intranquilo, Jekyll hizo "de tripas corazón" y en aras del cumplimiento del proyecto aceptó la responsabilidad técnica del mismo (es decir, desempeñar también el rol de arquitecto).

Requerimientos difusos

Continuando con el proyecto, Jekyll solicitó una vez más el apoyo de los lobos de mar para dimensionar el alcance a partir del análisis de los documentos de requerimientos. Consternados, detectaron varios problemas:

A pesar de que la consultora que realizó el levantamiento de requerimientos se anunciaba como “experta en análisis de negocio”, no existía ningún documento de un análisis del negocio como tal. Los únicos documentos disponibles eran especificaciones de casos de uso. El estilo de redacción de los casos de uso variaba desde vagas descripciones hasta verdadero pseudocódigo, y en ocasiones se apreciaba que eran un mero “copy-paste”.

Muchos de los términos y procedimientos del negocio eran completamente desconocidos para el equipo de trabajo, y no existía documentación disponible al respecto.

Se hacía mención a la necesidad del cumplimiento de legislaciones y estándares, pero no había ninguna información que los explicara.

Los casos de uso no tenían ningún tipo de agrupación o secuencia, ni venía documentada su alineación con los flujos de operación del negocio.

Se indicaban decenas de validaciones y cálculos, pero en ningún lugar se describían. Las reglas de negocio no estaban documentadas en ningún otro lado.

Las relaciones entre casos de uso eran inconsistentes o incorrectas, y los actores participantes no eran homogéneos ni se describían en ninguna parte de la documentación. De hecho, tampoco casaban con roles de los empleados de la empresa.

Procesos batch y reportes venían especificados usando los mismos formatos usados para documentar casos de uso. Sin embargo, los “layouts” de los reportes no estaban indicados.

Los requerimientos no funcionales no eran explícitos, además en ocasiones chocaban con notas y comentarios desperdigados en los casos de uso.

Ante lo anterior, Jekyll solicitó el apoyo de su jefe, el cual para tranquilizarlo procedió a negociar con el cliente un periodo "extraordinario" para aclaración de dudas del equipo (no quedando claro quién absorbería el impacto en costo y calendario). Como había que designar un responsable del lado del equipo, una vez más Jekyll aceptó la responsabilidad y se constituyó en el "líder de análisis" del proyecto.

Arquitectura incongruente

Recordando Jekyll que tenía también el rol de arquitecto, para aprovechar el tiempo procedió a revisar más a fondo la arquitectura base establecida por el cliente. Resultó ser un documento largo y pomposo el cual:

A todas luces se veía que había sido generado haciendo una mezcla de documentación proveniente de otros sistemas. Incluso el nombre de los mismos aparecía en varios lugares donde se les había pasado reemplazarlos por el nombre del sistema actual.

Incluía una combinación de tecnologías sin justificar la razón de su elección.

Sin ton ni son mezclaba vistas, modelos y conceptos de diferentes metodologías y marcos de descripción de arquitecturas.

Sus diagramas adolecían del “síndrome del rompecabezas abstracto”, es decir diagramas que encajan bien y se ven bonitos pero no es claro lo que representan las piezas. Esto ocurría porque los autores no acompañaron a los diagramas con descripciones de cada una de las partes que los componían, asumiendo que las etiquetas de nombres transmitían perfectamente el objetivo y características de cada parte.

Hacía referencia a funcionalidad no incluida en las especificaciones.

En ningún lugar se explicaba como la “arquitectura base” abordaba el cumplimiento de los requerimientos no funcionales.

Al preguntar Jekyll al cliente si esa arquitectura base ya se estaba utilizando en algún sistema que estuviera en operación de la empresa, recibió la respuesta de que "no, pero hay uno que casi la utiliza". Después de realizar diversas gestiones en diferentes frentes, Jekyll consiguió que se le mandara una versión "triple I" (incompilable, incompleta e ininteligible) de la arquitectura base. Al preguntar si era posible realizarle cambios, le contestaron que no porque la había definido el comité de arquitectura de la empresa (cuyos miembros no tenían participación ni responsabilidad alguna en el proyecto, ni tampoco era posible consultarles), y porque su uso estaba establecido por contrato.

Bajo involucramiento del cliente

Ante esto, Jekyll (que a estas alturas del partido aún no había leído el contrato de servicio), solicitó a su jefe una copia del mismo y procedió a leerla con detenimiento. Al hacerlo, entre otras cosas encontró que venían cláusulas leoninas de penalización por incumplimiento por parte del equipo de desarrollo, pero no se había incluido ninguna que tocara al cliente por incumplimiento de acuerdos o por períodos de espera atribuibles al cliente. Una vez más Jekyll acudió con su jefe para exponer estos puntos. Resultó que de muchos de ellos el jefe no estaba enterado, pero intentó tranquilizarlo con la indicación de que no se preocupara, porque contaba con todo el apoyo de la empresa y porque el cliente estaba comprometido.

Francamente alarmado, Jekyll comenzó a acudir a las sesiones de "aclaración de dudas", en donde los participantes del lado del cliente con frecuencia no acudían, llegaban tarde o se iban antes (porque no podían descuidar sus actividades normales), no sabían de qué se les estaba hablando (porque la consultora anterior jamás les había presentado los casos de uso), disentían de su contenido (porque la operación “no era” como la indicaban los casos de uso), planteaban sobre la marcha cambios y "mejoras" (discutiendo arduamente entre ellos por cuestiones triviales sin llegar a ningún acuerdo), o con una sonrisa comentaban que lo contenido en los casos de uso "ya no aplicaba" (porque en el inter habían ocurrido cambios en el negocio, en un sistema externo o en alguna legislación). Al acudir por enésima vez con el jefe a plantear los problemas, recibió una palmadita en la espalda y una vez más la recomendación de que no se preocupara, porque el proyecto "iba a salir".

¿Y todavía se preguntan de dónde sale el Mr. Hyde de esta historia?

Conclusión

Para los lobos marinos que todavía se hacen a la mar, sirva el presente cuento para provocarles sonrisas al reconocer escollos y monstruos marinos atemorizantes. Para aquellos que ya no se hacen a la mar y que viven en los puertos despachando navíos, sirva para revivirles sensaciones casi olvidadas de sus épocas marineras y crearles conciencia no sólo de la crueldad de designar a marinos jóvenes como capitanes para mandarlos inmediatamente a navegar en aguas tormentosas, sino de la más que probable posibilidad de que al hacerlo se vayan a pique los barcos que comandan.

Bio:

Efraín Cordero López es responsable técnico y líder de proyecto en grupo CARSO. Es licenciado en computación por la Universidad Autónoma Metropolitana y cuenta con acreditaciones como SCJP, SCWCD y Oracle SOA Implementation Champion. Es autor de cursos de diseño, pruebas unitarias y mejores prácticas de ingeniería de software.

El Testing de la Experiencia de Usuario

Autor:

Mauricio Angulo

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Tecno-lógico

Hablar de Experiencia de Usuario, comúnmente abreviada UX por sus siglas en inglés se ha vuelto algo común en los equipos de desarrollo y diseño de software de todo tipo. En la última década, el proceso de

"consumerismo" de la tecnología —es decir, el fenómeno cultural en el que la tecnología se ha convertido en un producto dirigido a los usuarios finales—, ha forzado a las personas que crean tecnología a incorporar en su trabajo diario conceptos de diseño, usabilidad, economía y varios más relacionados con la experiencia final de sus usuarios.

Sin embargo, el tema de UX aún no ha permeado del todo al proceso completo de desarrollo de software, especialmente a las tareas relacionadas con testing. Esto en buena parte se debe a que los conceptos de Experiencia de Usuarios son difíciles de analizar y medir. ¿Cómo puede medirse cuantitativamente algo que es subjetivo por naturaleza?

Definiendo la usabilidad de un producto

La UX se trata de cómo se siente una persona sobre el uso de un sistema o tecnología. La UX resalta los aspectos vivenciales, afectivos, significativos y valiosos de la interacción humano-máquina y de propiedad de producto, así como las percepciones de una persona sobre los aspectos prácticos de la tecnología, como su utilidad, su facilidad de uso y su eficiencia.

La Experiencia de Usuario es subjetiva por naturaleza, porque describe los sentimientos y pensamientos de un individuo acerca de un sistema. La UX también es dinámica, ya que cambia con el tiempo a medida de que cambian las circunstancias.

Muchas personas piensan que la UX es una cualidad nebulosa, que no puede ser medida o cuantificada, pero sí es posible al utilizar métricas de usabilidad, que pueden aportar información muy valiosa sobre la experiencia de usuario de un producto tecnológico.

Entonces, ¿qué es usabilidad?

La Organización Internacional de Estándares (ISO) identifica tres aspectos de la Usabilidad y la define como: "la capacidad en la que un producto puede ser utilizado por un grupo específico de usuarios, para lograr objetivos de eficiencia, efectividad y satisfacción, en un contexto específico de uso."

La Asociación de Profesionales de la Usabilidad (UPA) define la usabilidad en términos de un proceso de desarrollo: "Usabilidad es un acercamiento al desarrollo de un producto que incorpora directamente la retroalimentación del usuario durante el ciclo de desarrollo para reducir costos y así crear herramientas y productos que resuelvan las necesidades de sus usuarios".

El libro "No me hagas pensar" de Steve Krug, tiene una perspectiva más sencilla: "Usabilidad significa simplemente asegurarse que algo funciona bien: que una persona con habilidades y experiencia promedio (o incluso debajo del promedio) puede usar ese algo —ya sea un sitio web, un avión de propulsión o una puerta giratoria— sin sentirse frustrado".

Las tres definiciones, así como muchas otras sobre el tema, comparten tres aspectos:

- El usuario está involucrado.
- El usuario hace algo.
- El usuario hace algo con un producto, sistema o alguna otra cosa.

Métricas de usabilidad

Una métrica es una manera de medir o evaluar un fenómeno o cosa en particular. Las métricas por definición deberían ser:

a) útiles, ya que medir algo que no es útil es una pérdida de tiempo;

b) cuantificables, es decir, el valor de la métrica debe ser capaz de ser expresado de manera numérica;

c) concretas, en el sentido de que debe medir una cosa, no varias a la vez; y

d) observables, ya que integrar métricas que no podemos medir no aporta información útil al diagnóstico.

Las métricas de usabilidad no son un fin en sí mismas, sino que aportan información útil que no puede ser obtenida de otra manera. Las métricas de usabilidad pueden responder a las siguientes preguntas:

- ¿El producto le gustará a los usuarios?
- ¿El nuevo producto es más eficiente que la versión anterior?
- ¿Cómo se compara la usabilidad de este producto con su competencia?
- ¿Cuáles son los principales problemas de usabilidad del producto?
- ¿Se están logrando mejoras en cada interacción?

Las métricas de usabilidad pueden ayudar a revelar patrones que son difíciles o imposibles de observar a simple vista, o con pruebas automatizadas. Al evaluar un producto, incluso con un grupo de usuarios pequeño, normalmente se revelan los problemas más obvios de usabilidad, sin embargo, existen muchos problemas más sutiles que requieren de la ayuda de métricas. Justo estas métricas de usabilidad pueden ser de gran ayuda para analizar un producto desde nuevos y diferentes puntos de vista, para obtener un mejor entendimiento del comportamiento de los usuarios y su relación con los productos que utilizan.

Existen muchos mitos acerca del uso de métricas de usabilidad: desde malas experiencias con ellas (o la falta de las mismas), hasta justificaciones sobre cómo, debido a que implican trabajo extra, impactan en los costos y causan tiempo adicional en las etapas de testing, ya sea debido a la complejidad para obtener grupos de usuario suficientemente grandes, mostrar productos sin terminar a personas externas al proyecto o la dificultad de procesar las fórmulas asociadas a las métricas.

El argumento contundente en contra de estos mitos es que las métricas de usabilidad son un componente clave para el éxito de un producto y para su ROI (Return of Investment). Como parte de un plan de negocios, normalmente se pide determinar cuánto dinero es ahorrado o cuánto aumentan las ganancias como resultado de la nueva versión o rediseño de un producto. Sin métricas de usabilidad que respalde un análisis, cualquier recomendación es una mera adivinanza.

Finalmente, las métricas de usabilidad sirven para mostrar si se está mejorando la UX en un producto nuevo o entre las versiones de un producto.

Midiendo la UX

La definición de las métricas de usabilidad varía de un producto a otro pero el criterio para generarlas es siempre el mismo: ¿cuál es el sentimiento del usuario cuando interactúa con el producto? La gestión de estas métricas no es algo que deba delegarse al usuario, sino que debe ser medido por el equipo de testing mientras se realizan pruebas con usuarios finales con preguntas y parámetros que puedan ser comprendidos plenamente

por los usuarios. El objetivo de estas métricas siempre es lograr una mejor experiencia de usuario en los productos que creamos, no justificar decisiones de producción.

Actualmente, muchas metodologías alrededor de la creación de producto de software —incluyendo las de testing— están enfocadas en lograr ciertas calificaciones dirigidas únicamente a las características del producto: que funcione bien, que sea rápido, que sea barato, que sea fácil actualizarlo o que salga a tiempo.

La perspectiva del Diseño Centrado en el Usuario cambia el enfoque hacia las personas que utilizarán el producto, y en el mercado actual tener un buen producto ya no es suficiente para tener éxito, es necesario crear productos que los usuarios quieran y deseen utilizar, porque es fácil y placentero usarlos.

Bio:

Mauricio Angulo (@mauricioangulo) es programador desde 1989 divulgador, ávido escritor y emprendedor. Actualmente es CEO y fundador de Tesseract Space donde realiza funciones de asesor y consultor de innovación tecnológica, mercadotecnia digital y experiencia de usuario.

Programación en la Escuela ¿Para qué?

Autor:

Gunnar Wolf

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Programar es un Estilo de Vida

En el número de agosto 2012 de SG, Ignacio Cabral Perdomo presentó un interesante artículo titulado «Enseñando a niños a programar: ¿Imposible o una oportunidad?». La respuesta me parece clarísima: Claro que se puede. Esto viene siendo demostrado con gran éxito, desde los 1960s, empleando el lenguaje BASIC diseñado por Kemeny y Kurtz, y muy particularmente con el lenguaje Logo, conocido principalmente gracias al trabajo de uno de sus autores, Seymour Papert. En esta columna haré un planteamiento diferente.

El énfasis que presentan las conclusiones de Ignacio apunta al mercado del desarrollo de cómputo: “Es muy clara la necesidad de más profesionistas en el área de la Computación y las Tecnologías de Información, en especial en los departamentos de desarrollo de software de diferentes corporaciones pero, desgraciadamente, el interés de los alumnos por carreras de este tipo está reduciéndose de una forma alarmante. Una posible solución es el inculcar desde temprana edad el pensamiento lógico y algorítmico en los niños siguiendo el itinerario de aprendizaje que propongo”.

Si bien el artículo refiere que la enseñanza de programación a partir de nivel primaria “ayuda a los chicos a formar ese pensamiento lógico y algorítmico que tanto necesitan”, no profundiza en este aspecto, que considero fundamental. ¿Por qué los chicos pueden necesitar un pensamiento lógico y algorítmico?

Los hijos de Logo

Soy parte de una minoría afortunada (pido disculpas anticipadas si el presente artículo se ve como un viaje a mi anecdotario personal), aprendí computación cuando el acceso al equipo de cómputo era extremadamente poco común. Mi primera experiencia fue en la minicomputadora Foonly que había en el IIMAS (UNAM) en 1983, a los 7 años, escribiendo LaTeX con el editor Emacs. Cabe mencionar que el presente artículo, casi 30 años más tarde, lo estoy escribiendo con las mismas herramientas. Tuve acceso a la Foonly gracias a que mi padre trabajaba como investigador en dicho Instituto, y a que tuvo la paciencia de enseñar a su ávido niño ese lenguaje cargado de símbolos y comandos.

Pero creo que mi experiencia con la Foonly se habría mantenido como meramente incidental de no ser porque, uno o dos años más tarde, me inscribieron en IDESE, una de las primeras escuelas de verano dedicadas al cómputo en México. IDESE era una apuesta pedagógica muy interesante que consistía en que durante tres semanas alternábamos dos horas frente a la computadora con dos horas más con juegos de mesa. Si bien no recuerdo los detalles de la interacción, esta alternancia ilustra claramente cómo veían nuestros instructores su tarea: Llevar a los niños a emplear sus habilidades cognitivas de una manera más completa.

IDESE derivó de la versión de Logo desarrollado por el MIT para la Apple, traduciendo todos sus comandos y mensajes al español. Sólo otra vez, también en los 1980, vi un esfuerzo similar: El hecho por la BBC al traducir el lenguaje BASIC de su BBC Micro para crear el EBASIC. Esto permitía enseñar a los niños a programar la computadora sin preocuparse al mismo tiempo de aprender otro idioma. El caso del EBASIC me resulta notorio porque, con un comando, se podía ver el código escrito en EBASIC en BASIC "normal". Para 1985, me tocó formar parte del taller de computación que se impartía en mi escuela a partir de 4° de primaria. A partir de 1986, estuve inscrito para varios cursos de los Centros Galileo. Tuve la suerte de haber pasado por escuelas muy motivadoras, con lo cual a esas tempranas alturas ya estaba descubierta mi vocación.

El gran acierto de Logo que lo hizo tan importante como lenguaje educativo fue eliminar las capas de abstracción que debía tener en mente un niño. Si bien el lenguaje permite un desarrollo complejo y formal de programación funcional, el niño podía ver la concreción de sus programas graficándolos a través de una “tortuga”. Permitir que el niño viera directa e inmediatamente sus resultados, hace 45 años, resultó un cambio fundamental y un gran motivador.

Cuando Logo fue planteado, no existía el plan de formar a los niños en programación por la gran demanda que dichas habilidades tendrían en la sociedad. La enseñanza de programación era vista como una forma de enseñar pensamiento abstracto y algorítmico.

¿Y para qué enseñar pensamiento abstracto y algorítmico si no para formar profesionales que comprendan más fácil los paradigmas de cómputo? Citando a un buen amigo: “de lo que se trata no es de aprender más que a programar, aprender lo que significa programar”. Dicho de otro modo, ¿Para qué se enseñan matemáticas, filosofía, historia o biología? Para formar personas más completas, no sólo en su cultura, sino que en la

manera de estructurar el pensamiento. Habilidades que indudablemente impactan en su crecimiento como adultos.

La OLPC

Ninguna herramienta dura para siempre, sin embargo, y ni siquiera el gran Logo se salva de la obsolescencia que las nuevas tecnologías van causando. Hoy en día, sería iluso pensar que mover una "tortuga" por la pantalla pudiera impresionar a un niño. Afortunadamente, no han sido pocos los estudios en este campo que se han realizado — El artículo de Ignacio presentó cuatro entornos de programación orientados a la enseñanza en diferentes edades: Scratch, Alice, Greenfoot y BlueJ.

Agrego a los anteriores uno de los trabajos más comentados de los últimos años, que tiene un impacto muy medible: El proyecto OLPC (One Laptop Per Child)[1], iniciado –al igual que Logo– en el MIT y con el decidido apoyo de Seymour Papert, entre otras muchas personalidades.

La OLPC no es cualquier computadora: Fue planteada como el vehículo sobre del cual correría Sugar[2]. Yo no tengo experiencia de primera mano con el entorno, por lo cual prefiero dirigir a quienes estén interesados en una descripción más completa al artículo que publicó Werner Westermann dentro del libro «Construcción Colaborativa del Conocimiento»[3].

En resumen, Sugar es un entorno dirigido a facilitar un aprendizaje construccionista, en que cada alumno debe ir explorando y construyendo su camino por medio de la experiencia personal, lo cual lleva a una mayor apropiación del contenido que cuando éste es dictado. A partir de una interfaz sencilla, una orientación más a actividades que a aplicaciones y empleando a fondo la colaboración entre todos los alumnos, la computadora se vuelve un actor, un facilitador de la transmisión del conocimiento. Y una característica fundamental de Sugar es que el alumno no sólo puede utilizar las actividades, sino que puede modificarlas. Las actividades están escritas en Python, un lenguaje de sintaxis limpia y conceptualmente fácil de adoptar.

OLPC fue planteado como un proyecto necesariamente a gran escala: Su objetivo es que una computadora sea entregada a cada niño en edad escolar en los países receptores. El proyecto busca además resolver problemáticas específicas de los países menos favorecidos; con ciertas modificaciones al planteamiento inicial, actualmente hay despliegues de OLPC en once países de escasos recursos[4].

Y siguiendo con el tono personal que he mantenido en esta ocasión, relato lo que me contó Manuel Kauffman, un desarrollador argentino de Sugar, en una visita que hizo a una escuela en Uruguay: Un niño, de 11 o 12 años, le explicó que prefería editar en texto los iconos de las actividades que iba creando o modificando directamente en un editor de texto, en SVG (un lenguaje basado en XML para representar gráficos vectoriales) porque le quedaban más limpios que utilizando un editor gráfico.
Este ejemplo habla como pocas cosas de apropiación de la herramienta y de la apreciación estética de la representación en código de un objeto. Hay programadores de larga carrera profesional que son incapaces de desarrollar estas habilidades.

Conclusiones

Enseñar a programar a un niño va mucho más allá de transmitirle una habilidad para su futuro profesional. La enseñanza básica no puede basarse solamente en la transmisión de competencias medibles en el mercado.

Hay, sin embargo, puntos importantes a considerar. Si bien algunos tuvimos la gran suerte de aprender de la forma y en el momento correcto, es una materia con la que muchos se enfrentan con dificultad, el desarrollo de las capacidades de abstracción necesarias para esta materia se produce de forma muy desigual, y la frustración que esta materia puede causar en algunos alumnos puede ser muy grande. Cabe mencionar, claro, que esto mismo se presenta en varias otras materias que forman ya parte del currículo básico.

Otro punto importante a considerar es la formación de los docentes. Para incorporar una materia al currículo básico, es necesario contar con un cuerpo docente suficientemente amplio y capacitado, no sólo con las habilidades técnicas sino pedagógicas.

Referencias

[1] One Laptop per Child. http://one.laptop.org
[2] Sugar. http://sugarlabs.org
[3] Autores varios. Seminario Construcción Colaborativa del Conocimiento.http://seminario.edusol.info/seco3
[4] LPC Stories. http://one.laptop.org/stories

Bio:

Gunnar Wolf es administrador de sistemas para el Instituto de Investigaciones Económicas de la UNAM y desarrollador del proyecto DebianGNU/Linux. http://gwolf.org

De la Luna a Marte sin Cambiar de Equipo de Trabajo

Autor:

Francisco Estrada Salinas

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Carrera

Hemos vivido engañados: en la Universidad se nos enseñó que al hacer más eficientes las partes, el “todo” también se hace más eficiente. Sin embargo, considero que si seguimos dicha receta al pie de la letra, estamos precipitando el “todo” al fracaso.

Como toda área de negocio, las organizaciones de TI son pirámides cuya base está conformada por personas: el Recurso Humano. Optimizar este recurso, de inmediato propagará un efecto positivo en el resto de la pirámide. Optimizar lo demás sin atender el recurso humano, definitivamente tendrá poco o nulo efecto en lo general.

Las personas encargadas de dirigir una organización de TI típicamente se encuentran con un grupo de personas con una gran variedad de personalidades y situaciones: jóvenes con nuevas ideas, veteranos aburridos, candidatos que no lograron obtener la posición que ahora ocupas, personas marginadas, etcétera. El reto que tiene en sus manos el líder es hacer de este equipo, un equipo de cinco estrellas, óptimo y eficiente. ¿Suena familiar?

Los 2 caminos básicos son los siguientes:

- **Eliminar por bajo rendimiento**. Consiste en deshacerse de los menos destacados y contratar gente nueva, usualmente joven, lista para aprender y con los ánimos dispuestos para dedicar mucho tiempo a colaborar y mejorar lo que se les ponga en frente. No se requiere esfuerzo del Manager, solamente paciencia y mucha gestión-supervisión de la curva de aprendizaje. La posibilidad de éxito dependerá del compromiso de los nuevos integrantes.

- **Mejora de los recursos humanos existentes**. Este camino involucra un gran reto, ya que hay que hacer reaccionar a los integrantes actuales del equipo, compartir la idea de que tu gestión es nueva y busca integrarlos a la organización y a los proyectos de la compañía.

  El despido de elementos "malos" ha resultado popular los últimos años y se ve como la oportunidad de infundir sangre nueva a la empresa. Pero el verdadero liderazgo reside en hacer crecer a los elementos actuales: el líder debe ser capaz de crear el equipo de trabajo necesario, con los recursos que ya cuenta.

  Aunque en algunos casos pueda parecer imposible, a continuación destaco puntos particulares que pueden ayudar al éxito:

  Identificar a los elementos potenciales y enfrentarlos consigo mismos. Siempre hay personal con fuerte potencial: gente con ideas y avanzado nivel educativo que se encuentra aletargada, aburrida y sin expectativa. Enfréntalos con el proyecto más complicado, el de mayor exposición, mostrando las consecuencias positivas de su trabajo que revivirán su carrera profesional.
  Análisis forense: ir de la consecuencia a la acción. Me he topado en repetidas ocasiones con gente que no tiene ni idea de para qué sirve lo que está haciendo: se les piden tareas para proyectos nebulosos cuya información completa solo la tienen "los de arriba". Relaciona a cada persona con cada proyecto o tarea, desde el inicio hasta el final. Vende siempre la idea de que cada quien es pieza crucial en lo que le compete, generando beneficios propios y para el equipo completo.
  Asignar actividades y proyectos de reto y exposición. Nada como defender ante una audiencia las actividades realizadas, los proyectos, logros y alcances. Cada quien debe hacerlo con los suyos, asigna a cada persona de acuerdo a sus proyectos y tareas la obligación de retarse y defenderse en foros –de manera sana y dirigida al convencimiento- que pongan a prueba sus capacidades de comunicación.
  Exigir a los más instruidos: maestros, doctores y licenciados. Pero no se trata de exigir por exigir. Realmente las personas con títulos tienen altas expectativas de sí mismas y cualquier reto les parecerá merecido y necesario. Si alineamos ese ánimo con los planes de la organización, tendremos líderes a cargo de proyectos o piezas importantes para la organización.

  Todo lo anterior motiva directamente el sentido de pertenencia, de liderazgo, de utilidad, profesionalismo y ambiente meritorio que estamos buscando incentivar en nuestros grupos de trabajo y compañías en general.

  No siempre es dinero, no siempre es gente nueva. Se trata simplemente de escuchar a las personas y colocarlas en los proyectos correctos, reconocer su trabajo al permitirles exponerlo y desarrollarlo,

manteniendo siempre “in crescendo” el reto al conocimiento y las capacidades profesionales, cognoscitivas y de liderazgo.

Bio:

Francisco Estrada Salinas (estrada_fj@prodigy.net.mx) es Manager de infraestructura tecnológica y tiene más de 20 años creando equipos de trabajo y tecnologías para Bancos, Casas de Bolsa, Aseguradoras. Es experto en tecnologías de trading, sistemas de misión crítica, y continuidad del negocio.

OpenStack

Autor:

Por Pedro Galván y Juan Cáceres

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Infraestructura

OpenStack es una colección de proyectos de software open source que te permite establecer y administrar tu propia infraestructura de cómputo como servicio (IaaS), la cual es administrable de manera programática por medio de APIs. En otras palabras, cloud computing open source desde tu data center.

El proyecto nació en julio de 2010, fundado por la NASA y Rackspace, un proveedor de servicios de hosting. Desde entonces, OpenStack se ha ido ganando el respaldo de varias de las empresas más importantes de la industria, entre ellas IBM, Red Hat, HP, Cisco y Dell.

OpenStack es la base de OpenCloud, una iniciativa que busca la interacción entre nubes de forma estándar, evitando así quedar “atado” a un proveedor específico.
La versión más reciente de OpenStack al escribirse este artículo lleva el nombre “Folsom” y fue liberada en septiembre del 2012.

Arquitectura

OpenStack no es un solo proyecto, sino un conjunto de proyectos, cada uno orientado a desarrollar un componente de la plataforma. La figura 1 ilustra los diferentes componentes de OpenStack y como interactúan. A continuación describo cada uno.

Nova. Es el componente que se encarga de administrar los recursos de cómputo. Es lo que en el argot de cómputo distribuido se conoce como un fabric controller. Nova maneja todas las tareas requeridas para soportar el ciclo de vida de las instancias que se ejecutan en OpenStack. Nova expone sus capacidades por medio de un API compatible con el API de Amazon EC2.

Cinder. Es el servicio de almacenamiento por volumen o bloque (block storage). Permite que las máquinas virtuales puedan acceder e interactuar con dispositivos de almacenamiento virtuales por medio de una interfaz iSCSI. Brinda una funcionalidad comparable a la que tendrías utilizando un sistema SAN, con sus ventajas (alto desempeño para lectura/escritura) y limitaciones (un dispositivo solamente puede ser usado por una sola máquina virtual al mismo tiempo). Sin embargo, a diferencia de un SAN convencional, el API de Cinder te permite que de manera programática puedas asignar un dispositivo a una u otra máquina virtual conforme lo necesites.

Swift. Es un almacén de objetos distribuido que viene a cumplir una función análoga al Simple Storage Service (S3) de Amazon Web Services. Dada su naturaleza, Swift es altamente escalable tanto en términos de tamaño (varios petabytes) como capacidad (billones de objetos) y cuenta con capacidad nativa de redundancia y tolerancia a fallas.

Quantum. Es el subsistema para gestión de conectividad (network connectivity-as-a-service). Quantum está basado en OpenFlow [3] y su tecnología permite a las redes tomar decisiones de ruteo en base a aplicaciones, es decir las redes ya están empezando a entender el software y los servicios de negocio, y tomando las decisiones en base a esto.

Glance. Provee servicios para descubrir, registrar y activar imágenes de servidores. La habilidad de copiar o sacar un snapshot de una imagen de un servidor e inmediatamente almacenarla es una capacidad importante de OpenStack. Las imágenes almacenadas se pueden utilizar como plantillas para levantar nuevos servidores rápidamente. También se pueden utilizar para almacenar y catalogar respaldos.
Horizon. Es una interfaz web gráfica para administrar un ambiente OpenStack. Se puede utilizar para administrar instancias e imágenes de servidores, crear llaves de acceso, asignar volúmenes de almacenamiento, manipular contenedores Swift, etcétera.
Keystone. Provee servicios de identidad y política de acceso para los componentes de OpenStack. Disponible

de manera programática por medio de un API tipo REST.

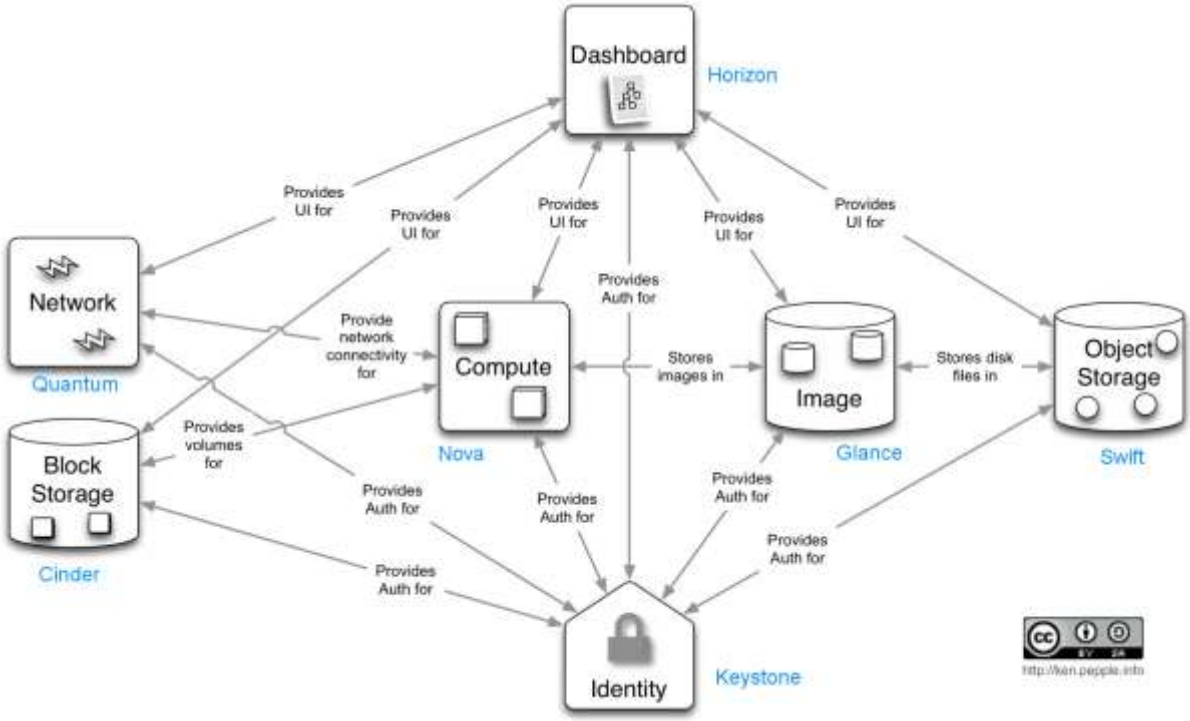


Figura 1. Arquitectura de OpenStack. Crédito: http://ken.pepple.info

Conclusión

Aunque OpenStack es una plataforma joven, ha cobrado gran interés, y su comunidad cuenta ya con más de 6,000 colaboradores y 800 empresas en 87 países. En el ámbito empresarial, cuenta ya con instalaciones importantes. De hecho mercadolibre.com está sobre OpenStack, así como el servicio Webex de Cisco. Proveedores como Red Hat, IBM y HP ya ofrecen también su versión de OpenStack con componentes y servicios de valor agregado. Lo mejor de todo, es que OpenStack puede correr en hardware estándar, no necesitas una marca o proveedor específico.

En México existe una naciente comunidad de OpenStack, acércate en @openstackmexico yhttp://facebook.com/OpenStackMexico.

Referencias

[1] "OpenStack Software". http://www.openstack.org/software
[2] "OpenStack 101", Piston Cloud Computing. http://bit.ly/openstack101
[3] "Open Cloud Manifesto". http://www.opencloudmanifesto.org

Bio:

Pedro Galván (@pedrogk) es cofundador y editor ejecutivo de Software Guru. De vez en cuando le gusta investigar sobre temas que no conoce y arreglárselas para escribir un artículo que parezca coherente.

Juan Manuel Cáceres (juan.caceres@jcgr.net) es director de JC Global Resources (www.jcgr.net), empresa proveedora de servicios de TI tales como run book automation, cloud management y software testing.

El Gigante Invisible

Autor:

Pedro Soldado

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Columna invitada

En la industria de las nuevas tecnologías, la innovación se produce a un ritmo extremadamente veloz. No hay año en el que no se anuncie un "nuevo producto", un descubrimiento radical que transformará la manera de hacer negocios. Paralelamente al nacimiento de tecnologías, unas más efímeras y otras verdaderamente innovadoras como SOA, la virtualización o el Cloud Computing, la idea de que una práctica que ha sido parte de la industria de TI durante décadas, resurja ahora como una de las áreas de crecimiento más importante puede resultar inusual. Se trata de software de testing, cuyo mercado no ha pasado desapercibido entre los ejecutivos y los inversores. Durante el pasado año y medio hemos visto cómo esta práctica se volvía a hacer espacio entre las áreas de crecimiento de la industria, cuando unos 10 años atrás no era reconocida ni como una propia disciplina en sí misma.

Los datos de crecimiento y valor de las herramientas de pruebas ($2 mil millones de dólares) entre otros, ilustran la idea de que el testing es un área clave de crecimiento dentro de la industria de TI, además de una preocupación principal para las empresas.

Y no hay ninguna razón por la cual no deba de ser así. Se estima que el proceso de prueba consume entre 1/3 y 1/2 de todo el presupuesto de desarrollo de software, por lo que es absolutamente crucial que estos esfuerzos económicos proporcionen un resultado exitoso.

Lo más importante de todo, sin embargo, es la extensión en el ámbito de aplicaciones de estas herramientas: desde una simple Web 2.0 flash hasta el sistema de una gran empresa. El nivel de exigencia por parte de los usuarios se ha elevado y se demandan aplicaciones más sofisticadas que ayuden a garantizar el mínimo error en los sistemas.

Con las organizaciones de todos los tamaños, moviendo todos sus sistemas críticos de negocio a la nube, los

profesionales de TI están ante un gran reto. Más que nunca, la adopción de prácticas ágiles, que permitan realizar pruebas a lo largo de todo el desarrollo, es indispensable.

En la lucha por superar el malestar económico en la mayoría de las empresas proveedoras de herramientas de TI, el testing se sitúa como una de las pocas áreas que va a contracorriente y que ofrece unas perspectivas de fuerte crecimiento. Todo esto sugiere el gran potencial del sector de software de testing y confirma su importancia en todos los campos, por lo que legítimamente se le puede llamar "El Gigante Invisible".

Bio:

Pedro Soldado es Director General de Micro Focus Iberoamérica. Micro Focus es una empresa perteneciente al FTSE 250 que provee software para modernización, pruebas y gestión de aplicaciones empresariales. www.microfocus.es.

Gadgets - Noviembre 2012

Publicado en :

SG #38 (Noviembre 2012 - Febrero 2013)

Sección:

Gadgets

Drobo mini

Sistema RAID portátil y amigable

Posiblemente ya conozcas los productos de Drobo, pero en caso de no ser así ahí va una explicación rápida: son appliances de hardware para arreglos de almacenamiento (storage arrays); una especie de caja con ranuras en las que insertas discos duros de cualquier marca y capacidad, y el Drobo automáticamente se encarga de administrarlos para exponerlos como un sistema de almacenamiento RAID sencillo de usar.

La novedad es que Drobo recientemente lanzó el modelo mini, basado en discos de 2.5 pulgadas, lo cual le permite ser muy portátil. El Drobo mini cuenta con 4 bahías para discos duros —tanto magnéticos como SSD—, puertos Thunderbolt y USB 3.0. Utilizando discos de 1 TB, esto se traduciría en alrededor de 3 TB ya bajo un esquema RAID. Para aumentar la capacidad, simplemente reemplazas uno de los discos por otro de mayor capacidad y el Drobo automáticamente se encarga de incorporarlo al arreglo.

El Drobo mini tiene un precio de lista de $843 dólares y está disponible en México por medio de distribuidores autorizados.

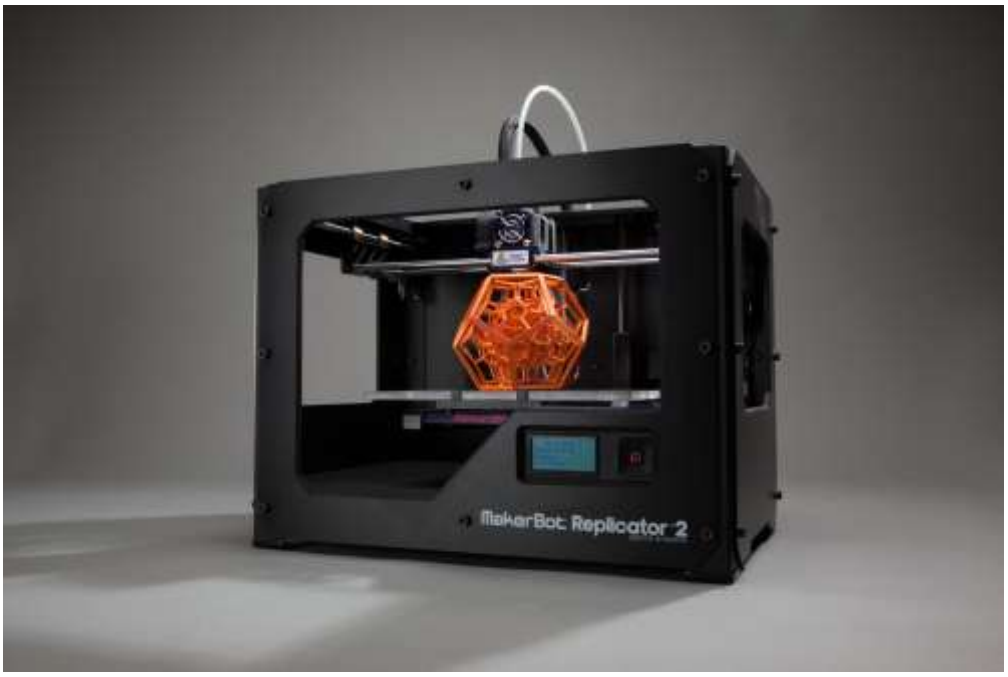
Replicator 2

Impresora 3D para escritorio

Hasta hace poco, la noción de tener en tu escritorio una impresora 3D con una calidad decente estaba fuera de la realidad. Pero actualmente esto ya es una posibilidad, tanto en cuestión de precio, como tamaño y calidad, gracias a la Replicator 2 de Makerbot.

La Replicator 2 es una impresora 3D dirigida a consumidores profesionales (prosumer) que viene a reemplazar la edición anterior de la Replicator. Entre las principales mejoras respecto a la edición anterior está una resolución de 100 micrones (el grosor de una hoja de papel) y la capacidad de imprimir objetos más grandes (hasta 28.5 cm x 15.5 cm x 15.2 cm). Para imprimir, la Replicator 2 utiliza un material llamado PLA, que es un plástico a base de maíz que no se expande ni contrae con cambios de temperatura.

Puedes comprar la Replicator 2 vía Internet en la tienda de Makerbot por un precio de $2,199 dólares. El envío a México cuesta alrededor de $145 dólares, aunque no sabemos qué pueda suceder en la aduana. El carrete de hilo PLA de 1 kg cuesta 48 dólares.



Hardware para Windows 8

¿Cual es para tí?

Los principales fabricantes de computadoras están aprovechando la llegada de Windows 8 para lanzar nuevos modelos de ultrabooks, tablets y all-in-ones. Entre los que más nos llaman la atención en SG están:

- Microsoft Surface, con su tapa-teclado y disponible tanto en versión Intel como ARM.
- Sony Vaio Tap 20, una “tablet” de 20 pulgadas con 10 puntos de contacto, principalmente pensada para el hogar pero también serviría en una sala de juntas.
- Asus Zenbook Prime, una ultrabook elegante y poderosa, a nuestros ojos lo más comparable a una Macbook para Windows.
- Asus Transformer AiO, la all-in-one de 18 pulgadas que al quitarla de su dock se convierte en una tablet con Android.

Seguramente durante los próximos meses continuaremos viendo innovación en el segmento de hardware para Windows.


I LOVE
MOM