Continuous Integration

Contenido

Principios del Manifiesto Ágil

A continuación se mencionan dos de los principios ágiles que se deben tener en cuenta para entender porqué la integración continua es importante al hablar de metodologías de desarrollo ágiles.

- ***Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.***
- ***Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.***

Concepto

¿Qué es integrar?

Podemos decir que integrar consiste colocar todo el código de la aplicación junto en un solo lugar previo a llevar a cabo un build.

¿Qué implica construir?

Construcción (Build): una construcción implica algo más que compilar, podría consistir en compilar, ejecutar pruebas, usar herramientas de análisis de código, desplegar... entre otras cosas. Un build puede ser entendido como el proceso de convertir el código fuente en software que funcione.

Entonces, ¿Qué es la Integración Continua?

La Integración Continua (Continuous Integration - CI) es una práctica de desarrollo de SW donde los miembros de un equipo integran su trabajo frecuentemente (por lo menos una vez al día al final del día), lo que lleva a múltiples integraciones por día.

El término se comenzó al utilizar desarrollar software utilizando como proceso de desarrollo al conocido como Extreme Programming (programación extrema) el cual hace mención a la integración continua dentro de sus doce prácticas (http://en.wikipedia.org/wiki/Extreme_programming_practices).

Cada integración es verificada por un build que puede ser llevado a cabo de manera automática o manual (diferencia que veremos más adelante siendo cada uno más o menos adecuado dependiendo del contexto) pero nunca dejando de lado el hecho que debe incluir la ejecución de los tests con el objetivo de **detectar errores de integración lo más rápido posible**.

Muchas veces esta práctica suele ser difícil de implementar por la resistencia de los miembros del equipo de trabajo. Sin embargo, una vez que la misma se va incorporando como una parte propia de las tareas de los miembros del equipo, hace que resulte mucho más fácil de lo que la teoría dice.

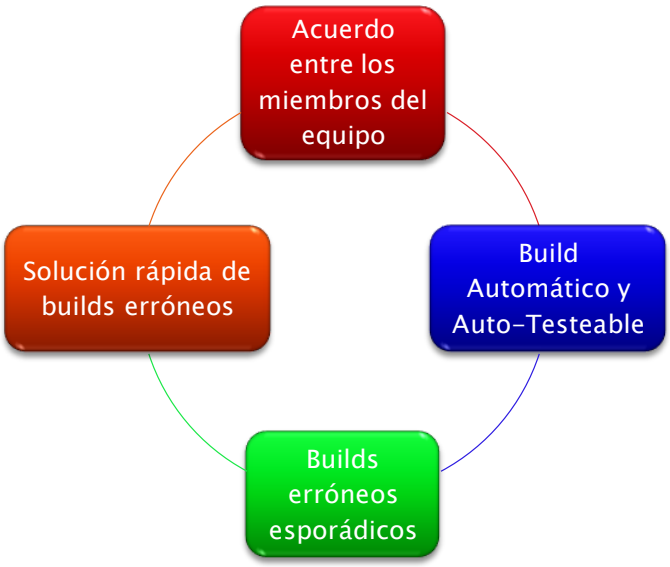
Es importante tener en cuenta que para que la integración continua pueda ser implementada de manera exitosa es necesario que todos los miembros del equipo estén de acuerdo en trabajar en el desarrollo de SW llevando a cabo esta práctica.

Cuando se logra la implementación de esta práctica en un ambiente de trabajo estamos en condiciones de cumplir otro objetivo íntegramente relacionado con el desarrollo ágil: **estar listos para entregar SW que funcione**.

Para poder lograr esto es necesario que removamos uno de los principales problemas a la hora de dar por finalizado un desarrollo: Las demoras existentes entre que se “termina” el desarrollo y la aplicación realmente puede liberarse. Para ello debemos actuar sobre las causas de este problema, como realizar el merge de todas las partes, crear el instalador, cargar la base de datos, etc.

Factores Críticos de Éxito

A continuación se mencionan los **Factores Críticos de Éxito** para poder implementar esta práctica:



- Acuerdo entre los miembros del equipo: los miembros del equipo deben asentir el trabajar realizando commits frecuentemente y que cada vez que los realizan el build no debe romperse. Si el build se rompe, deben estar conscientes que deben arreglarlo.

  Acuerdo: De ahora en adelante nuestro código del repositorio va a compilar exitosamente y va a pasar las pruebas.

- Build Automático y Auto-Testeable: los builds que se llevan a cabo después de cada integración de código deben ejecutarse automáticamente y deben incluir pruebas, a fin de detectar y resolver errores rápidamente.
- Builds erróneos esporádicos: los builds con errores todavía van a seguir existiendo (porque todos somos seres humanos y cometemos errores), pero van a ser poco frecuentes y cada vez la cantidad de builds fallados será menor.
- Solución rápida de builds erróneos: debido a que los commits se realizan más frecuentemente y por lo tanto también las integraciones y los builds, cuando hay errores se pueden resolver más rápidamente porque se encuentran más acotados.

Prácticas de Integración Continua

Mantener un único repositorio de fuentes.

Se debe hacer foco en mantener la información del proyecto, sea cual fuere, dentro de un único repositorio común.

Para la práctica de integración continua lo que interesa principalmente es que el código fuente se encuentre en el repositorio y sea actualizado constantemente. Sin embargo, tengamos en cuenta que esta es una práctica que puede llevarse a cabo en cualquier proceso de desarrollo sea cual fuere, el cual podría involucrar la creación y uso de más elementos de trabajo. Además, en la actualidad existen muchas herramientas que permiten gestionar el repositorio.

Hacer el Build Auto-Testeable

Tradicionalmente existe un concepto en donde la tarea de "construir el build" consiste en compilar, linkear, agregar librerías, y todo aquello que sea necesario para que un programa se pueda ejecutar sin inconvenientes. Cuando esto se logra no quiere decir que el SW es correcto, solamente significa que está funcionando, pero hay una diferencia entre ***"funcionar"*** y ***"funcionar correctamente"*** (es decir, tener un build realmente exitoso). Para poder decir que tenemos un build exitoso es necesario que ejecutemos pruebas. Una buena manera para detectar errores rápida y eficientemente es incluir pruebas automatizadas en el proceso de construcción del build. Como todos sabemos, el testing no es perfecto, pero puede detectar una gran cantidad de errores (bugs), los cuales serán posteriormente corregidos antes de tener el build exitoso que tanto deseamos y que cumple con uno de los lineamientos de trabajar ágilmente.

El aumento en la utilización de metodologías ágiles para el desarrollo de SW (como Programación Extrema (XP) y Test Driven Development (TDD)) ha dado un gran impulso a esta tarea de incorporar pruebas en la construcción de los build, pero no quiere decir que tengamos que practicar alguna de ellas (o ambas) para poder lograrlo.

Una de las herramientas más utilizadas al trabajar con la creación de pruebas automáticas son las herramientas **XUnit**. **XUnit** es el nombre dado a la familia de frameworks de pruebas que se han vuelto ampliamente conocidos entre los desarrolladores.

Automatizar la construcción del Build

El objetivo principal es ***simplificar y agilizar*** la tarea de construcción de los builds.

Debido a que a menudo las integraciones para llevar a cabo un build de manera manual pueden ser tediosas, complicadas y además insumir demasiado tiempo, sobre todo en proyectos de gran escala, es recomendable contar con entornos automatizados para llevar a cabo las compilaciones y que estas sean lanzadas con la ejecución de un simple script.

A su vez, debemos tratar de que TODOS los aspectos referentes a la construcción de un build estén incluídos dentro de esta "automatización" (scripts de bases de datos, poblamiento de tablas, pruebas unitarias). De esta manera cualquiera debería de ser capaz de descargar el código fuente del repositorio, ejecutar un comando (referenciando al script) y tener el sistema andando.

Un punto importante a tener en cuenta es que deberíamos de llevar a cabo el armado de los builds en un ambiente separado y con las mismas características del ambiente de destino final de la aplicación. Esto suele no ser posible por lo que puede ser una buena alternativa la utilización de máquinas virtuales.

La automatización del build es sumamente importante para poder trabajar con integración asíncrona.

Commits diarios a la línea base

La integración ronda principalmente alrededor de la comunicación. La integración permite a los desarrolladores decir a otros desarrolladores acerca de los cambios que han hecho. La comunicación frecuente permite a la gente a conocer más rápidamente los cambios que se desarrollan.

El pre-requisito para que un desarrollador realice un commit a la línea principal (Base line – Línea Base) es que pudo construir correctamente **su** código lo que incluye, por supuesto, superar las pruebas de compilación.

Al hacer esto con frecuencia, los desarrolladores rápidamente se dan cuenta si hay un conflicto. Esta es la clave para solucionar rápidamente los problemas: encontrarlos rápidamente. Si los desarrolladores realizan commits seguido entonces los conflictos o problemas pueden ser detectados rápidamente y son más fáciles de resolver ya que no han ocurrido demasiados cambios. Aquellos conflictos que se quedan sin ser detectados durante semanas son muy difíciles de resolver.

Cada commit implica un build en la línea base en una máquina de Integración

Al hacer commits diaria y frecuentemente se obtienen builds frecuentes y exitosos, los cuales deberían estar testeados y pasados ya que el build debe ser auto testeable (Ver Práctica Hacer el Build Auto-Testeable).

En la práctica, para poder lograr esto, es imprescindible que quien quiere llevar a cabo un commit actualice localmente el código de la aplicación y recompile.

Luego, la compilación debe (idealmente) llevarse a cabo en una máquina destinada únicamente a fines de integración. El commit llevado a cabo sólo se considerará exitoso si el build (con sus tests) ha sido exitoso en este ambiente. Como hemos dicho, el desarrollador que está llevando a cabo el commit es el responsable de que el build sea exitoso en ese momento y por lo tanto es el responsable del mismo y debe supervisarlo y resolver cualquier conflicto para asegurar su éxito y estabilidad.

Hay dos formas principales para asegurar el build:

- Hacer un build manual o
- Contar con un servidor de integración continua.

El Build Manual es el más simple de describir. Esencialmente es similar a la construcción que un desarrollador hace en su ambiente local antes del commit al repositorio. El desarrollador va a la máquina de integración, luego de haber commiteado su código, comprueba la cabeza de la línea principal, actualiza el código en dicha PC (realiza un check out) y comienza a realizar el build. En la integración manual el desarrollador debe estar atento al progreso y si la compilación (build) es exitosa, entonces su tarea está completa y se considera como “DONE” la integración.

Un servidor de integración continua actúa como un monitor del repositorio. Cada vez que se finaliza un commit al repositorio, el servidor realiza automáticamente un check out en la PC de integración, inicia la construcción (build) y notifica a quien realizó el commit el resultado. Quien hizo el commit NO PUEDE CONSIDERAR COMO DONE esta tarea hasta que no recibe la notificación correspondiente - por lo general un correo electrónico.

Más adelante se profundizan las ventajas y desventajas de cada uno de ellos. (Ver sección Builds Manuales y Servidores de Integración Continua)

Como hemos dicho, el objetivo de la integración continua es encontrar problemas tan pronto como sea posible y para lograr esto es fundamental que si el build falla sea identificado o notificado (dependiendo de la técnica de integración utilizada) y corregido de inmediato. El punto de trabajar con integración continua es que siempre desarrollamos sobre una línea base estable.

Mantenga el build rápido

El punto de integración continua es proporcionar una retroalimentación rápida.

Se dice que para lograr esto el tiempo ideal de compilación es de diez minutos. La idea de tener tiempos bajos para la construcción del build es ahorrar tiempo a los desarrolladores porque cada minuto menos de reducción es un minuto ahorrado para cada desarrollador. Sin embargo esta es una tarea difícil ya que el tiempo del build implica también la ejecución de pruebas, agregando más tiempo a la tarea de de construcción. El punto aquí está en encontrar el equilibrio entre tiempo y capacidad de detectar errores.

Hay tres estrategias que se pueden implementar para mantener el build rápido:

1. ***Estrategia 1: tener 2 builds (Integración en etapas – Síncrona y Asíncrona)***

   Esta es una de las estrategias más utilizadas a la hora de cumplir con esta práctica y consiste en dos etapas:

   La primera etapa sería hacer las pruebas de compilación y de ejecución, que son las pruebas unitarias y de integración (de componentes, no de sistema) y algunas pruebas críticas que involucren poco o ningún acceso a datos. Estas pruebas pueden correr muy rápido, manteniendo el build dentro del tiempo ideal de diez minutos. La desventaja es que cualquier error que pueda existir relacionado con el acceso a datos no será encontrado.

   En la segunda etapa se ejecuta una suite diferente de pruebas que sí involucran el acceso a datos reales y además de pruebas asociadas al comportamiento de la aplicación. Esta suite podría llegar a tardar un par de horas en ejecutarse. Se trata de un build secundario que está separado del principal y que se ejecuta cuando se puede a partir del último ejecutable bueno surgido a partir del último build exitoso. Si el build secundario falla no es necesario detener todo, como sí lo sería en el caso del principal, pero el equipo sí debe arreglar los errores lo más rápido posible para evitar que se sigan propagando y que el impacto sea mayor. Puede ocurrir que lleve uno o dos días poder atender estos errores, dependiendo del estado del proyecto. Si esto ocurre, no es algo crítico, ya que quien falló fue el build secundario, pero se debe hacer lo posible por solucionar los errores rápidamente. De esta forma en esta etapa se pueden ejecutar pruebas más complejas como pruebas de performance, estabilidad, carga, etc. Debido a que

este build puede ejecutarse sin que el desarrollador que realizó el commit tenga que finalizar su finalización (a diferencia del primero) se suele decir que corre de manera asíncrona o que es un build asíncrono.

Una aclaración importante a esta estrategia es que hay quiénes consideran que tener múltiples builds no es practicar integración continua, ya que uno de los objetivos es estar siempre listos para liberar software que funcione, y no podemos decir que nuestro producto está libre de errores y que podemos liberarlo cuando solamente ha superado exitosamente las pruebas del build rápido. (Ver Builds Manuales y Servidores de Integración Continua)

2. ***Estrategia 2: mejorar los test por etapas***

   Una estrategia a considerar cuando se encuentran errores en el build secundario y no en el primario es agregar los tests que fallaron al build principal para asegurarse de que han sido corregidos y no tener que esperar a que el segundo build se ejecute, ya que como dijimos este build no se atiende de manera inmediata.

3. ***Estrategia 3: Agrupar de acuerdo al diseño***

   Esta estrategia es la predominante cuando se habla de grandes grupos trabajando en un mismo producto. El grupo se divide en subgrupos que trabajan sobre diferentes módulos de la aplicación.

   En teoría, este tipo de modelo basado en la propiedad del código tiene responsabilidades e interfaces claramente definidas, que permiten a cada equipo trabajar en su propia área sin tener que preocuparse en la integración con el resto del producto. Sin embargo, la realidad indica que las aplicaciones nunca se comportan tal y como se espera y que las integraciones **Sí** fallan, por lo que la integración continua sigue siendo una buena alternativa para trabajar bajo esta estrategia.

   En este entorno, la integración continua se enfrenta a problemas asociados con grandes números de personas trabajando sobre la misma rama al mismo tiempo. Sin embargo, es posible dar a cada equipo su propia rama de integraciones locales sobre la cual cada equipo puede trabajar sin afectar a los demás equipos. Cada una de estas ramas se integra con la principal regularmente, por ejemplo cada dos horas. Esto reduce la cantidad de commits sobre la línea principal y por lo tanto la cantidad de integraciones.

   La desventaja que tiene esta estrategia es que se requiere que los módulos, los puntos de integración y las API estén claramente definidos, por lo que se requiere llevar a cabo el diseño de la aplicación por adelantado. Esto implica que el diseño no podrá ser modificado fácilmente y sabemos que no siempre se puede diseñar bien la primera vez y a veces estos diseños necesitan ser refinados. Esta desventaja es de sumo peso a la hora de tomar decisiones y es el principal motivo por el cual las organizaciones prefieren que el código sea de propiedad colectiva, y no de propiedad de equipos.

Probar en un ambiente clonado del entorno de producción

El objetivo aquí es eliminar el riesgo asociado al entorno, ya que si los entornos son diferentes no estamos en condiciones de asegurar que lo que pasó en el entorno de pruebas será lo que sucederá (o no) en el entorno de producción.

Sabemos que puede ser muy difícil e incluso hasta imposible tener ambientes clonados por cuestiones de costo. Debido a esto, una buena opción es utilizar máquinas virtuales para simular los ambientes.

Debe ser fácil para cualquier persona conseguir el último ejecutable

Una de las partes más difíciles del desarrollo de software es asegurarse de que se construye el software adecuado. Es muy difícil precisar lo que un cliente quiere de antemano y que esta especificación sea correcta es aún más difícil. Suele ser más fácil para las personas ver algo que no es del todo correcto y decir a partir de esto qué es lo que necesita más precisamente. Los procesos ágiles usan esta característica de los clientes para mejorar el producto bajo desarrollo.

Es por esto que cualquiera que esté involucrado con un proyecto de software debe ser capaz de obtener la última versión ejecutable y ser capaz de ejecutarla a fines de: demostraciones, pruebas exploratorias, o simplemente para ver lo que ha cambiado esta semana.

Todo el mundo debe poder ver lo que está sucediendo

La integración continua se basa en la comunicación y es necesario que todo el mundo pueda ver fácilmente el estado del sistema y los cambios que se han hecho a él.

Por ejemplo, cada día el grupo de control de calidad podría poner un color verde en el día si recibieron una versión estable que pasó las pruebas o un cuadrado rojo en caso contrario.

Automatizar el despliegue

Para trabajar con Integración Continua se necesitan múltiples entornos, uno para ejecutar las pruebas al realizar un commit sobre la línea base y uno o más para ejecutar las pruebas secundarias. Esto implica instalar ejecutables varias veces al día por lo que es una muy buena idea automatizar esta tarea. Con esto se logra u rápido despliegue en diferentes ambientes.

Frecuencia de integración

Como dijimos, uno de los objetivos de la integración continua es tener el código listo para entregar. Para ello necesitamos responder una pregunta: ***Cuán frecuentemente debemos integrar nuestro código?*** Lo cierto es que no hay una respuesta exacta para esta pregunta. Algunos dicen que lo ideal es integrar cada dos horas y que en el peor de los casos debemos integrar nuestro código una vez al día. Sin embargo, cuando empezamos a trabajar aplicando esta práctica, el peor de los casos es el mejor inicio. En mi opinión, este tiempo debería ser acordado entre los miembros del equipo considerando, al definir el mismo, las siguientes premisas:

A Mayor tiempo sin integrar el código, Mayor es la probabilidad de tener conflictos y Mayor será el esfuerzo necesario para la resolución de los mismos.

Mientras más frecuente se lleven a cabo las integraciones, lo más probable es las mismas sean triviales reduciendo la probabilidad de conflictos.

Sin embargo, tengamos en cuenta también que si las integraciones son demasiado frecuentes y hay muchas personas integrando (porque el equipo es grande) se puede llegar a formar una cola de espera de integración.

Builds Manuales y Servidores de Integración Continua

Cuando hablamos de ejecutar builds manuales, tarea conocida también como ***"ejecutar el Script de Integración Continua"***, estamos hablando de un build en el cual el desarrollador lleva a cabo todas las acciones desde que sube su código al repositorio hasta que el build finaliza y ha verificado que el mismo está libre de errores, es decir, está confirmando que el build y las pruebas pasaron exitosamente antes de pasar a la siguiente tarea.

Cuando hablamos de servidores de integración continua hacemos referencia a servidores que se encargan del proceso de armado del build, generación de ejecutables y ejecución de pruebas sin la presencia o control del desarrollador. Solamente basta con que este lleve a cabo un commit para que el servidor comience a realizar su trabajo. Mientras el desarrollador puede seguir adelante con su trabajo.

Se suele decir que cuando ejecutamos el script estamos ejecutando una ***integración síncrona*** y cuando utilizamos servidores de integración, estamos llevando a cabo una ***integración asíncrona***.

Dicho esto, vamos a explicar de qué se trata cada uno de ellos en más detalle:

Build Manual - El Script de Integración Continua – Integración Síncrona

Para ejecutar este tipo de integración necesitamos dos elementos:

1. Una PC de integración
2. Un Token de integración

Pasos:

Lo primero que tenemos que realizar es actualizar nuestro código con el que se encuentra en el repositorio. Para ello debemos:

1. Comprobar que el token integración está disponible. Si no lo está, otro miembro del equipo está haciendo commit de su código (checking in), por lo que debemos esperar hasta que termine.

2. Si el token está disponible lo tomamos y comenzamos a hacer check out (bajamos localmente el código) de los últimos cambios que se llevaron a cabo en el repositorio. No hay problemas si varios miembros del equipo realizan check out al mismo tiempo, pero nadie puede tomar el token hasta que terminemos de subir nuestro código y de integrarlo. Aquí puede suceder:
    a. Que ocurran conflictos cuando se hace el check out. En este caso deberás resolverlos (haciendo merge o lo que corresponda) y luego realizando el build en tu PC (incluyendo la ejecución de las pruebas).
    b. Que no existan conflictos (es posible) y que comencemos a realizar el build localmente (siempre incluyendo la ejecución de las pruebas)

3. Hacer check in del código (o commit).

4. Ir a la máquina de integración, bajar los cambios y ejecutar el build (obviamente incluyendo las pruebas). Aquí puede suceder lo siguiente:
    a. Que el build falle en la máquina de integración, por lo que deberemos solucionar el problema antes de dar por finalizada la integración.
    b. Que el build sea exitoso, por lo que se considera que la integración ha finalizado y se debe de devolver el token de integración.

Nota:

Puede suceder que el build esté fallando en nuestra PC (ambiente local) a causa de alguna configuración que no tengamos seteada apropiadamente. Una alternativa para determinar el origen de la falla es hacer el build en la PC de integración para validar que efectivamente sea este el problema. Si no es un problema de configuración, como dijimos anteriormente, deberemos hacer un rollback para determinar la causa de los errores y resolverlos.

Servidores de Integración Continua – Integración Asíncrona

La integración asíncrona requiere la utilización de una herramienta de CI (IC). Estas herramientas tienen la particularidad de no requerir que el build pase antes de que el código sea verificado. Como resultado, los errores que se encuentran en el build se propagan y provocan demoras.

El mayor problema con la integración asíncrona es que tiende a resultar en builds rotos. Si se hace check in de código que no funciona y recién nos enteramos media hora o una hora más tarde, tendremos que interrumpir la tarea que estamos haciendo para arreglar el problema. Si alguien hizo check out de este código, entonces va a tener problemas.

En resumen

Manual o Síncrona	Server o Asíncrona
1. Check Out 2. Build 3. Chequear resultados 4. Check In 5. Check Out en la PC de Integración 6. Build 7. Chequear resultados	1. Check Out Automático 2. Iniciar el build 3. Avisar al desarrollador (generalmente por mail) 4. Si hay errores, se deben resolver 5. Si no hay errores, se considerada terminada la integración

Beneficios de la Integración Continua

Reduce el riesgo de fallas en la integración sobre la línea base

Al llevar a cabo esta práctica no queremos decir que no van a haber errores en las integraciones que llevemos a cabo, pero estamos reduciendo el mismo ya que la tendencia es integrar código que está funcionando, porque los builds y los tests pasaron exitosamente.

Algunos de los riesgos que ayuda a mitigar esta práctica son:

- La falta de cohesión de software, despliegue
- Descubrimiento de defectos de última hora

- Baja calidad del software
- La falta de visibilidad del proyecto

Integraciones frecuentes

Las integraciones frecuentes permiten llevar cabo otra práctica conocida como entregas frecuentes (continuous deployment), lo que permite poder entregar frecuentemente código al cliente para que sea validado y de esta manera poder recibir feedback respecto de si nuestro trabajo es lo que el espera.

Un beneficio adicional es que nos permite estar en contacto con el cliente, lo cual es sumamente importante en entornos de desarrollo ágiles.

Conocimiento del estado de la aplicación (qué funciona, qué no funciona, bugs)

Nos permite conocer qué está funcionando, qué no y cuáles son los errores existentes. Aporta mayor visibilidad del estado del software. Algunas de las ventajas de conocer el estado de la aplicación son:

- ✓ Decisiones eficaces: Un sistema de CI puede proporcionar información JIT (Just In Time - justo a tiempo) sobre el estado del build y métricas de calidad. Algunos sistemas de CI también pueden mostrar la tasa de defectos y el estado de completitud de las funcionalidades a ser entregadas.
- ✓ Identificación de tendencias: tendencias en el éxito o el fracaso de construcción, calidad general y otra información del proyecto.

Menos bugs

Cuando integramos continuamente estamos ejecutando un set de pruebas que dijimos no debe arrojar errores. La reducción de los bugs está asociada a cuán buena sea la suite de pruebas que tenemos asociada al producto.

Los bugs se encuentran y corrigen más rápido

Debido a que integramos y ejecutamos las pruebas e inspecciones (de código) varias veces al día, hay una mayor probabilidad de que se descubran y reparen los defectos encontrados más rápidamente. Esto se debe a que es más acotado el espectro en el cual se debe buscar y corregir el mismo porque la cantidad de código modificado en cada integración es menor.

Reducir procesos repetitivos

Mediante la automatización podemos asegurar que en cada integración:

- ✓ El proceso se ejecuta de la misma manera cada vez.
- ✓ Se sigue un proceso ordenado. Por ejemplo, se ejecutan las inspecciones (análisis estático) antes de ejecutar los scripts de prueba.

Esta automatización de los procesos permite a la gente dedicar su tiempo a resolver cuestiones más importantes y productivas relacionadas al desarrollo del producto de software.

Genera software desplegable

Al trabajar con esta práctica, y cumpliendo con los principios de Agile mencionados anteriormente, podemos estar en condiciones de entregar SW que funcione cuando nos lo soliciten, a fin de que el mismo sea validado de manera temprana y los defectos sean descubiertos y solucionados rápidamente.

Establecer mayor confianza en el producto

Esto se debe a que el build se considera correcto si las pruebas estáticas y funcionales han sido superadas exitosamente y estas se ejecutan cada vez que alguien hace un commit al repositorio.

Aclarando Conceptos

Integración Continua (Continuous Integration)

Builds frecuentes que pasan los tests.

Problema: NO tener una PC de integración.

Entrega Continua (Continuous Delivery)

Esto significa que no solamente estamos integrando y compilando frecuentemente incluyendo la ejecución de las pruebas, sino también que estamos desplegando nuestro código frecuentemente en un ambiente de pruebas.

Clave: Automatizar el despliegue.

Despliegues Continuos (Continuous Deployment)

La aplicación está lista para entregarse en el ambiente de producción

Problema: Más riesgo porque se despliega en el ambiente de producción (o sea al cliente)

Forma de Mitigación del riesgo: Incrementar la cantidad de pruebas sobre el código. Probablemente sea necesario combinar los dos tipos de integración.

Nota:

Esta distinción entre los tres conceptos tiene otras definiciones dependiendo del autor. A continuación cito una distinción realizada por Martin Fowler, la que otorga esta definición diferente a lo ya visto. El menciona que la definición anterior de Despliegues Continuos aplica para Entrega Continua, y redefine el concepto de Despliegues Continuos. De esta forma, las definiciones quedarían de la siguiente manera para los conceptos mencionados:

Entrega Continua (Continuous Delivery)

La aplicación está lista para entregarse en el ambiente de producción.

Despliegues Continuos (Continuous Deployment)

Cada build bueno es liberado a producción.

Problema: Más riesgo porque se despliega en el ambiente de producción (o sea al cliente)

Forma de Mitigación del riesgo: Incrementar la cantidad de pruebas sobre el código. Probablemente sea necesario combinar los dos tipos de integración.

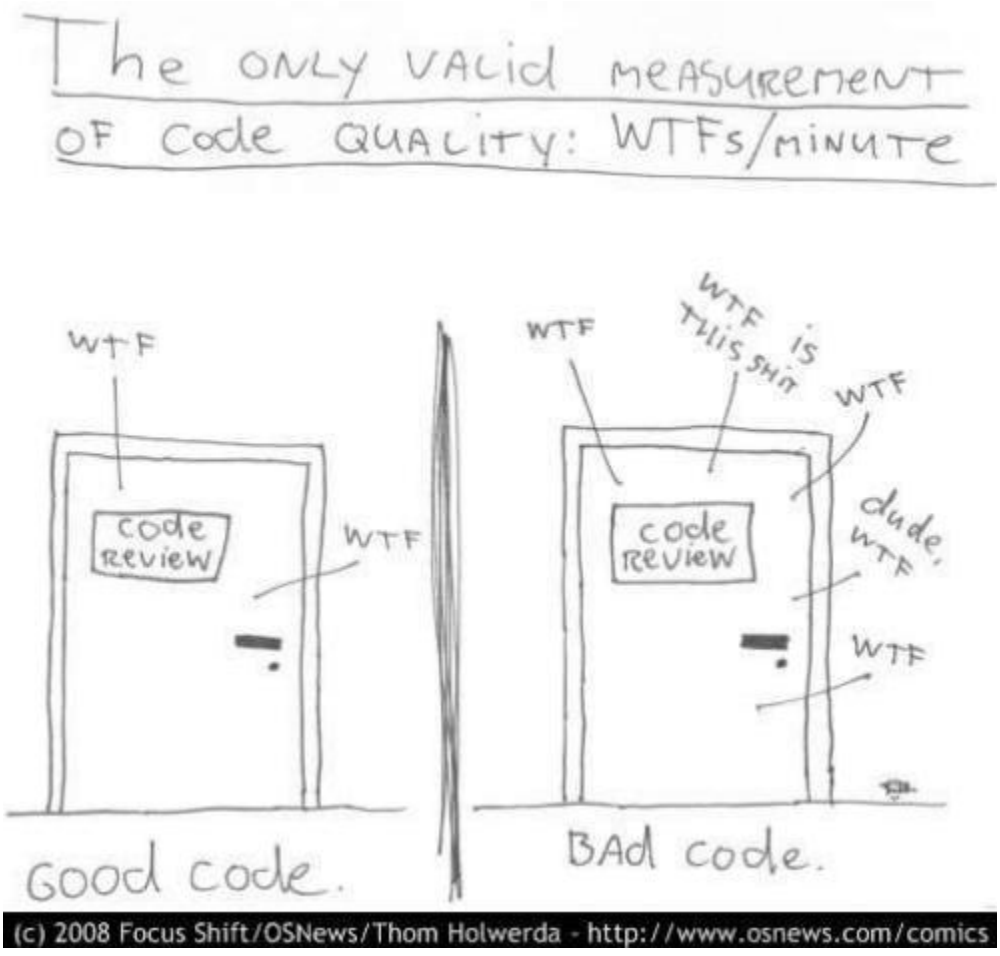
Métricas

Entre las métricas que se suelen tomar de esta práctica se pueden identificar:

- Mantenibilidad

- Extensibilidad
- Seguridad
- Fiabilidad
- Cumplimiento de estándares
- Tasa de errores
- Tests ejecutados exitosos
- Tests ejecutados fallidos
- Cantidad de builds exitosos
- Cantidad de builds fallidos
- Duplicidad de código
- Cantidad de líneas de código
- Complejidad ciclomática
- Cobertura de código
- Comentarios en el código

Las métricas proporcionan visibilidad del estado del proyecto y la forma más sencilla de obtenerlas es a través de la utilización de una herramienta de integración continua.



Herramientas

A pesar de que la integración continua es una práctica que no requiere de ninguna herramienta, es sumamente útil contar con un servidor de integración continua que nos ayude, como dijimos anteriormente, con la construcción automática de los builds y la ejecución de las pruebas contenidas en los mismos. A continuación se muestra un resumen de algunas de las herramientas más utilizadas. La fuente se obtuvo del siguiente informe:

Continuous Integration - What companies expect and solutions provide

Author: Georg Fleischer

Contact: Fleischer.Georg@gmail.com

Se comparó el mismo con información obtenida de los siguientes sitios que hacen referencia a diferentes herramientas utilizadas para integración continua, tanto pagas como no:

http://www.continuousintegrationtools.com/?opensource

http://www.continuousintegrationtools.com/?commercial

http://www.continuousintegrationtools.com/?reviews

http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix

El informe lista: herramienta, plataforma de instalación, como implementa los builds, sistema de control de versiones contra el cual puede funcionar, qué dispara la generación del build, formas de visualizar los resultados de las pruebas, notificación de los resultados.

Herramienta

Solution	Origin	Open source	Current version	First release	Link
AnthillPro	Urbancode	-	3.5	2001	http://www.anthillpro.com
Bamboo	Atlassian	-	2.1.5	2007	http://www.atlassian.com
Continuum	Apache project	●	1.2.3	2005	http://continuum.apache.org
Cruise [1)]	ThoughtWorks	-	1.1	2008	http://studios.thoughtworks.com/cruise
Cruise Control	Sourceforge project	●	2.8.2	2001	http://cruisecontrol.sourceforge.net
FinalBuilder	VSoft Technologies	-	6.2	2001	http://www.FinalBuilder.com
Hudson	java.net project	●	1.274	2007	http://hudson.dev.java.net
Lunt build	Javaforge project	●	1.6	2004	http://luntbuild.javaforge.com
Parabuild	Viewtier	-	3.2	2005	http://www.viewtier.com
Pulse	Zutubi	●	2.0	2006	http://www.zutubi.com/
Quick build [2)]	PMEase	●	2.0	2004	http://www.pmease.com
TeamCity	JetBrains	●	4.0	2006	http://www.jetbrains.com/teamcity

Plataforma

Solution	Installation platform	Build languages	Web-frontend	Configuration
AnthillPro	Java 1.5	All	Yes	Web-frontend
Bamboo	Java 1.5	All	Yes	Web-frontend
Continuum	Java 1.5	All	Yes	XML file
Cruise	Java 1.6	All	Yes	Web-frontend
Cruise Control	Java 1.4	All	Yes	XML file
FinalBuilder	Windows	All	Yes	Web-frontend
Hudson	Java 1.4	All	Yes	Web-frontend
Lunt build	Java 1.4	All	Yes	Web-frontend
Parabuild	Windows, Linux, MacOS X, Solaris, HPUX-11, Generic Unix	All	Yes	Web-frontend
Pulse	Java 1.5	All	Yes	Web-frontend
Quick build	Java 1.4	All	Yes	Web-frontend
TeamCity	Windows, Linux, MacOS X	All	Yes	Web-frontend

Build

Build implementations	AnthillPro	Bamboo	Continuum	Cruise	Cruise Control	FinalBuilder	Hudson	Lunt build	Parabuild	Pulse	Quick build	TeamCity
Ant	●	●	●	●	●	●	●	●	-	●	●	●
Command line tool	●	●	●	●	●	●	●	●	●	●	●	●
Maven	●	●	●	-	●	-	●	●	-	●	●	●
MSBuild	-	●	-	-	-	●	P	-	-	●	-	●
NAnt	-	●	-	●	●	●	P	-	-	-	●	●
Rake	-	-	-	●	●	-	●	●	-	-	●	●

Sistema de control de versiones

Version control system	AnthillPro	Bamboo	Continuum	Cruise	Cruise Control	FinalBuilder	Hudson	Lunt build	Parabuild	Pulse	Quick build	TeamCity
Accurev	●	-	-	-	●	●	●	●	-	-	●	P
AlienBrain	-	-	-	-	●	●	-	-	-	-	-	-
Bazaar	-	-	●	-	-	-	-	-	-	-	-	-
BitKeeper	-	-	-	-	-	-	P	-	-	-	-	-
ClearCase	●	●	●	-	●	●	P	●	●	-	●	●
CVS	●	●	●	-	●	●	●	●	●	●	●	●
Dimension	●	●	-	-	-	-	-	-	-	-	-	-
Git	-	-	-	●	●	-	P	-	-	●	-	-
Harvest	●	-	-	-	●	-	-	-	-	-	-	-
Jedi	-	-	-	-	-	●	-	-	-	-	-	-
Mercurial	●	●	●	●	●	-	P	-	-	-	-	P
MKS Source Integrity	●	-	-	-	●	●	-	-	●	-	-	-
Perforce	●	●	●	●	●	●	P	●	●	●	●	●
PureCM	-	-	-	-	-	●	-	-	-	-	-	-
PVCS	●	-	-	-	●	●	P	-	●	-	-	-
QCVS	-	-	-	-	-	●	-	-	-	-	-	-
SourceGear Vault	●	-	-	-	-	-	-	-	●	-	-	-
StarTeam	●	-	●	-	●	●	P	●	●	-	●	-
Subversion	●	●	●	●	●	●	●	●	●	●	●	●
Surround	-	-	-	-	●	●	-	-	●	-	-	-
Synergy	●	-	●	-	●	-	P	-	-	-	-	-
Team Coherence	-	-	-	-	-	●	-	-	-	-	-	-
Team Foundation Server	●	-	-	-	●	●	P	-	-	-	-	●
Visual Source Safe	●	-	●	-	●	●	P	●	●	-	●	●

Disparadores del build

Build triggers	AnthillPro	Bamboo	Continuum	Cruise	Cruise Control	FinalBuilder	Hudson	Lunt build	Parabuild	Pulse	Quick build	TeamCity
API call	●	●	●	●	-	-	-	●	-	●	●	-
Manual	●	●	●	●	-	●	●	●	●	●	●	●
Repository check-in	●	●	●	●	●	●	●	●	●	●	●	●
Scheduled	●	●	●	-	●	●	●	●	●	●	●	●
Successful finished build	●	●	●	●	●	●	●	●	●	●	●	●

Herramientas de prueba

Test tools result evaluation and rendering	AnthillPro	Bamboo	Continuum	Cruise	Cruise Control	FinalBuilder	Hudson	Lunt build	Parabuild	Pulse	Quick build	TeamCity
Agitar	●	-	-	-	●	-	-	-	-	-	-	-
AQTest	-	-	-	●	-	●	-	-	-	-	-	-
CppUnit	●	●	-	●	-	-	-	-	●	●	-	P
Gallio	-	-	-	-	-	-	-	-	-	-	-	P
JUnit	●	●	-	●	●	-	●	-	●	●	-	●
MbUnit	-	-	-	●	-	●	-	-	-	-	-	-
MSTest	-	●	-	●	-	●	-	-	-	-	-	●
Nose	-	●	-	●	-	-	-	-	-	-	-	P
NUnit	●	●	-	●	●	●	-	-	●	-	-	●
ocunit	-	-	-	-	-	-	-	-	-	●	-	-
PHPUnit	-	-	-	●	-	-	-	-	●	-	-	-
Python Unit	-	-	-	-	-	-	-	-	-	-	-	P
Quality Center & QuickTest Pro	●	-	-	●	-	-	-	-	-	-	-	-
SilkCentral	●	-	-	●	-	-	-	-	-	-	-	-
TestNG	-	-	-	-	-	-	●	-	-	-	-	-
Typemock	-	-	-	●	-	●	-	-	-	-	-	-

Servicios de notificación

Notification services	AnthillPro	Bamboo	Continuum	Cruise	Cruise Control	FinalBuilder	Hudson	Lunt build	Parabuild	Pulse	Quick build	TeamCity
Blog	-	-	-	-	-	-	-	●	-	-	●	-
Email	●	●	●	-	●	●	●	●	●	●	●	●
Google Calendar	-	-	-	-	-	-	● (p)	-	-	-	-	-
Google Talk	●	●	●	-	-	-	-	-	-	-	-	-
ICQ	-	-	-	-	-	●	-	-	-	-	-	-
IRC	-	-	●	-	-	-	● (p)	-	-	-	-	-
Jabber	●	●	●	-	●	-	● (p)	●	●	●	●	●
MSN	●	-	●	-	-	●	-	●	-	-	●	-
Nabaztag	-	-	-	-	-	-	● (p)	-	-	-	-	● (p)
Newsgroup	-	-	-	-	-	●	-	-	-	-	-	-
RSS	-	●	-	-	●	●	●	-	●	●	●	●
Sametime	-	-	-	-	●	-	-	●	-	-	●	-
System tray notifyer	-	-	-	-	●	●	● (p)	-	●	●	● (p)	●
Twitter	-	-	-	-	-	-	● (p)	-	-	-	-	-
Wagon	-	-	●	-	-	-	-	-	-	-	-	-
Yahoo	●	-	-	-	●	-	-	-	-	-	-	-

RTC y el Soporte a la Integración Continua

Como dijimos anteriormente, existen varias herramientas que nos permiten llevar a cabo la integración continua. Una de estas herramientas es Rational Team Concert (en adelante RTC) que entre otras cosas ayuda a los equipos de desarrollo de software a implementar un sistema de integración continua. A continuación veremos como cada una de las prácticas que mencionamos anteriormente (ver Prácticas de Integración Continua) puede ser soportada por esta herramienta:

Mantener un único repositorio de fuentes.

RTC implementa un componente de control de código fuente que administra el código fuente, documentos y otros artefactos que un equipo crea. Un aspecto a destacar en la utilización de RTC es que permite el desarrollo simultáneo de múltiples versiones de artefactos compartidos, por lo que los equipos pueden trabajar en varias líneas (branches) de desarrollo al mismo tiempo.

Automatizar la construcción del Build

Se debe reducir el tiempo necesario para construir versiones automáticas a través del análisis de que ha cambiado respecto del build anterior con el fin de determinar que tiene que ser reconstruido.

RTC implementa un componente denominado Team Build, que proporciona apoyo para la automatización, monitoreo y conocimiento de los builds de un equipo. Este componente incluye un Build Engine y un conjunto de herramientas Ant. El kit de herramientas Ant puede realizar diversas operaciones como indicar el progreso del build, publicar los artefactos construidos, mantener logs de los builds y publicar los resultados del build y de los tests. Cabe aclarar que no sólo se pueden utilizar las herramientas de Ant, sino también otros lenguajes de scripting pueden ser considerados para integrarse con el Team Build.

Hacer el Build Auto-Testeable

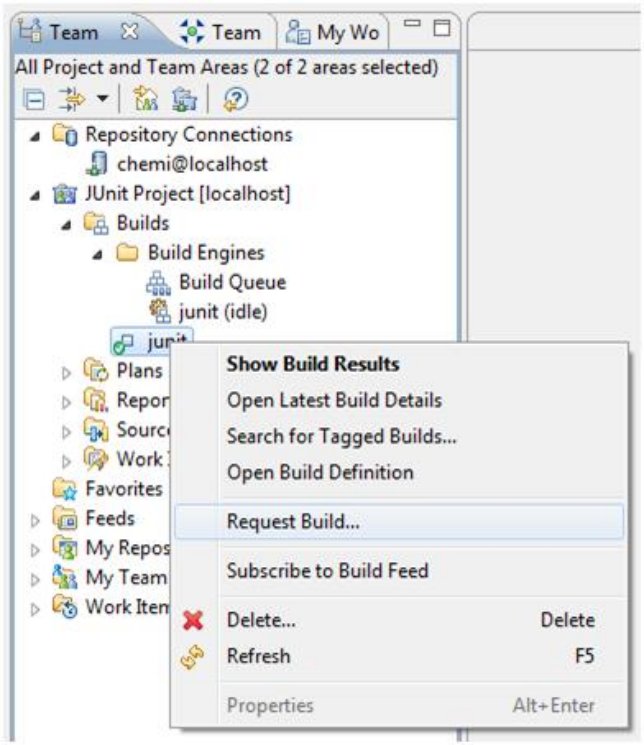
RTC no es un IDE, por lo que no tiene las características para definir e implementar las pruebas unitarias. Sin embargo, su componente Team Build incluye soporte para la ejecución y el análisis de los resultados de los distintos frameworks de pruebas unitarias, como JUnit, NUnit, etc. Así los equipos de desarrollo serán capaces de ejecutar y comprobar los resultados de todas las pruebas unitarias del proyecto.

Commits diarios a la línea base

El componente de RTC que permite llevar a cabo los builds incluye una característica denominada construcción personal, la cual permite a un desarrollador de ejecutar un build en su propio espacio de trabajo (workspace) en lugar de la línea principal. El resto del equipo no será notificado de este tipo de construcciones.

Cada Commit implica un build en la línea base en una máquina de Integración

El componente Team Build de RTC permite a los desarrolladores (si tienen los permisos adecuados) solicitar la ejecución de un build.



Otra forma de llevar a cabo un build es que el mismo sea disparado automáticamente cuando se realiza un commit sobre el repositorio.

Una tercera alternativa que RTC ofrece es integrar el código fuente y el Team Build basado en el concepto de acciones de seguimiento. Estas acciones de seguimiento son declaraciones lógicas que se ejecutan después de que un evento específico se ha producido en el servidor. Así la idea principal sería crear una acción de seguimiento cada vez que se hace un commit sobre el repositorio.

RTC incluye un gran número de acciones de seguimiento ya incorporadas, pero pueden ser creadas algunas nuevas.

Por último, una alternativa que también es utilizada es crear tareas programadas con RTC para que los builds sean disparados en una determinada frecuencia de tiempo.

Mantenga el build rápido

RTC no puede ayudar mucho en este aspecto. Lo que el Team Build puede hacer es proporcionar información para las áreas que provocan más demoras en el armado del build. Así se pueden detectar las posibles áreas a mejorar para futuras compilaciones.

Probar en un ambiente clonado del entorno de producción

El componente Team Build de RTC puede ser instalado en varios ambientes de manera simultánea y apuntar al mismo servidor de RTC.

Debe ser fácil para cualquier persona conseguir el último ejecutable

Como dijimos anteriormente, el componente Ant de RTC posee varias herramientas las cuales incluyen una serie de tareas, las que permiten ejecutar varias acciones cuando se lleva a cabo un build. Entre ellas hay dos que permiten al build publicar los artefactos. Estas son: son **artifactLinkPublisher** y **artifactFilePublisher**.

Todo el mundo debe poder ver lo que está sucediendo

RTC contiene un componente denominado Dashboard. Es un componente con una interfaz Web que provee información rápida respecto del estado del proyecto. Provee además la capacidad de poder hacer drill down de la información a fin de tener mayor nivel de detalle. Provee información respecto de todas las actividades incluidas en el build, historial de las ejecuciones, fecha de las ejecuciones de los builds, etc.

RTC también provee un componente denominado Reporting, el cual provee reportes con información de la duración de un build, estado del build, errores en las pruebas que se ejecutaron como parte del armado del build, etc.

Automatizar el despliegue

Como dijimos anteriormente, RTC posee el componente Team Build, el cual puede ser instalado en múltiples ambientes y plataformas, todos apuntando al mismo servidor de RTC y con las funcionalidades que tiene incorporadas este componente podemos automatizar los despliegues.

Opiniones, ejemplos y comentarios

1. Muchos equipos encuentran que este enfoque conduce a reducir significativamente los problemas de integración y permite a un equipo desarrollar software cohesivo más rápidamente.
2. Comúnmente hay dos reacciones cuando se habla de esta práctica:
   a) Decir "Esto no va a funcionar acá" y
   b) "Practicarlo (trabajar de esta manera) no hará mucha diferencia".
   Una vez que los equipos de trabajo comienzan a trabajar con esta práctica surge la tercera reacción, donde dicen:
   c) "Sí lo hacemos – No podríamos no hacerlo".
3. Respecto del negocio
   Lo ideal es que el código en el repositorio está siempre listo para ser liberado, pero a pesar de que pueda ser técnicamente capaz llevar a cabo un release (casi libre de errores, con las pruebas pasadas, etc.), no siempre va a estar listo para ser liberado desde una perspectiva de negocios.
4. Para garantizar un build que esté siempre operativo debemos:
   - Asegurarnos que lo que funciona en nuestra PC va a funcionar en la PC de cualquier miembro del equipo.
   - Asegurarnos que nadie baje código para el que no se ha verificado que genere un build exitoso.
5. La práctica de integración continua representa un cambio en el proceso de construcción del software. Se pretende que la integración de los componentes, la cual es una tarea poco frecuente y muy dolorosa (porque lleva mucho tiempo y esfuerzo), sea simple y parte de las actividades diarias de un desarrollador. Esta práctica permite avanzar constantemente con pequeños pasos.

Referencias

	Link
1.	http://agilemanifesto.org/
2.	http://continuousdelivery.com/
3.	http://www.agilejournal.com/articles/current-edition/6101-agile-journal-may-2011
4.	http://www.martinfowler.com/bliki/Xunit.html
5.	http://martinfowler.com/articles/continuousIntegration.html
6.	https://jazz.net/library/article/474
7.	http://www.itwriting.com/blog/4797-continuous-integration-vs-continuous-delivery-vs-continuous-deployment-what-is-the-difference.html
8.	http://www.extremeprogramming.org/rules.html
9.	http://en.wikipedia.org/wiki/Extreme_programming_practices
10.	http://jamesshore.com/Blog/Continuous-Integration-is-an-Attitude.html
11.	http://jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html
12.	http://jamesshore.com/Blog/Forces-Affecting-Continuous-Integration.html
13.	http://jamesshore.com/Agile-Book/continuous_integration.html
14.	http://martinfowler.com/delivery.html
15.	http://www.continuousintegrationtools.com/?opensource
16.	http://www.continuousintegrationtools.com/?commercial
17.	http://www.continuousintegrationtools.com/?reviews
18.	http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix
19.	http://blogs.oracle.com/theaquarium/entry/continuous_integration_needs_and_solutions
20.	http://www.javaworld.com/javaworld/jw-06-2007/jw-06-awci.html
21.	http://www.slideshare.net/gabrielspmoreira/software-product-measurement-and-analysis-in-a-continuous-integration-environment
22.	http://msdn.microsoft.com/en-us/library/cc948982(v=office.12).aspx