Integración Continua:
Solución a los problemas de productividad y calidad del código en un entorno ágil testigo

Ingeniería de Sistemas: Proyecto de Grado

Alumnos: ***Gonzalo Orozco*** *(IUA: Facultad de Ciencias de la Administración)*
Ligia G. Righi *(IUA: Facultad de Ingeniería)*

Tutor: ***Natalia Mira*** *(IUA: Facultad de Ingeniería)*

Carrera: ***Ingeniería de Sistemas***

INDICE

1. Dedicatoria

2. Agradecimientos

3. Resumen

El presente Trabajo Final de Grado nace como solución a una problemática de un equipo de desarrollo testigo. Problemática que es consecuencia del uso de metodologías ágiles sin tener el soporte y la infraestructura necesaria para conseguir los beneficios que estas metodologías aseguran.

La solución se basa principalmente en la utilización de la práctica de Integración Continua como eje del ambiente de desarrollo del equipo. Apoyados principalmente en esta buena práctica y en otras prácticas y metodologías complementarias, y con el soporte de herramientas especializadas y de libre uso, presentaremos una solución integral para el equipo de desarrollo testigo, y para cualquier otro equipo bajo condiciones similares.

Con este trabajo, intentaremos también justificar la elección de las herramientas elegidas para el equipo testigo y como cada una de ellas ayuda a mitigar los problemas identificados.

4. Glosario

Software Libre: Es el software que respeta la libertad de los usuarios y la comunidad. En grandes líneas, significa que los usuarios tienen la libertad para ejecutar, copiar, distribuir, estudiar, modificar y mejorar el software.

Open Source: Se califica como tales a los programas informáticos que permiten el acceso a su código de programación, lo que facilita modificaciones por parte de otros programadores ajenos a los creadores originales del software en cuestión. Dispone de la característica de presentar su código abierto, y el software libre (que puede descargarse y distribuirse de manera gratuita).

Agile: El desarrollo ágil de software refiere a métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan mediante la colaboración de grupos auto organizados y multidisciplinarios. Los métodos ágiles enfatizan las comunicaciones cara a cara en vez de la documentación.

Backlog: Es un conjunto de requisitos de alto nivel priorizados que definen el trabajo a realizar.

***Sprint**:* Organización del proceso de creación de software, en el cual el trabajo completo, se divide en distintos apartados o bloques, que pueden ser abordados en periodos cortos de tiempo.

Historia de Usuario: (User Story)Una historia de usuario describe funcionalidad que será útil para el usuario, o comprador, de un sistema software.

Framework: Es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Build: Es el producto generado por la realización de las tareas de compilación, prueba, inspección e instalación. Realizar un "build" tiene como resultado situar el código todo junto, en un ambiente donde funciona de una forma cohesiva.

Merging: Es la acción de aplicar diferencias entre dos rutas de trabajo o ramas (*branches*). Es muy útil para descartar cambios que se hayan subido por error al repositorio, o para incorporar cambios que se haya realizado paralelamente a la rama principal.

Commit: Acción de comprometer el código a un repositorio o rama específica para que otros puedan disponer de él.

Trunk: Se refiere a la rama de desarrollo principal de un determinado repositorio.

Pruebas de regresión: Se denominan pruebas de regresión a cualquier tipo de pruebas de software que intentan descubrir errores (bugs), carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del software, causados por la realización de un cambio en el programa.

Refactorizar: En ingeniería del software, el término refactorización se usa a menudo para describir la modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por limpiar el código. La refactorización se realiza a menudo como parte del proceso de desarrollo del software donde los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Las pruebas aseguran que la refactorización no cambia el comportamiento del código.

Mentoring: Es el ofrecimiento de consejos, información o guía que hace una persona que tiene experiencia y habilidades en beneficio del desarrollo personal y profesional de otra persona.

***Mocks**:* En la programación orientada a objetos se llaman objetos simulados (pseudoobjetos, mock object) a los objetos que imitan el comportamiento de objetos reales de una forma controlada. Se usan para probar a otros objetos en pruebas unitarias que esperan mensajes de una clase en particular para sus métodos.

5. Introducción

El aumento de la complejidad del desarrollo de software es un problema cada vez más importante, dado que los clientes piden software cada vez más sofisticado, con menos errores y con mayores capacidades [*Groth, 2004*].

Los desarrolladores de software valoran su precioso y escaso tiempo y por lo tanto necesitan tener el apoyo de la organización en la selección del conjunto de herramientas adecuadas y de las mejores prácticas para optimizar su trabajo. Este trabajo se centra en la integración continua (IC) como buena práctica y cómo esta permite a los desarrolladores automatizar y agilizar determinados procesos como así también eliminar los inconvenientes causados a menudo por largos ciclos de integración: construcciones de código que fallan, integracion manual de código y largas pruebas de regresión entre otros.

Al amparo del concepto de Integración Continua este trabajo intenta defender la idea que le da el título a nuestro proyecto de grado: "*Al aplicar Integración Continua se puede mejorar la productividad y a la vez asegurar la calidad del código*". Para ello utilizamos como objeto de estudio el trabajo diario de un equipo de desarrollo "ágil" de software. Con equipo ágil nos referimos a un equipo de desarrollo que utiliza alguna de las metodología agiles conocidas para llevar a cabo el análisis, diseño e implementación de productos de software.

Nuestro proyecto de grado nace de las necesidades que hemos identificado en un equipo de desarrollo real en el cual uno de los tesistas formo parte activa. De estas necesidades, y basados en el propio estudio que Martin Fowler hace de las prácticas de Integración Continua [*Fowler, 2006*] nace la solución que hemos identificado y para la cual intentaremos demostrar su viabilidad en este trabajo.

Con la aplicación de la práctica de IC queremos demostrar que es posible, entre muchos otros beneficios, mejorar la productividad y asegurar la calidad del código que entregaría este equipo de desarrollo ágil y por consiguiente cualquier otro equipo en una situación similar.

El desarrollo de este trabajo final se dirigirá entonces a identificar una solución viable a los problemas de integración de código y calidad. Esta solución definirá un nuevo esquema de trabajo (un Entorno de Desarrollo Integrado) valiéndose de herramientas existentes y, por necesidad, de libre uso.

El desarrollo de proyectos tradicionalmente se ha sustentado en herramientas comerciales de un elevado coste, desde el punto de vista de las licencias, como desde el punto de vista de la complejidad de su uso.

La ingeniería del software ha evolucionado a todos los niveles, gracias al desarrollo del software libre, a las metodologías denominadas ágiles y a las buenas

prácticas puestas a disposición a través de Internet. No obstante, la abundancia de alternativas puede suponer en muchos casos, un problema extra.

Existen muchas herramientas libres, de control de versiones, de construcción de componentes, de gestión de dependencias, de aprovisionamiento de artefactos, de despliegue en entornos distribuidos, de gestión de errores, etc. Para cada una de estas categorías es posible encontrar herramientas con distintas capacidades y alcance.

Uno de los objetivos del presente estudio parte de la necesidad de encontrar una solución óptima, basada en la interrelación de estas herramientas.

Es también nuestra intención demostrar al final de este trabajo como conviven estas herramientas en un ambiente testigo o prototipo.

6. Objetivos y Alcance del Trabajo

6.1. Propósito del Trabajo

El propósito de este trabajo, el cual le da el nombre al mismo es:

- Demostrar que con la práctica de Integración Continua podremos mejorar la productividad del equipo de desarrollo testigo asegurando la calidad del código entregado.

6.2. Objetivos

De este propósito se desprenden los siguientes objetivos:

- Definir un Entorno de Desarrollo Integrado apoyado en la práctica de IC.
- Encontrar una solución óptima basada en herramientas de Software libre.
- Instalar, configurar y aplicar esta solución a modo de demostración.

6.3. Alcance

El alcance de este trabajo es, como lo expresamos previamente, hacer frente a las problemáticas planteadas proponiendo un nuevo entorno de desarrollo integrado para el equipo de desarrollo bajo estudio.

Se buscará dentro del conjunto de herramientas de Software Libre, las mejores combinaciones posibles, teniendo en cuenta factores como capacidad de integración, extensibilidad, nivel de compromiso de sus respectivas comunidades de desarrolladores, disponibilidad de documentación, etc. Se planteará un modelo de Entorno de Desarrollo Integrado de Software apoyado en la Integración Continua que intentará probarse como

viable y se documentará todo el proceso seguido en el estudio así como las razones que motivaron el camino elegido en cada caso.

También es cierto que identificamos otros problemas inherentes a la etapa de Entrega del código (*Delivery*) que quedan fuera del alcance de este proyecto, limitándonos solo a la problemática del entorno de desarrollo del equipo bajo estudio. Solo agregaremos que hay una práctica totalmente complementaria a la IC (En realidad IC forma parte de esta) que se denomina Entrega Continua (*Continuous Delivery* [*J.Humble/D.Farley, 2010*]) con la cual se puede hacer frente a las problemáticas que enfrentan los equipos de delivery o de operaciones.

Quedaran fuera del alcance de este trabajo los siguientes puntos:

- Exponer el proceso formal de desarrollo utilizado, que como ya expresamos, se basa en una metodología ágil (a saber: SCRUM).
- El proceso de captación y formulación de requerimientos.
- Aspectos de la administración y planificación de los proyectos.
- La problemática existente en la etapa de Entrega de software al cliente (Delivery).

7. Marco Teórico

Introducimos a continuación los conceptos principales a explorar en nuestro trabajo. Algunos nos ayudaran a entender los problemas actuales y otros forman parte de la solución que tenemos intención de implementar.

7.1. Integración Continua (IC)

Si pensamos en la complejidad técnica en la que se basan los servicios actuales, en los que intervienen múltiples programas embebidos y conectados entre sí, no resulta extraño que se produzcan fallos. Sin embargo, hay que tener siempre presente que un error de programación, un error de cálculo de una aplicación o un error en el equipamiento de una red, puede ser suficiente para acabar con el negocio más atractivo y prometedor. La verdad es que *resulta complicado demostrar que la calidad del software es un factor decisivo para el éxito, pero lo que sí está claro es que la falta de calidad es un factor decisivo para el fracaso*, por lo que la responsabilidad que los equipos de desarrollo tienen sobre sus hombros es altísima.

En esta realidad, los responsables técnicos de los proyectos se ven obligados a superar los nuevos retos que plantean los entornos de desarrollo complejos, en los que intervienen múltiples equipos y en los que resulta habitual encontrarse con errores que incrementan los costes, la complejidad del proceso de integración y que ponen en riesgo el cumplimiento de los plazos, tales como el uso de librerías incompatibles que eternizan

los despliegues, código que no cumple con la normativa definida, generación de nuevos errores al corregir otros, etc.

Para lograrlo, se hace necesario plantearse la Calidad del Software desde una perspectiva global, que abarque el ciclo de vida completo del producto, teniendo en cuenta tanto la calidad final del mismo como la del proceso de su desarrollo. Así, contar con metodologías, estándares, buenas prácticas, normativas y directrices correctos resulta fundamental para construir el armazón que permita desarrollar y poner en producción un producto de software de calidad porque, de esta forma, los errores y los costes serán menores y la agilidad y eficacia crecerán. Es decir, no basta con probar y probar, sino con probar con criterio. Cuando se une un buen proceso con un buen producto, el éxito está prácticamente asegurado.

7.1.1. Definición

El concepto de Integración continua surgió en la comunidad de "Extreme Programming, XP" y patrocinada por Martin Fowler. Kent Beck fue el primero quién escribió acerca de la integración continua alrededor del año 1999.

Según la definición de Martin Fowler: "*La integración continua es una práctica de desarrollo de software en la cual los miembros de un equipo integran su trabajo frecuentemente, como mínimo de forma diaria. Cada integración se verifica mediante una herramienta de construcción automática para detectar los errores de integración tan pronto como sea posible. Muchos equipos creen que este enfoque lleva a una reducción significativa de los problemas de integración y permite a un equipo desarrollar software cohesivo de forma más rápida.*" *[Fowler, 2006]*

Para ello se pueden utilizar un software especializado en automatizar tareas y que estas se ejecuten de forma automática cuando se genere un evento (subir cambios –*commit*- al repositorio de código; compilación exitosa o no exitosa; ejecución no exitosa de pruebas automáticas; despliegue de componentes; etc.)

Algunas de las tareas que se pueden automatizar en estas plataformas son:

- Compilación de los componentes
- Integración de todo el desarrollo
- Ejecución de pruebas unitarias
- Ejecución de pruebas de integración
- Ejecución de pruebas de aceptación
- Obtener métricas de calidad de código
- Despliegues automáticos

Estos son solo algunos ejemplos de las tareas automáticas que se pueden realizar si aplicamos una metodología de Integración Continua apoyados en alguna herramienta

especializada. Realmente se pueden automatizar muchas más tareas, ya que son herramientas muy versátiles a las que se le pueden añadir nuevas funcionalidades.

7.1.2. Principios

7.1.2.1. Mantener un único repositorio de código fuente

Los proyectos de software implican cientos o miles de ficheros que necesitan ser organizados para construir un producto. Hacer el seguimiento de todos estos ficheros es costoso, sobre todo cuando hay varias personas involucradas. Para facilitar esta tarea han surgido las herramientas de gestión del código fuente o control de versiones, como Subversión o Git.

Todo se debe incluir en el repositorio. Se debe ser capaz de construir un proyecto a partir del código descargado del repositorio. También es útil incluir recursos de configuración del código. Se recomienda no subir todo aquello que se pueda construir.

Estos sistemas de control de versiones permiten la creación de ramas (*branches*) para el desarrollo. El uso de ramas es problemático y se debe usar lo mínimo posible, como para la corrección de errores de versiones anteriores o experimentos temporales.

7.1.2.2. Automatizar la construcción del proyecto

La construcción de un proyecto implica la compilación del código fuente, mover ficheros de un sitio a otro, cargar esquemas en base de datos, resolver dependencias del código con librerías de tercero, etc. Todas estas tareas deberían automatizarse para evitar errores y ganar tiempo.

Existen algunas herramientas como Maven o Ant que nos permiten facilitar la automatización de estas tareas.

7.1.2.3. Autodiagnóstico de la construcción

Dentro del proceso de construcción de un proceso se deberían realizar las pruebas unitarias y de integración del código, que pueden implementarse mediante herramientas de la familia XUnit. Estos tests pueden detectar errores en etapas tempranas del desarrollo haciendo mucho más ágil su detección y resolución.

7.1.2.4. Entregar los cambios diariamente a la línea base

La integración permite que los desarrolladores informen unos a otros de los cambios que han hecho. Si esto se hace de forma frecuente, todo el mundo sabrá rápidamente los cambios que se han producido.

El único prerrequisito para que un desarrollador entregue sus cambios a la línea principal o base es que se pueda construir el código de forma correcta, sin fallas. El desarrollador debe actualizar primero su copia local, resolver cualquier conflicto y construir su copia local. Si se hace de forma correcta, se subirá a la línea principal.

Haciendo esto de forma frecuente, lo desarrolladores encuentran rápidamente si hay un conflicto y lo arreglan tan pronto lo detectan, cuando todavía es fácil de arreglar. Como mínimo, los desarrolladores deberían entregar sus cambios una vez al día.

7.1.2.5. Construir la línea base tras cada entrega

A pesar de las entregas diarias de los desarrolladores, todavía se pueden producir errores de integración debido a falta de disciplina del equipo o diferencias ambientales entre las máquinas de los desarrolladores.

Para evitar estos errores hay que asegurarse de que se construya el proyecto en el ambiente destinado para la integración tras cada entrega a la línea base (*commit*). Es el desarrollador que está haciendo la entrega el responsable de que esta construcción se realice de forma correcta.

La construcción en el ambiente de integración se puede hacer de forma manual o mediante un servidor de integración continua como Jenkins, CruiseControl, Continuum, etc.

Un servidor de integración continua vigila el repositorio de código, y cada vez que se hace una entrega, el servidor recoge el código fuente del repositorio, lo construye y notifica al responsable de la entrega del resultado.

7.1.2.6. Mantener una ejecución rápida de la construcción del proyecto

Puesto que en un proceso de integración continua lo que cuenta es obtener resultados del proceso tan pronto como sea posible, es preciso que el proceso de construcción no se demore excesivamente.

7.1.2.7. Probar en una réplica del entorno de producción

Si estamos probando en un entorno distinto al de producción, cada diferencia entre entornos es un potencial riesgo. Para minimizar estos riesgos, se debería usar en ambos entornos el mismo servidor, el mismo gestor de base de datos, la misma versión del sistema operativo, librerías, etc.

Una práctica común es mantener un ambiente de pre-producción lo más similar posible al ambiente de producción, para realizar allí pruebas de integración.

7.1.2.8. Fácil obtención del último ejecutable del proyecto

Todo el mundo que esté involucrado en el desarrollo de un determinado software, como los desarrolladores mismos o el equipo de QA, debería ser capaz de obtener fácilmente el último ejecutable y de ejecutarlo, facilitando las demostraciones, la prueba y la revisión de los últimos cambios.

7.1.2.9. Publicar el estado del proyecto

La integración continua considera que la comunicación es muy importante, así que hay que asegurarse que todo el mundo puede ver fácilmente el estado del sistema y los cambios que se han realizado.

Uno de las cosas más importantes que hay que comunicar es el estado de la línea principal de desarrollo. Si se está usando un servidor de integración continua, probablemente se dispondrá de una herramienta web donde se puedan ver los trabajos en progreso en cada momento.

Si se quiere hacer esta información más visual, se pueden usar los llamados "radiadores de información", que no son más que elemento visuales como luces o muñecos que informan de un determinado estado. Por ejemplo, se puede usar una luz roja para indicar que se ha producido un error en el servidor de integración continua y una luz verde para cuando no haya errores.

7.1.2.10. Automatizar el despliegue

Para llevar a cabo la práctica de Integración Continua eficazmente se necesitan varios entornos o ambientes, como el de desarrollo, el de prueba y el de producción. Ya que se deben mover ejecutables entre múltiples entornos varias veces al día, será mejor hacerlo de forma automática, así que es necesario tener scripts que permitan el despliegue de aplicaciones entre entornos de forma sencilla.

7.1.3. Beneficios

Podemos clasificar los beneficios de la Integración Continua desde dos puntos de vista: Beneficios en términos técnicos y beneficios en término del negocio.

7.1.3.1. Beneficios en términos Técnicos

Con la práctica de la IC un equipo puede obtener una retroalimentación temprana ya que en el caso de que el software no compile o no pase las pruebas automáticas el sistema alertará al equipo. Se puede obtener un "build" de forma automática cada vez que se suba el código al repositorio si así se lo desea. Esto implica una reducción del riesgo a la hora de integrar el producto y permite la detección temprana de errores, ya que estos se detectaran también en la subida de código al ejecutarse las pruebas de integración y aceptación de forma automática.

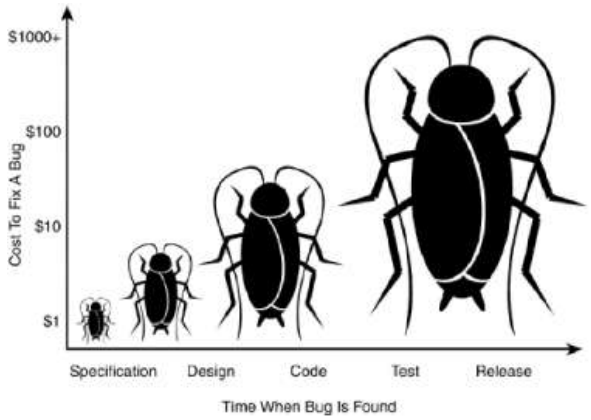
Al disponer en todo momento de ejecutables del proyecto, permite la rápida adopción por parte de los usuarios de las nuevas características añadidas al proyecto, permitiendo que valoren estos cambios y sugieran cambios nuevos de forma rápida.

La monitorización constante de las métricas del software (duplicidad de código, coberturas, complejidad, etc.) es también un gran beneficio que se puede aplicar fácilmente gracias a esta práctica y realmente a un bajo costo.

7.1.3.2. Beneficios en términos del Negocio

La integración frecuente de código conduce a la reducción de los niveles de riesgo de cualquier proyecto. Los defectos de código se pueden detectar más pronto y se arreglan más rápido, por tanto podemos tener métricas para medir el estado de la aplicación. Cuando un equipo de desarrollo integra su trabajo con frecuencia significa que hay poco recorrido entre el estado actual de la aplicación y lo que el desarrollador está implementando. Y así se reduce la posibilidad de supuestos.

Los recursos destinados a la creación/adopción de un sistema de IC se ve compensado por las horas hombre que se ahorran posteriormente en integración y despliegue. Encontrar un error en la etapa más temprana posible siempre es menos oneroso. Si el fallo se encuentra en fases más avanzada el coste es más alto. Todos hemos visto alguna vez un diagrama como el siguiente, donde especifica la relación del coste de encontrar un problema en las diferentes etapas del desarrollo.



En general, la aplicación efectiva de prácticas de Integración Continua puede proporcionar una mayor confianza en la producción software. Con cada construcción, el equipo sabe qué pruebas se ejecutan para verificar el comportamiento, que las normas de codificación de los proyectos se están cumpliendo, y que el resultado es un producto comprobable funcionalmente y repetible.

7.2. Tipo de Pruebas de Software

Si bien existen muchos tipos de clasificaciones para las pruebas de software, nosotros solo vamos a distinguir las que se pueden realizar en el ámbito de un equipo de desarrollo.

7.2.1. Pruebas de Unidad

En programación, una prueba unitaria es una forma de comprobar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto.

El objetivo de estas pruebas es aislar cada parte del programa y mostrar que las partes individuales son correctas. Proporcionan un contrato escrito que el trozo de código debe satisfacer.

7.2.2. Pruebas de Integración

Las pruebas de integración son aquellas que se realizan una vez que se han aprobado las pruebas unitarias. Se refieren a la prueba o pruebas de todos los elementos unitarios que componen un proceso, hecha en conjunto, de una sola vez.

Las pruebas de integración es la fase de la prueba de software en la cual módulos individuales de software son combinados y probados como un grupo. Son las pruebas posteriores a las pruebas unitarias y preceden a las pruebas del sistema.

Tienen como objetivo verificar el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente con el fin de comprobar que interactúan correctamente a través de sus interfaces, tanto internas como externas, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.

7.2.3. Pruebas de Sistema

Las pruebas de sistemas buscan discrepancias entre el programa y los objetivos o requerimientos, enfocándose en los errores hechos durante la transición del proceso al diseñar la especificación funcional.

Tienen el propósito de comparar el sistema o programa con sus objetivos originales (requerimientos funcionales y no funcionales).

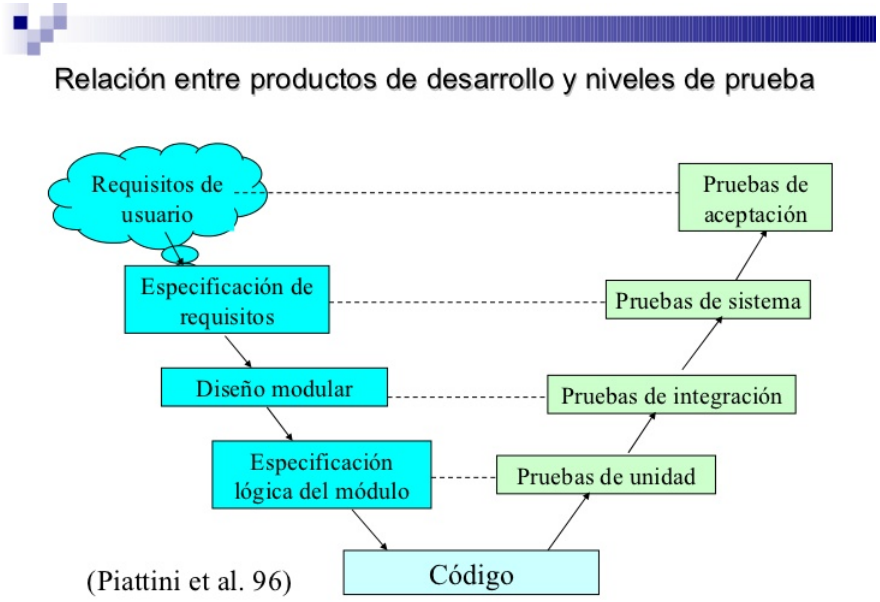
7.2.4. Pruebas de Aceptación

Una prueba de aceptación es un escenario de utilización del sistema y el comportamiento que de él se espera, visto desde la perspectiva del cliente, usuario o sistema externo que interactúa con el programa.

Una prueba de aceptación tiene como propósito demostrar al cliente el cumplimiento de un requisito del software y tiene las siguientes características:

- Describe un escenario (secuencia de pasos) de ejecución o uso del sistema desde la perspectiva del cliente
- Puede estar asociada a un requisito funcional o requisito no funcional
- Un requisito tiene una o más pruebas de aceptación asociadas
- Las pruebas de aceptación cubren desde escenarios típicos/frecuentes hasta los más excepcionales
- Una prueba de aceptación puede tener infinitas instanciaciones (ejecuciones con valores concretos).

Las pruebas de aceptación constituyen el criterio de éxito en cuanto a la implementación de un requisito del sistema. Las mismas deben ser validadas con el cliente y verificadas (en cuanto a corrección y completitud) por otros miembros del equipo asignados a la unidad de trabajo.



7.3. Desarrollo guiado por pruebas (TDD)

El Desarrollo Guiado por Pruebas de software, o "*Test-driven development*" (TDD) es una práctica de programación orientada a objetos que involucra otras dos prácticas: Escribir las pruebas primero (*Test First Development*) y Refactorización (*Refactoring*). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (*unit test*). En primer lugar, se escribe una prueba y se verifica que las pruebas fallan. A

continuación, se implementa el código que hace que la pruebas pasen satisfactoriamente y seguidamente se *refactoriza* el código escrito.

El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.

7.3.1. Características y Requisitos

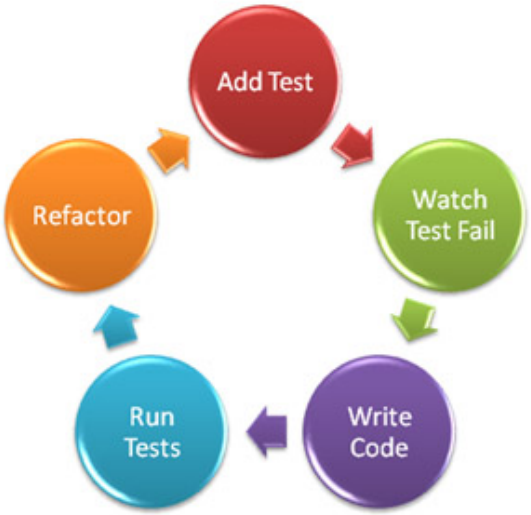
Para que funcione el desarrollo guiado por pruebas, el sistema que se programa tiene que ser lo suficientemente flexible como para permitir que sea probado automáticamente. Cada prueba debe ser suficientemente pequeña como para que permita determinar unívocamente si el código probado pasa o no la verificación que ésta le impone. El diseño se ve favorecido ya que se evita el indeseado "sobre diseño" de las aplicaciones y se logran interfaces más claras y un código más cohesivo.

Una ventaja de esta forma de programación es el evitar escribir código innecesario. Se intenta escribir el mínimo código posible, y si el código pasa una prueba aunque sepamos que es incorrecto nos da una idea de que tenemos que modificar nuestra lista de requerimientos agregando uno nuevo.

La generación de pruebas para cada funcionalidad hace que el programador confíe en el código escrito. Esto permite hacer modificaciones profundas del código (posiblemente en una etapa de mantenimiento del programa) pues sabemos que si luego logramos hacer pasar todas las pruebas tendremos un código que funcione correctamente.

Otra característica de esta práctica es que requiere que el programador primero haga fallar los casos de prueba. La idea es asegurarse de que los casos de prueba realmente funcionen y puedan recoger un error.

7.3.2. Ciclo de desarrollo conducido por pruebas



En primer lugar se debe definir una lista de requisitos y después se ejecuta el siguiente ciclo:

1) Elegir un requisito

Se elige de una lista el requerimiento el que se cree que nos dará mayor conocimiento del problema y que a la vez sea fácil de implementar.

2) Escribir una prueba

Se comienza escribiendo una prueba para el requisito. Para ello el programador debe entender claramente las especificaciones y los requisitos de la funcionalidad que está por implementar. Este paso fuerza al programador a tomar la perspectiva de un cliente considerando el código a través de sus interfaces.

3) Verificar que la prueba falla

Si la prueba no falla es porque el requerimiento ya estaba implementado o porque la prueba es errónea.

4) Desarrollar la implementación

Escribir el código más sencillo que haga que la prueba funcione.

5) Ejecutar las pruebas automatizadas

Verificar si todo el conjunto de pruebas funciona correctamente.

6) Eliminación de duplicación

El paso final es la refactorización, que se utilizará principalmente para eliminar código duplicado. Se hacen de a una vez un pequeño cambio y luego se corren las pruebas hasta que funcionen.

7) Actualización de la lista de requisitos

Se actualiza la lista de requisitos tachando el requisito implementado. Asimismo se agregan requisitos que se hayan visto como necesarios durante este ciclo y se agregan requerimientos de diseño.

7.3.3. Beneficios

A pesar de los elevados requisitos iniciales de aplicar esta metodología, el desarrollo guiado por pruebas (TDD) puede proporcionar un gran valor añadido en la creación de software, produciendo aplicaciones de más calidad y en menor tiempo. Ofrece más que una simple validación del cumplimiento de los requisitos, también puede guiar el diseño de un programa. Centrándose en primer lugar en los casos de prueba uno debe imaginarse cómo los clientes utilizarán la funcionalidad. Por lo tanto, al programador solo le importa la interfaz y no la implementación.

El poder del TDD radica en la capacidad de avanzar en pequeños pasos cuando se necesita. Permite que un programador se centre en la tarea actual y la primera meta es, a menudo, hacer que la prueba pase. Inicialmente no se consideran los casos excepcionales y el manejo de errores. Estos, se implementan después de que se haya alcanzado la funcionalidad principal.

Otra ventaja muy importante es que, cuando es utilizada correctamente, se asegura de que todo el código escrito está cubierto por una prueba. Esto puede dar al programador un mayor nivel de confianza en el código.

7.4. Inspección Continua

Esta práctica, la de Inspección Continua, va muy ligada a la Integración Continua. Esta se expande y construye sobre esta práctica, agregando una capa de análisis de calidad en cada iteración.

Mientras la Integración Continua asegura la estabilidad y minimiza el esfuerzo de integración del código fuente en el proyecto, la Inspección Continua rastrea los requerimientos de calidad en un esfuerzo para controlar la calidad del producto final. Para permitir la inspección continua, se requiere la recolección y análisis de datos después de que cada *build* es producido.

La Inspección Continua se basa en la ejecución frecuente de revisiones de código, para detectar puntos de refactorización e incluso potenciales errores. Algunos de los aspectos que podemos monitorear son: nivel de acoplamiento y complejidad del codigo, codigo duplicado, que tan cubierto esta el código con test unitarios, errores potenciales, reglas de codificación, entre muchos otros.

Gracias a la inspección continua conseguiremos que los equipos de trabajo tengan un conocimiento objetivo, unificado y compartido del estado y salud del proyecto. Esta práctica nos permitirá conocer en todo momento la solidez de nuestros cimientos.

7.5. Deuda Técnica (*Technical Debt*)

Todo proyecto de software incurre en un concepto llamado deuda técnica (*Technical Debt*) que es inevitable a pesar de que se pueda reducir con el uso de buenas prácticas.

La metáfora de deuda técnica, desarrollada por el desarrollador *Ward Cunningham* [*Wikipedia - Technical Debt*], explica cómo el proceso de desarrollar de forma "*ágil*" nos hace incurrir en una deuda, que al igual que una deuda financiera, nos obliga al pago de intereses, que se traducen en un esfuerzo extra a realizar en las siguientes iteraciones de desarrollo.

El uso del concepto de deuda técnica es útil a la hora de transmitir a personas menos técnicas los conceptos, estrategias, compromisos y consecuencias de las elecciones y decisiones que se toman a lo largo de un proyecto de desarrollo ágil para poder cumplir con los objetivos trazados.

7.5.1. Gestión de la Deuda técnica

Teniendo en cuenta que las metodologías ágiles se basan en ciclos cortos de desarrollo, entregas rápidas y correcciones continuas es necesario preguntar: ¿cómo se debe gestionar la deuda técnica?

Antes de iniciar el camino de pagar o cancelar la deuda técnica, se debe establecer un mecanismo de seguimiento y gestión adecuado. Cada situación debe ser descrita en una historia de usuario y colocada en un *backlog* de deuda técnica.

La deuda debe ser gestionada por lo que es y no en base a quién o qué la ha creado, razón por la cual es necesario alentar a todos los miembros del equipo a contribuir con el *backlog* de deuda técnica al igual que en el desarrollo de las historias de usuario relacionadas. Es importante resaltar que este *backlog* nunca está completo ya que siempre está cambiando a medida que surgen nuevos desafíos y se pagan deudas previamente adquiridas.

En un entorno *Scrum*, la reunión de revisión de *Sprint* se ajusta como anillo al dedo para gestionar de forma continua la deuda. Otra opción podría ser colocar una versión impresa del *backlog* sobre un tablero *Scrum*, ya que es crucial mantenerlo siempre visible y vigente.

El paso final es implementar una estrategia para reducir el número de actividades presentes en el *backlog* y por tanto pagar la deuda que se ha adquirido. Dicha estrategia debe estar enfocada a la reducción continua de la deuda.

Obviamente surgirán inquietudes sobre los posibles impactos en la productividad o la obtención de resultados menos directos a lo largo de las distintas iteraciones, y es aquí donde la preparación adecuada y el uso de los mecanismos previamente mencionados darán sus frutos.

7.6. Control de Versiones

Los sistemas de control de versiones permiten a grupos de personas trabajar de forma colaborativa en el desarrollo de proyectos, ya sea a través de una red privada o a través de Internet.

Son sistemas que ponen marcas en las diferentes versiones para identificarlas posteriormente, facilitan el trabajo en paralelo de grupos de usuarios, permiten analizar la evolución de los diferentes módulos del proyecto, y mantienen un control detallado sobre los cambios que se han realizado; funciones que son indispensables durante la vida del proyecto.

Los sistemas de control de versiones se basan en mantener todos los archivos del proyecto en un lugar centralizado, normalmente un único servidor, aunque también hay sistemas distribuidos, donde los desarrolladores se conectan y descargan una copia en local del proyecto. Con ella, envían periódicamente los cambios que realizan al servidor y van actualizando su directorio de trabajo que otros usuarios a su vez han ido modificando.

Los sistemas de control de versiones de código están integrados en el proceso de desarrollo de software de muchas empresas. Cualquier empresa que tenga más de un programador trabajando en un mismo proyecto acostumbra a tener un sistema de este tipo, y a medida que crece el número de personas que se involucran en un proyecto, más indispensable se hace un sistema de control de versiones.

7.6.1. Arquitecturas de almacenamiento

Podemos clasificar los sistemas de control de versiones atendiendo a la arquitectura utilizada para el almacenamiento del código:

7.6.1.1. Repositorios Centralizados

Existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o un conjunto de ellos). Se facilitan las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación del responsable. Algunos ejemplos son *CVS* y *Subversion*.

En los sistemas centralizados las versiones vienen identificadas por un número de versión. Sin embargo en los sistemas de control de versiones distribuidos no hay números de versión, ya que cada repositorio tendría sus propios números de revisión dependiendo de los cambios. En lugar de eso cada versión tiene un identificador al que se le puede asociar una etiqueta (*tag*).

7.6.1.2. Repositorios Distribuidos

Cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales. Ejemplos: *Git* y *Mercurial*.

Estos repositorios necesita menos veces estar conectado a la red para hacer operaciones. Esto produce una mayor autonomía y una mayor rapidez. Aunque se caiga el repositorio remoto la gente puede seguir trabajando.

Al hacer los distintos repositorio una réplica local de la información de los repositorios remotos a los que se conectan, la información está muy replicada y por tanto el sistema tiene menos problemas en recuperarse si por ejemplo se quema la máquina que tiene el repositorio remoto.

Permiten mantener repositorios centrales más limpios en el sentido de que un usuario puede decidir que ciertos cambios realizados por él en el repositorio local, no son relevantes para el resto de usuarios y por tanto no permite que esa información sea accesible de forma pública.

El servidor remoto requiere menos recursos que los que necesitaría un servidor centralizado ya que gran parte del trabajo lo realizan los repositorios locales.

7.7. Gestión de Artefactos

Durante el proceso de construcción de software es necesario realizar tareas como: compilar, generar un ejecutable o librería, generar la documentación del proyecto, etc. Estas acciones se pueden automatizar mediante el uso de herramientas específicas que se detallarán más adelante.

Durante este proceso, además, pueden necesitarse artefactos externos para cumplir las tareas especificadas, y se plantean diferentes opciones sobre qué hacer con estos artefactos. Existe la opción de almacenarlos junto al proyecto, de modo que la compilación sea rápida y sin dependencias externas (ver figura 1).
Sin embargo esto exige la duplicación descontrolada de los artefactos entre diferentes módulos o proyectos, con el consiguiente perjuicio en cuestiones de espacio y sobre todo de organización, por ejemplo al publicarse una nueva versión de un artefacto.

La otra opción, que estén siempre disponibles en Internet en sus páginas de origen (o incluso en un servidor externo), es una solución que tampoco es del todo adecuada, pues introduce un factor de riesgo en el proceso: la compilación será una tarea de tiempo variable y muy dependiente del ancho de banda disponible (ver figura 2).

Para evitar las desventajas de uno u otro método, se introduce una solución intermedia, que es la de construir un repositorio de artefactos, que contendrá una única copia de los artefactos y que podrá estar en un entorno de red local, lo cual mejorará el tiempo de compilación, aparte de otras ventajas que se verán posteriormente (ver figura 3).

Figura 1: Fuentes y artefactos de 3as partes en mismo Repositorio

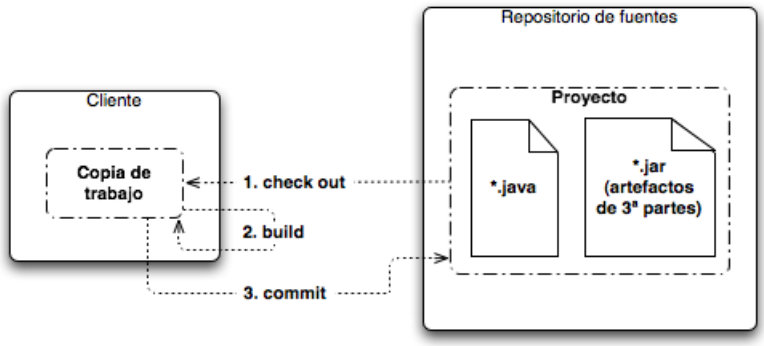
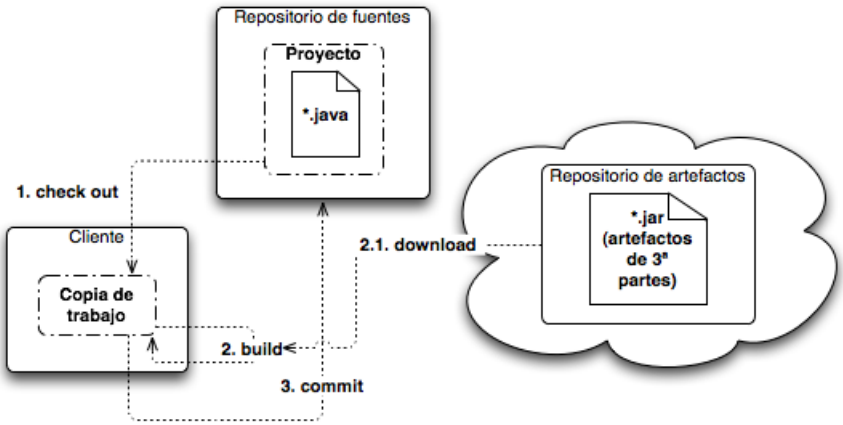


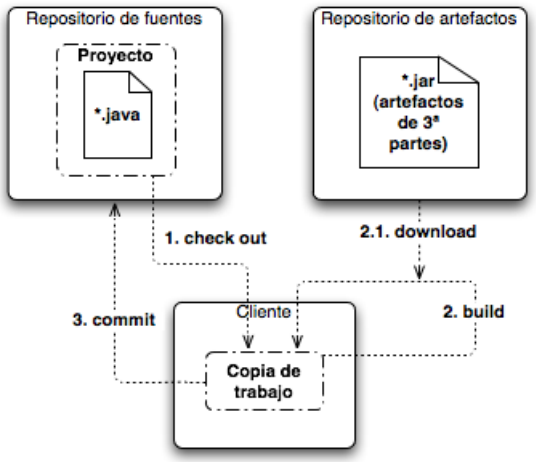
Figura 2: Fuentes en servidor local y artefactos de 3as partes en Internet



7.8. Gestión de Dependencias

Se puede pensar que en proyectos pequeños o de poca complejidad, las dependencias entre distintos componentes son fácilmente manejables mediante una buena documentación del código. El modelo de desarrollo de software libre, por propia necesidad, ha implantado una manera de hacer y de trabajar en la que el control de las dependencias es clave.

Figura 2.3: Fuentes en servidores locales separados



La comunidad Java rápidamente vio esta necesidad y de ahí surgieron las primeras herramientas de gestión de dependencias como lo fue el proyecto *Maven* [*http://maven.apache.org/*].

La gestión de dependencias es un elemento clave y aunque el proyecto se considere mínimo es una buena práctica incorporar técnicas que contemplen esta problemática en las fases tempranas de desarrollo ya que en ocasiones es difícil prever el crecimiento de un proyecto.

7.9. Gestión de Despliegue

En este trabajo se considera despliegue (*Deployment*) al proceso de instalar y actualizar componentes software en el contenedor de componentes. En un sentido más amplio y dependiendo del contexto se puede considerar como parte del proceso de despliegue también la generación de los artefactos y su mantenimiento, pero se ha preferido establecer distinción entre estos dos conceptos como procesos separados porque el modelo resultante aporta más claridad al conjunto y más flexibilidad a la hora de seleccionar herramientas o patrones.

Los artefactos java por ejemplo pueden tener distintos acabados o formatos: *jar, war, ear*, etc. Cuando un artefacto o conjunto de artefactos se considera que debe ser desplegado en el entorno de pruebas o el de producción, lo deseable es que dependiendo del propio artefacto y del destino elegido se conozca con detalle qué acciones deben realizarse, como pasos previos y/o pasos posteriores a su puesta en funcionamiento o activación. No será lo mismo si se despliega en un servidor o en varios. Si se trata de producción, posiblemente habría que activar unas tareas de comprobación más exhaustivas, emitir alarmas, prever mecanismos para volver rápidamente a la última versión estable en caso de falla. La posibilidad de aplicar algún tipo de automatismo o planificación también son funcionalidades que pueden resultar útiles.

8. Relevamiento

8.1. El equipo objeto de estudio

Como ya se ha introducido, la necesidad de este estudio nace de la observación y experiencia adquirida en los diferentes trabajos que nosotros hemos tenido a lo largo de nuestras carreras, pero en particular de la experiencia adquirida al trabajar con un equipo de desarrollo en particular, desde ahora en más: equipo bajo estudio.

El equipo bajo estudio forma parte de una Software Factory terciarizada cuyo cliente pertenece a la industria de las telecomunicaciones y telefonía móvil con sedes en varios países. El equipo se encuentra distribuido en dos locaciones distintas, Córdoba y Buenos Aires.

El equipo bajo estudio es el encargado de crear los nuevos sistemas, servicios o paquetes a partir de los requerimientos de negocio que el cliente envía en forma de "Historia de Usuario" (*User Story*). El diseño e implementación, como la elección de

herramientas y librerías de terceros es en la mayoría de los casos decidido por el propio equipo, con el líder técnico a la cabeza.

El proceso de desarrollo del equipo es *Scrum*, de la familia de procesos agiles.

Las únicas limitaciones técnicas que existen es que se utiliza Java como lenguaje de implementación y salvo excepciones el conjunto de herramientas y librerías a utilizar deben ser de libre uso y sin cargo.

8.2. Problemas detectados

En el trabajo diario de este equipo hemos detectado varios problemas que son comunes a todos los proyectos que emprende. Estos problemas son la base que justifica nuestro proyecto de grado, ellos son:

- *Excesivo esfuerzo de integración de código luego que dos o más proyectos convergen,*
- *Compilación del código a entregar en ambientes no controlados,*
- *La mayoría del código no está cubierto por pruebas unitarias (pobre cobertura del código),*
- *Mayor detección de errores en entorno de producción (Detección tardía de errores),*
- *Código de pruebas unitarias y de integración que quedan obsoletos,*
- *Falta de gestión de artefactos de terceros u de otros proyectos (Gestión de dependencia pobre)*
- *Falta de confianza en la robustez del código,*
- *Inexistencia de indicadores de calidad del código.*

A continuación exponemos que implica cada problema y sus causas.

8.2.1. *Problemas de Integración*

Hemos identificado un excesivo esfuerzo de integración de código luego que dos o más proyectos convergen, e incluso en el simple acto de entregar el código a la línea base del controlador de versiones cuando trabajan dos o más desarrolladores en la misma funcionalidad o módulo.

Por este mismo motivo los integrantes del equipo bajo estudio experimentan cierto temor a integrar sus trabajos, lo que los lleva a retrasar todo lo posible el proceso de integración. Haciendo esto no solo no consiguen evitar lo inevitable (tener que integrar), sino que el número de problemas que se encuentran son mayores conforme pasa el tiempo.

La causa principal es la falta de hábito del equipo en entregar el código regularmente al sistema de versionado. Es normal, en este equipo, que un desarrollador entregue su código quizás cada una semana o más.

Otra causa es la creación de múltiples ramas de desarrollo dentro del controlador de versiones que luego deben converger en la línea base. Algunas de esas ramas pueden llegar a tener meses de desarrollo.

8.2.2. *Construcción en ambientes no controlados*

El método que sigue el equipo para construir el código es el siguiente: El líder técnico del equipo (u en ocasiones otro desarrollador) actualiza el código en su entorno bajando todos los cambios del repositorio de código y compila el código desde su entorno de desarrollo. Una vez hecho esto, se resguarda el producto compilado en el sistema de archivo compartido por el equipo, y se despliega manualmente en el entorno que corresponde, en el entorno de pruebas para el acceso al equipo de QA o en el entorno de pre-producción para el acceso del cliente.

Cabe destacar que nada ni nadie puede asegurar que el mismo producto es instalado en los diferentes ambientes, e incluso nada ni nadie puede asegurar que el producto puede ser repetible (volverse a construir idéntico desde sus archivos fuentes) ya que *el entregable es permeable a la configuración del entorno en el cual se construyó*, entorno que puede cambiar sin control alguno.

Son varios los factores que pueden afectar un producto compilado, desde el sistema operativo instalado en el entorno, pasando por la versión de la Máquina Virtual Java (JVM) utilizada, y hasta las versiones de las librerías de terceros que se utilizan para compilar el código.

8.2.3. *Pobre cobertura del código*

La mayoría del código no está cubierto por pruebas unitarias ni por pruebas de integración, ni siquiera para los procesos más significativos que cubre el proyecto en cuestión.

Si bien realizar pruebas unitarias es un requerimiento de la Software Factory, no hay una forma fácil de comprobar que esto se cumpla, no existen en el equipo indicadores que demuestren en que porcentaje el código está cubierto por pruebas unitarias.

Otras de las causas que descubrimos es que los desarrolladores no ven que la inversión de tiempo en generar pruebas unitarias de sus frutos, y con razón, ya que en el mejor de los casos, las pruebas una vez construidas solo son ejecutadas en un par de ocasiones por el desarrollador que las creo, ya que la ejecución no está automatizada.

8.2.4. *Detección tardía de errores*

Verificamos que la mayor detección de errores se realiza en el entorno de producción, es decir, una vez que el proyecto termino y está a disposición de los usuarios u otros sistemas.

Si bien existe también una falla en el área de QA de la Software Factory, ya que al menos deberían identificar ellos más errores que los que identifica el usuario final, la misma esta fuera del alcance de este trabajo. Aun así, creemos que un alto porcentaje de los errores encontrados en producción pudieron ser evitados en etapa de desarrollo con una buena política de pruebas.

Por supuesto, la causa es la falta de pruebas en tiempo y forma. Si bien el equipo confecciona algunas pruebas de unidad, no son suficientes y no se utilizan más que una o dos veces, ya que no están automatizadas.

La propia experiencia, y el consenso general, nos dice que el coste de corrección de un error se dispara conforme nos alejamos del momento en que se ha producido. Esto es así porque un error arrastra tras de sí otros trabajos que correctos o no, se verán alterados por la corrección del mismo. Por ejemplo, una mala especificación de un requisito que llega a producción tendrá repercusión en el producto final, pero también su corrección requerirá un esfuerzo considerable por los diferentes artefactos que se tienen que modificar o incluso rehacer.

8.2.5. *Pruebas obsoletas*

Las pruebas de unidad e integración deben ser actualizadas según lo demande el cambio de lógica que se de en un proyecto, y esto no se está haciendo en este equipo de trabajo, lo que se traduce en pruebas de unidad que quedaron obsoletas, y por lo tanto, significaron un desperdicio de tiempo.

La causa es la falta de automatización de las pruebas, automatización que obliga a los desarrolladores a mantener actualizada su batería de pruebas, porque de lo contrario el código no pasaría las pruebas y fallaría la construcción. Esta automatización es lo que le da el real valor que puede tener disponer de una batería apropiada de pruebas.

8.2.6. *Gestión de dependencia pobre*

La gestión de dependencias del equipo bajo estudio es informal e ineficiente. Con dependencias nos referimos a artefactos o librerías de terceros o de otros proyectos.

La forma de proceder de este equipo consistía en delegar a una herramienta de gestión de dependencias (*Maven*) la resolución de las mismas. Lo que no es malo, solo que la configuración no es la adecuada en este caso, ya que la herramienta se limita a buscar las últimas versiones de las dependencias que se informan en un archivo de configuración (*POM*) en repositorios publicos a travéz de internet, pero a su vez estas

dependencias dependen de otras, lo que dificulta el control de las librerías que se interactúan con el proyecto.

Esta herramienta busca por defecto las dependencias en repositorios públicos de Internet, imposibilitando el control por parte de la organización en términos de licencias, versiones e inventario. A la vez que cada desarrollador tiene que descargar las librerías directamente de internet, en lugar de definir un repositorio dentro de los límites de su propia red, haciendo un ineficiente uso de los recursos de la empresa.

8.2.7. *Falta de confianza en el código*

Esto quizás sea una consecuencia a muchos de los problemas detectados, pero quisimos darle el énfasis que se merece, ya que esto impacta negativamente en el equipo, causando desmotivación y hasta incluso mala reputación de la Software Factory.

A menudo las características del producto a entregar bajo este contexto son: un producto único, no repetible, el cual la robustez es discutible y la confiabilidad escaza, debido a la pobre cobertura de pruebas unitarias y de integración.

8.2.8. *Inexistencia de indicadores de calidad*

La calidad del código en el contexto del equipo bajo estudio se basa solamente en la revisión de código por parte del líder técnico quien plasma en una planilla todos los puntos del código que requieren atención y deben ser cambiados. Pero esta revisión manual efectuada por una persona que cumple un rol y que casi nunca tiene el tiempo necesario para hacer un revisión adecuada no es suficiente, y no arroja estadísticas inmediatas que sean útiles y den visibilidad de la calidad del proyecto en cada momento del desarrollo.

Bajo esta realidad, el nivel de calidad del código no es más que una percepción, haciendo imposible imponer metas para mejorarla.

8.3. Entorno de Desarrollo Actual

A continuación exponemos un diagrama que detalla el entorno de desarrollo del equipo bajo estudio, en el cual conviven los problemas identificados.

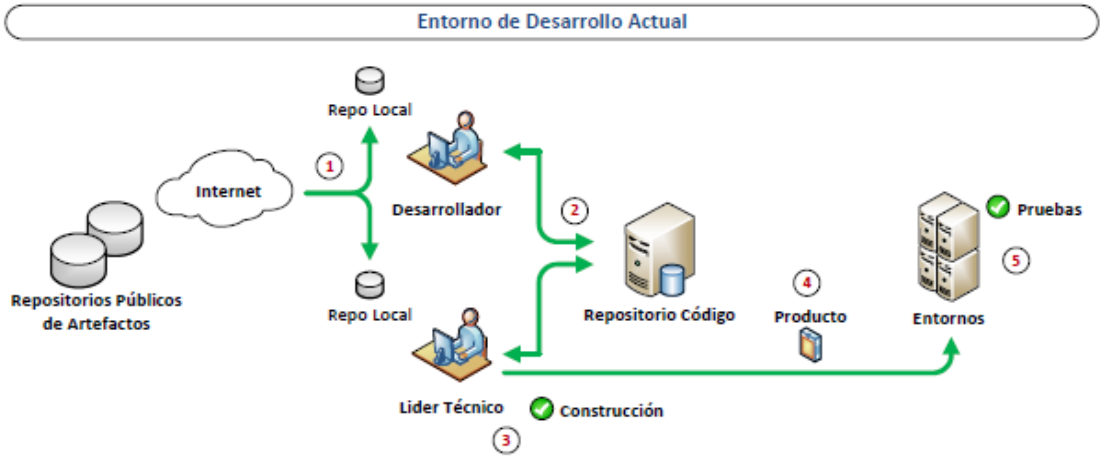


Figura 8.1 – Entorno de Desarrollo Actual

(1) Cada desarrollador debe resolver las dependencias del código por su cuenta, descargándolas directamente desde repositorios públicos en la Internet a través de *Maven*.
(2) Cada desarrollador sube sus cambios (*commit*) al repositorio de código sin control alguno de que sea código que compile o cumpla su cometido.
(3) Actualmente es un Líder Técnico el que compila en su ambiente personal (ambiente no controlado) la versión del código a entregar, siendo este entregable permeable de la configuración de su entorno, el cual puede cambiar sin control alguno.
(4) El producto a entregar es, en este entorno, un producto único, no repetible, el cual la robustez es discutible y la confiabilidad escaza, debido a la pobre cobertura de pruebas unitarias y de integración.
(5) Cuando este producto es entregado a la siguiente etapa del proceso (QA), se debe desplegar en entornos de pruebas. Este despliegue se hace de forma manual y desprolija y a menudo tardía, dejando entre etapa y etapa un tiempo ocioso muy costoso. Y una vez hecho las pruebas del mismo son únicamente manuales, agregando más tiempo sin valor agregado al proceso.

8.4. Herramientas utilizadas en el Entorno de Desarrollo actual

A continuación haremos un relevamiento de las herramientas, lenguajes y frameworks más utilizados por el equipo de desarrollo bajo estudio.

Algunas de estas herramientas puede tener un papel importante en nuestra solución y otras pueden tener que interactuar con otras herramientas propuestas en la solución de los problemas planteados.

No necesitamos en este punto mayores detalles, solo dejar claro la funcionalidad de cada una de estas herramientas y cuales tendríamos la potestad de cambiar a efectos de dar una solución y cuales son impuestas por el cliente o por la Software Factory.

8.4.1. *Herramientas de Gestión*

Nuestro equipo bajo estudio desarrolla soluciones bajo una de las metodologías ágiles más conocidas, *Scrum.* El desarrollo de la citada metodología queda fuera del alcance de este trabajo, pero daremos una pequeña referencia a la misma:

[*Wikipedia - Scrum*]:

"*Scrum es un modelo de desarrollo ágil caracterizado por:*

- *Adoptar una estrategia de desarrollo incremental, en lugar de la planificación y ejecución completa del producto.*
- *Basar la calidad del resultado más en el conocimiento tácito de las personas en equipos auto-organizados, que en la calidad de los procesos empleados.*
- *Solapamiento de las diferentes fases del desarrollo, en lugar de realizar una tras otra en un ciclo secuencial o de cascada*."

La herramienta que utiliza el equipo bajo estudio, y el resto de la Software Factory, para gestionar su proceso basado en Scrum es **Rally** [*Rally Software*]. Este sistema web expone todos los requerimientos del cliente como "historias de usuario" y permite llevar su seguimiento fácilmente.

Para el seguimiento y administración de errores u otras acciones se dispone de otro sistema web muy conocido: **Jira** [*Atlassian - Jira*].

Ambas herramientas son de pago e impuestas por el cliente y por lo tanto no tenemos la potestad de cambiarlas.

8.4.2. *Entornos Integrados de Desarrollo (IDEs)*

El principal Entorno Integrado de Desarrollo utilizado es **Eclipse** [*Eclipse- IDE*]. Es el IDE más conocido y utilizado para el desarrollo de aplicaciones basadas en el lenguaje Java.

Para la interacción con la base de datos se utiliza **SQLDeveloper** [*Oracle-SQLDeveloper*], la opción de libre uso de facto para manipular base de datos Oracle.

Otra herramienta muy útil que se utiliza es **SoapUI** [*Smart Bear - SoapUI*], la misma es usada por el equipo para hacer pruebas a los distintos servicios web con los que interaccionan y también para probar los que ellos mismos crean.

Son todas herramientas de libre uso y seleccionadas por el equipo, por lo tanto podríamos usar otras que cumplan su función en la solución si así lo necesitáramos.

8.4.3. *Herramientas para la construcción y gestión de dependencias*

Para la gestión, construcción y control de dependencias de los proyectos del equipo se utiliza una herramienta de software conocida como **Maven** [*Apache Maven*].

Es en verdad una herramienta muy útil y versátil, y tendrá un papel muy importante en nuestra solución para solucionar algunos de los problemas encontrados.

Esta herramienta fue seleccionada por el equipo, y también podríamos cambiarla por otra que cubra sus capacidades a efectos de una mejor solución.

8.4.4. *Sistema de Control de Versiones*

El sistema de control de versiones utilizado por el equipo en **SVN** [*Apache Subversion*], un sistema de control de versiones centralizado muy conocido y ampliamente utilizado.

SVN es el estándar que utiliza el cliente para todos sus proyectos actualmente, por lo tanto no tenemos la opción de reemplazarlo por otro, *aun cuando creamos que **Git** sería mucho más adecuado a la topología del equipo distribuido.*

8.4.5. *Motores de Base de Datos*

El Cliente utiliza para todos sus proyectos la base de datos **Oracle 11g** [*Oracle DB*], por tal motivo tampoco nos es posible pensar en alguna alternativa.

8.4.6. *Servidor de Aplicaciones*

El Servidor de Aplicaciones usado para la mayoría de los desarrollos en la Software Factory es **Tomcat** [*Apache Tomcat*], servidor ampliamente utilizado y de libre uso. Tampoco tenemos mucho margen de maniobra aquí.

8.4.7. *Lenguajes de programación y Frameworks*

El lenguaje de programación principal en toda la Software Factory es **Java** [*Oracle Java*], y por lo tanto la gran mayoría de los desarrolladores son desarrolladores Java.

Un Framework muy utilizado por el equipo de desarrollo es **Spring Framework** [*Spring Framework*]. El mismo permite simplificar drásticamente el desarrollo de aplicaciones Java sin ser intrusivo.

9. Diagnóstico y Conclusiones

En la etapa de relevamiento descubrimos los siguientes problemas (síntomas):

- *Excesivo esfuerzo de integración de código luego que dos o más proyectos convergen,*
- *Compilación del código a entregar en ambientes no controlados,*
- *La mayoría del código no está cubierto por pruebas unitarias (pobre cobertura del código),*
- *Mayor detección de errores en entorno de producción (Detección tardía de errores),*
- *Código de pruebas unitarias y de integración que quedan obsoletos,*
- *Falta de gestión de artefactos de terceros u de otros proyectos (Gestión de dependencia pobre)*
- *Falta de confianza en la robustez del código,*
- *Inexistencia de indicadores de calidad del código.*

Hemos visto hasta aquí como se desenvuelve el equipo bajo estudio y cuál es su ámbito, es decir, su entorno de desarrollo. También hemos identificado los síntomas que padece. Los problemas que presenta este equipo tienden a ser más profundos y difíciles de revertir conforme pasa el tiempo.

El hecho que el equipo este distribuido en dos diferentes ciudades agrega otro condimento, que profundiza aún más estos problemas.

El entorno de desarrollo de un equipo distribuido que se autodenomina "ágil" (por utilizar una metodología de desarrollo ágil) necesita apoyarse en una práctica que complemente ese proceso de desarrollo utilizado, y claramente el entorno de desarrollo actual no corresponde con una metodología ágil. Quizás sea suficiente para una clásica metodología en cascada, pero no lo resulta para una metodología en donde se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto, una metodología donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos, donde la innovación, la competitividad, la flexibilidad y la productividad son fundamentales.

Creemos que para solucionar todos y cada uno de estos problemas debería haber un cambio significativo en el entorno de desarrollo actual, el entorno debería al menos:

- Ayudar en el proceso de Integración, para hacerlo iterativo e incremental.
- Automatizar la construcción y el despliegue del código
- Administrar la gestión de dependencia y de artefactos propios
- Alentar a la creación y actualización de pruebas de unidad / integración
- Automatizar la pruebas unitarias y de integración
- Proveer visibilidad en todo momento del nivel de calidad del código en varios aspectos.

De todo este análisis hemos podido identificar tres focos a los cuales debemos atacar para revertir los problemas identificados en el equipo bajo estudio, y ellos son:

- El entorno de desarrollo desactualizado
- La falta de una práctica que incentive la creación de pruebas
- La falta de recolección de métricas de calidad

Para cada uno de estos focos propondremos soluciones, pero primero nos explayaremos un poco más en cada uno de ellos para llegar a entender que implica cada uno de ellos y que problemas derivan de su existencia.

9.1. Entorno de Desarrollo Desactualizado

De los problemas identificados, un subconjunto de ellos los podemos agrupar aquí, siendo ellos consecuencia de no disponer de un Entorno de Desarrollo adecuado y que se corresponda con la metodología ágil que el equipo implementa. Ellos son:

- *Excesivo esfuerzo de integración de código luego que dos o más proyectos convergen,*
- *Compilación del código a entregar en ambientes no controlados,*
- *Falta de gestión de artefactos de terceros u de otros proyectos (Gestión de dependencia pobre)*
- *Falta de confianza en la robustez del código*

Conocemos los beneficios que entrega la práctica de Integración Continua desde nuestra exposición de la misma en el capítulo siete, por lo que a esta altura ya no es un secreto, este equipo necesita empezar a disponer de esas bondades cuanto antes. Estamos seguro que al aplicar esta práctica, el equipo resolvería al menos los problemas agrupados aquí.

En la sección 7.1.2, principios de la Integración Continua, dentro del marco teórico de este trabajo, expusimos los diez principios que se deber aplicar para lograr los beneficios que brinda esta práctica. Ahora, por cada uno de esos principios, veremos cómo está parado nuestro equipo de desarrollo bajo estudio:

1) Mantener un único repositorio de código fuente

El equipo hace uso, quizás excesivo, de la creación de diferentes ramas de desarrollo en el repositorio de código para soportar los diferentes proyectos y mejoras. Por lo tanto hay una oportunidad de mejora aquí.

2) Automatizar la construcción del proyecto

La automatización en la construcción ya lo están logrando en parte al utilizar la herramienta de construcción *Maven*, pero la construcción se compone también de la gestión de dependencias, que si bien *Maven* también lo resuelve, la manera por defecto en cual lo hace trae aparejado uno de los problemas detectados, que es la pobre gestión de dependencia.

3) Autodiagnóstico de la construcción

Una vez construido el código no se dispone de ningún proceso que ejecute los casos de prueba automáticamente, por lo que el principio de autodiagnóstico no se cumple. Si bien se dispone de algunas pruebas de unitarias, como ya dijimos, las mismas pueden estar obsoletas porque no se utilizan a menudo ni se mantienen.

El único diagnostico que este equipo puede tener de una construcción de código es la prueba manual de sus características principales.

4) Entregar los cambios diariamente a la línea base

El equipo no actúa de esta manera, incluso pueden pasar semanas antes que un miembro entregue sus cambios a la línea base del repositorio. Aún peor, este equipo suele crear múltiples ramas en el repositorio de código para mantener distintos proyectos o mejoras del mismo producto de software.

5) Construir la línea base tras cada entrega

La línea base es construida en pocas ocasiones, quizás luego de que el largo proceso de integración de las distintas ramas se haya realizado y no cuando cada miembro del equipo entrega código a la línea base o almenos diariamente.

6) Mantener una ejecución rápida de la construcción del proyecto

Si bien con la herramienta Maven pueden mantener una velocidad aceptable para la construcción del código, el equipo debe realizar antes unos pasos manuales para editar algunos ficheros de configuración, configuración que cambia para cada ambiente. Esto si agrega tiempo considerable a la construcción del código.

7) Probar en una réplica del entorno de producción

El equipo bajo estudio no dispone de un ambiente similar al de producción para realizar las pruebas, y las mismas son hechas localmente en el ambiente de cada desarrollador antes de entregar el producto al equipo de QA o al cliente. Estas pruebas son en su mayoría manuales, por no disponer de una batería actualizada y medianamente abarcativa del código.

8) Fácil obtención del último ejecutable del proyecto

Los ejecutables son almacenados en un sistema de archivo perteneciente al cliente, donde existe una estructura de carpetas para cada sistema, donde se divide por fecha y versión. Este sistema es poco eficiente y permeable al error humano, cualquiera puede cambiar la versión o ubicación de un ejecutable.

La obtención de un determinado ejecutable por parte del equipo de desarrollo o del equipo de QA o incluso del cliente es tan complejo como navegar un sistema de archivo y buscar la versión deseada.

9) Publicar el estado del proyecto

No es conocido en todo momento el estado de cada uno de los proyectos. Solo cuando alguien construye el código correspondiente a un proyecto puede saber si el mismo compila sin problemas o no.

10) Automatizar el despliegue

El despliegue de los ejecutables en los entornos de desarrollo y QA es responsabilidad del equipo de desarrollo, y vemos que el mismo se realiza manualmente. Primero se ubica la versión y luego se despliegue desde la consola del servidor web. El proceso no solo es manual, si no que carece de políticas de despliegue adecuadas.

Este diagnóstico con foco en los principios de la IC nos ayudará a encontrar la mejor implementación de esta práctica para el equipo. Propondremos una solución tangible más adelante, en la sección de “Propuesta”.

9.2. Falta de una Práctica que Incentive la Creación de Pruebas

Al igual que en la sección anterior, un subconjunto de los problemas identificados los podemos agrupar aquí, siendo ellos consecuencia de la falta de una práctica correcta de creación de pruebas de unidad y de integración. Ellos son:

- *La mayoría del código no está cubierto por pruebas unitarias (pobre cobertura del código),*
- *Mayor detección de errores en entorno de producción (Detección tardía de errores),*
- *Código de pruebas unitarias y de integración que quedan obsoletos,*
- *Falta de confianza en la robustez del código,*

La mayoría de ellos se deben a la falta de una batería de pruebas unitarias y/o de integración lo suficientemente abarcativa. Para gozar de los beneficios de la práctica de Integración Continúa es imperativo revertir esta situación.

Recordamos que la falta de pruebas unitarias se debía, aparte de por la falta de disciplina, a que los desarrolladores no veían que el esfuerzo hecho en desarrollar esas pruebas dieran sus frutos, ya que las pruebas eran ejecutadas en un par de ocasiones y luego si los requerimientos cambiaban, o incluso si el código de las pruebas dejaba de compilar simplemente se excluían y quedaban obsoletas.

Al aplicar los principios de la IC, esta situación debería cambiar, y los desarrolladores van a poder ver que el esfuerzo que invierten en desarrollar estas pruebas va a ser recompensado, ya que las mismas se ejecutarán automáticamente una infinidad de veces, determinando si un *build* (construcción de código) falla o no, obligándolos a mantenerlas siempre actualizadas y por ende, muy útiles.

Aun así, el proceso de desarrollo debería de alguna forma poner en foco la creación de estas pruebas de unidad y/o integración, de modo de transformar esta acción en una sana y necesaria disciplina.

Identificaremos una solución aquí también, la misma será desarrollada en la sección de “Propuesta”.

9.3. Falta de Recolección de Métricas de Calidad

Un par de los problemas identificados son consecuencia de la falta de recolección de métricas o indicadores de calidad del código. Ellos son:

- *Inexistencia de indicadores de calidad del código,*
- *Falta de confianza en la robustez del código.*

El monitoreo constante de la calidad del software en un entorno de desarrollo bajo una metodología ágil es muy importante, ya que los equipos que actúan de esta forma, normalmente para atajar los tiempos de entrega adquieren una "*deuda técnica*", cuyos "*intereses*" se tendrán que acabar pagando, y supondrán mayor cuantía cuanto más tiempo pase.

Esta deuda técnica (*technical debt*) no es más que la deuda de trabajo que se adquiere al producir código pobre, incumpliendo prácticas aconsejadas para el desarrollo de software. Este concepto está desarrollado en la sección 7 (Marco Teórico).

Por este motivo es muy importante mantener a raya esta deuda y para eso necesitamos monitorearla. El equipo bajo estudio solo se basa en la revisión de código por parte del líder técnico quien plasma en una planilla todos los puntos del código que requieren atención y deben ser cambiados. Pero esta revisión manual efectuada por una persona que cumple un rol y que casi nunca tiene el tiempo necesario para hacer un

revisión adecuada no es suficiente, y no arroja estadísticas inmediatas que sean útiles y den visibilidad de la calidad del proyecto en cada momento del desarrollo.

Este equipo necesita un monitoreo permanente de estos aspectos de la calidad, y sobre todo necesita comenzar a dar visibilidad a la deuda tecnica para poder empezar a pagar los intereses de su deuda.

Propondremos una práctica que solucione este problema en la sección de "Propuesta" a continuación.

10. Propuesta

Creemos que se pueden solucionar todos y cada uno de los problemas enunciados que afectan al equipo bajo estudio a un bajo coste, haciendo uso de prácticas ya probadas, y apoyándose en herramientas sólidas y de libre uso.

La solución por supuesto se basará en la práctica de Integración Continua como estandarte. Utilizaremos también una metodología de desarrollo muy difundida en la actualidad para nutrir de pruebas al entorno de desarrollo, esta metodología es la de *Desarrollo Guiado por Pruebas* (*TDD*). Y para completar, dotaremos al equipo de la capacidad de obtener toda la información sobre la calidad del código que necesitan y en todo momento, gracias a la aplicación de la práctica de Inspección Continua.

Expondremos primero las soluciones aisladamente, para luego demostrar cómo funcionan de manera mancomunada, retroalimentándose unas de otras, formando un ecosistema apto para el desarrollo ágil de soluciones de software.

10.1. Aplicar Integración Continua

La mejor forma de aplicar la Integración Continua a un entorno de desarrollo es analizando cada principio de la práctica, como ya lo hicimos, y cambiar lo que no estamos haciendo bien. Trataremos de enfocarnos en acciones concretas para llegar al cambio que necesitamos. El propósito de cada acción será, por consiguiente, transformar el actual entorno de desarrollo para cumplir con uno o más de los principios de la Integración Continua.

10.1.1. Cambiar la Estrategia de Control de Versiones

El primer cambio que este equipo debe realizar, es la forma en que utiliza el repositorio de código. Debe pasar de utilizar una estrategia de desarrollo en paralelo o en ramas a una estrategia lineal. Con estrategia lineal nos referimos a que todos los desarrolladores comprometen su código (*commit*) a la línea base (*trunk*) como regla. Con este sencillo acto obligamos al equipo a integrar frecuentemente su código.

Otro hábito que el equipo debe cambiar es la frecuencia en la cual ellos comprometen su código. Es muy importante que lo hagan diariamente o lo más cercano a ello para encontrar las posibles fallas cuanto antes y así el esfuerzo para solucionarlas será menor.

Ambas premisas, utilizar una estrategia lineal y comprometer el código diariamente deben convertirse en una política para este equipo y deben ser comunicada eficazmente.

Con esta nueva política hacemos frente a dos de los diez principios de la IC, "*Mantener un único repositorio de código fuente*" y "*Entregar los cambios diariamente a la línea base*".

10.1.2. Utilizar un Servidor de Integración Continua

A lo largo de todo el trabajo hemos mencionado las bondades de esta práctica, documentada en la *sección 7.1*, pero para materializar las mismas necesitamos apoyarnos en herramientas especializadas. Empezaremos con la más importante de todas, el servidor de Integración Continua (IC).

Instalar un Servidor de Integración Continua sería el segundo paso a tomar, si vemos el proceso de una manera secuencial.

10.1.2.1. Elección del Servidor de IC

Uno de los lineamientos de este trabajo es el uso de herramientas *open source* o de libre uso, por lo cual, para hacer la elección de un Servidor de IC comparamos las tres opciones libres más populares al momento de realizar este trabajo, y a priori, las tres opciones más viables. Ellas son: ***CruiseControl***, ***Jenkins*** y ***Continuum***. Los detalles de esta comparación se podrán ver en el *ANEXO A: Evaluación Comparativa de Servidores de IC.*

De esta comparación surgió como mejor alternativa el servidor de Integración Continua ***Jenkins***, que será nuestro Servidor de IC de aquí en más a lo largo de nuestro trabajo.

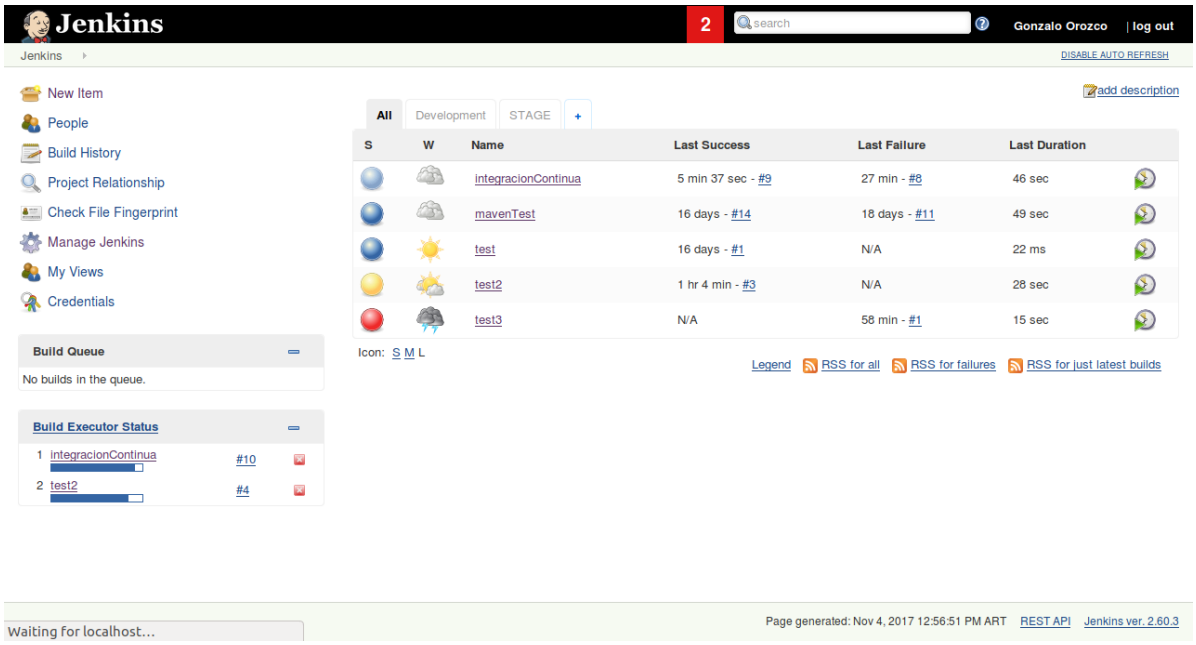
Jenkins es una premiada aplicación que supervisa ejecuciones de trabajos repetitivos, como la construcción de un proyecto de software o trabajos ejecutados por un *Cron* (programa informático que realiza tareas a intervalos regulares).

Entre otras cosas, Jenkins se centra en dos trabajos:

- Construir y Probar proyectos de software continuamente.
- Monitorear la ejecución de trabajos ejecutados externamente.

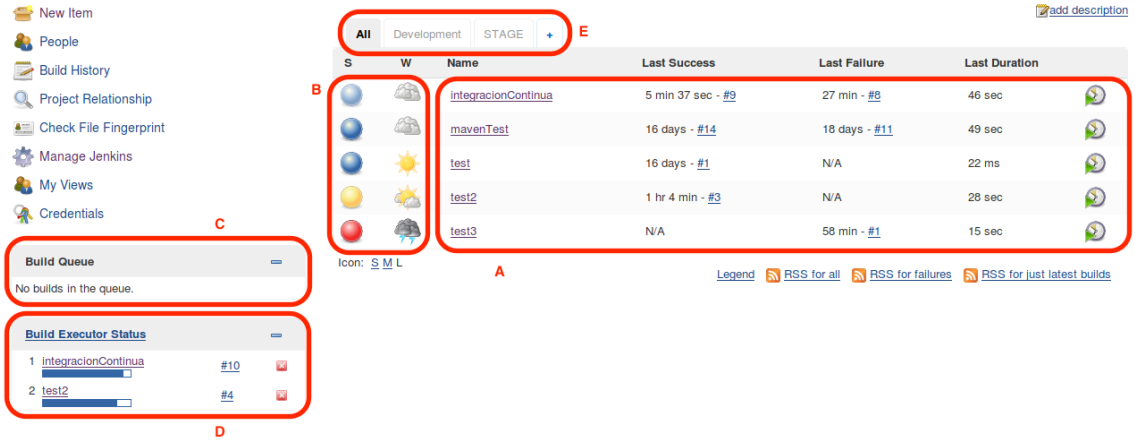
Jenkins dispone de la funcionalidad de extenderse mediante complementos (*plugins)*. Existen multitud de complementos que permiten cambiar el comportamiento de Jenkins o añadir nueva funcionalidad.

Jenkins dispone de una consola web para su administración y gestión de las diferentes tareas que, como podemos ver en la siguiente imagen, tiene una forma muy clara y grafica de mostrar el estado de los diferentes proyectos. Es fácil de usar y hay muchísima documentación disponible en la web.



Captura de Pantalla 10.1: Ejemplo Consola de Jenkins

A continuación explicaremos las distintas secciones de la consola:



Captura de Pantalla 10.2: Explicación Consola de Jenkins

A. Se muestra la lista de tareas (Jobs) que tenemos configuradas, normalmente cada tarea se encarga de construir el codigo fuente, ejecutar los test, ejecutar la inspección del código e incluso hasta desplegar el producto construido.

B. Indicadores visuales de la salud de esa tarea, las esferas de colores indican el estado actual de la ultima ejecución y los indicadores climáticos la estabilidad teniendo en cuenta las últimas ejecuciones.

Estado de la última ejecución:

finalización exitosa sin errores del job (construccion, test, analicis de calidad del codigo, etc)

finalización inestable del job (construccion exitosa del código, pero fallaron las pruebas unitarias o de integracion)

finalización fallida del job (errores de compilación)

Estabilidad del job: Aquí se grafica la estabilidad haciendo un paralelo con el clima, mientras más soleado está el job más estable es. Un icono de sol pleno indica que todas las ultimas ejecuciones an sido exitosas, mientras que en el otro extremo, un icono de tormenta indica que todas las ultimas ejecuciones fallaron.

C. La cola de builds, por aquí pasa cada job que es ejecutado, si hay un build server disponible se ejecuta, de lo contrario espera en la cola su turno.

D. Aquí se muestran los jobs que se estan ejecutando en ese momento y el estado de esa ejecución, habrá tantos jobs aquí como build servers halla disponible.

E. Las tareas pueden organizarse en distintas pestaña para una mejor gestión, por ejemplo, se podría tener una pestaña para las tareas en ambientes de desarrollo, otra para ambiente de Stage o pre-producción, etc.

Agunas de las cosas que Jenkins es capaz de automatizar son:

- Detectar cambios y descargar el código desde el repositorio de versiones.
- Lanzar la construcción del sistema, y la ejecución de las pruebas.
- Ejecutar automáticamente herramientas de análisis de la calidad del código.
- Publicar los artefactos generados en la construcción.
- Etiquetar el código tras una construcción exitosa.
- Ante errores, notificar al equipo de desarrollo de los mismos.

- Gestionar el histórico de construcciones y de estadísticas de pruebas, etc.

10.1.3. Utilizar un Gestor de Repositorio de Artefactos

Como ya indicamos, el equipo bajo estudio utiliza la herramienta *Maven*, que facilita el proceso de gestión y construcción de los proyectos. Entre los aspectos más importantes a destacar de esta herramienta se encuentra la gestión de dependencias que nos permite saber qué librerías y qué versiones son necesarias en el proyecto para ejecutarse.

Maven utiliza una matriz de repositorios remotos que le permite localizar y descargar todo lo necesario para generar nuestro proyecto de forma transparente al desarrollador. Además de los repositorios remotos también existe un repositorio local que lo utiliza como caché evitando la descarga en las siguientes generaciones del proyecto y así reducir el tiempo que supondría volver a descargarse todas las librerías.

En determinados entornos, *Maven* puede ser más que suficiente pero en grandes organizaciones es muy probable que no. La restricción de acceso a Internet, el control de acceso a los repositorios, la exclusión de ciertas librerías, la reducción del consumo del ancho de banda o la administración de los repositorios de la propia organización son aspectos que quedan fuera del alcance de *Maven*. Debido a estas necesidades aparecieron en el mercado los Gestores de Repositorio de Artefactos, necesidades que también tiene nuestro equipo bajo estudio.

10.1.3.1. Elección del Gestor de Repositorio de Artefactos

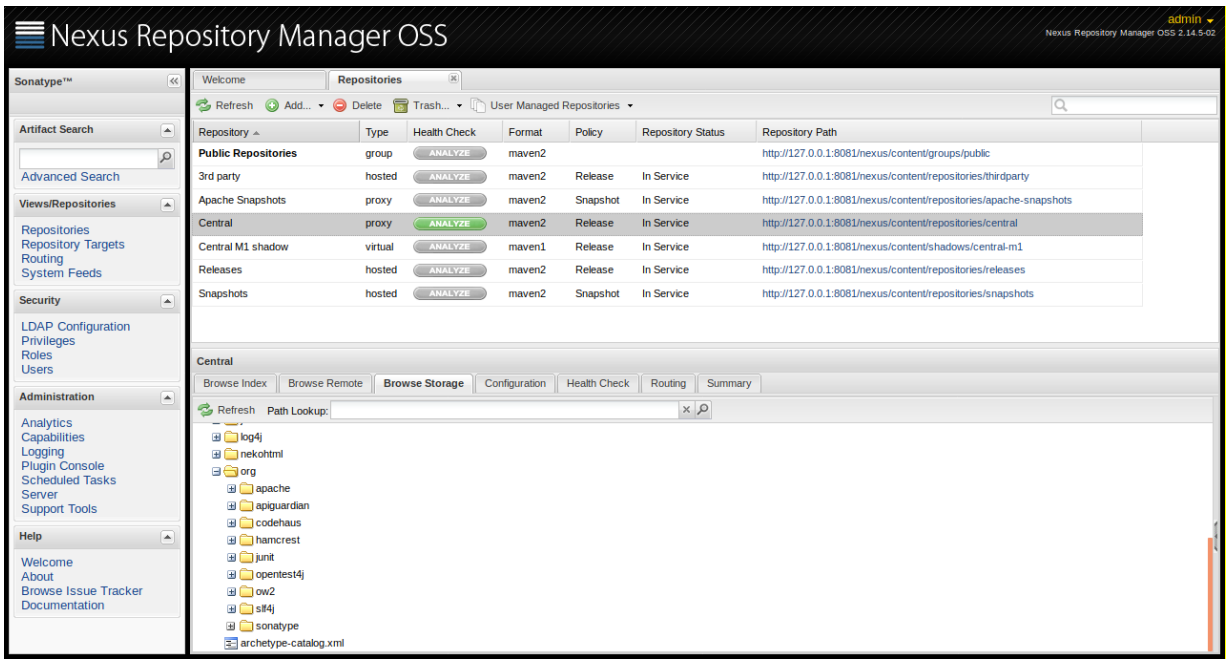
Siguiendo con los lineamientos *open source*, hasta la fecha podemos destacar tres gestores de repositorios *Maven**: Archiva, Artifactory y Nexus***. De la comparación que realizamos, la cual se puede ver en detalle en el *ANEXO B: Evaluación Comparativa de Gestores de Repositorio de Artefactos*, surge como mejor opción el gestor de artefactos **Nexus**, en su formato *Nexus Repository OSS*, que será nuestro gestor de artefactos de referencia en este trabajo.

Nexus es un gestor de repositorios de artefactos de software para Maven (entre otras herramientas) que tiene dos propósitos, actuar como una interfaz altamente configurable entre su organización y los repositorios públicos, permitiendo filtrar determinados artefactos y/o versiones y también proporciona a la organización una nueva manera de compartir sus propios artefactos internamente.

Nexus nos otorga las siguientes ventajas:

- Disponer de un repositorio para los artefactos propios de la organización.
- Facilitar el almacenamiento en caché de repositorios remotos y eliminar la latencia en la cadena de suministro de software.
- Control de versiones y licencia de artefactos de terceros.
- Soportar los principales formatos y tipos de artefactos.

Nexus dispone de una interfaz grafica para la gestion tanto de los repositorios externos como de los internos de la compañía. En ella se pueden definir permisos, excluir dependencias, validar licencias de estas dependencias antes de su uso, etc.



Captura de Pantalla 10.3: Consola de Nexus

Cada proyecto, en cada ambiente de desarrollo, dentro de sus configuraciones, estará resolviendo sus dependencias a través de este gestor de dependencias, incluso el mismo servidor de IC resolverá las dependencias de ese modo, como se puede ver en el *ANEXO C: Configuración Del Nuevo Entorno De Desarrollo Integrado.* Del mismo modo los artefactos creados por el equipo de desarrollo dentro de este nuevo ambiente podrá versionar los mismos dentro de este gestor para que estén disponibles para el resto de la organización.

Más adelante expondremos con más detalle la interrelación de este herramienta con el servidor de IC y las demás herramientas propuestas.

10.2. Adoptar la Metodología de Desarrollo Guiado por Pruebas

Como adelantamos, utilizaremos una metodología de desarrollo muy difundida en la actualidad para nutrir de pruebas al entorno de desarrollo, esta metodología es la de Desarrollo Guiado por Pruebas (TDD). El marco teórico de esta metodología está desarrollado completamente en el punto *7.3. Desarrollo Guiado por Pruebas (TDD)* del presente trabajo, por lo que no nos detendremos nuevamente a explicar cómo funciona y que beneficios nos brinda.

Por supuesto, la forma de aplicar esta metodología depende del conocimiento que se tenga de la misma. Lo que nosotros proponemos es brindar una capacitación a todos los involucrados, sea esta capacitación realizada por alguien perteneciente a la organización o por alguna consultoría externa. Darle un marco formal ayudaría a darle la importancia que merece.

Tanto la preparación, la generación del contenido de la capacitación y la capacitación en sí, quedan fuera del alcance del presente trabajo de grado, pero expondremos a continuación los retos que se podrían tener al adoptar esta metodología y algunos concejos para abordarlos.

10.2.1. Curva de Aprendizaje

Esta técnica requiere de entrenamiento adecuado, situación que es retadora dado que la literatura existente tiende a enfocarse en problemas más fáciles de resolver que los de la vida real. Su implementación requiere de tiempo para experimentar y probar, lo cual puede resultar contraproducente de cara a los compromisos con las áreas de negocio de la organización.

Técnicas propias del TDD, como por ejemplo la refactorización de código legado, aislamiento de pruebas unitarias y aislamiento de pruebas integrales son difíciles de dominar.

El reto de la curva de aprendizaje debe enfrentarse brindando suficiente tiempo al equipo para la adopción, por lo cual debería estar libre de mayor presión para realizar sus entregas. Asimismo, los miembros más experimentados de la organización pueden hacer *mentoreo*, el cual se percibe como más efectivo.

10.2.2. Énfasis en las Pruebas en lugar del Desarrollo

Esta técnica implica un cambio de paradigma en los desarrolladores, quienes deben pasar de tener un foco en la programación de código funcional, a otro en donde se pone mayor énfasis en el diseño de casos de pruebas de las futuras funcionalidades priori a una posterior codificación. De hecho, se estima que el 60% del esfuerzo se consume en diseñar y escribir las pruebas y el restante 40% en el código cuando se trabaja con TDD.

Esto, implica lograr que los desarrolladores puedan repensar la forma en que aprenden, diseñan y trabajan.

Para enfrentar este reto es necesario incorporar desarrolladores con experiencia, que puedan explorar y explicar a los demás los aspectos relacionados con el diseño de software y la refactorización constante. Asimismo, se pueden incorporar analistas de pruebas para que trabajen con los desarrolladores y les transfieran conocimientos en pruebas de software.

10.2.3. Obtener el apoyo de la organización

Cuando se explora un nuevo enfoque como el de TDD, se puede estar tentado a aplicarlo sin informar a la gerencia, pero cuidado, la adopción de TDD puede reducir temporalmente la productividad, ocasionando compromisos incumplidos y un rechazo de la técnica por parte de la organización sin siquiera haberla experimentado.

Es imperativo buscar apoyo de la gerencia, en especial de las áreas de negocio para reprogramar los compromisos de entregas, todos los involucrados en la organización deben estar alineados con esa idea.

Al igual que el equipo de desarrollo, la gerencia debe cambiar paradigmas, como por ejemplo, anti valores establecidos como que definir los casos de prueba antes es una pérdida de tiempo o la constante presión para entregar productos de software rápido sin que los integrantes del equipo puedan tomarse el tiempo necesario para mejorar el código.

10.2.4. Adaptar Código de Sistemas Legados

TDD es difícil de aplicar en situaciones en las cuales se está dando mantenimiento o evolucionando sistemas legados de alta complejidad. Está demostrado que *refactorizar* código legado para que pueda probarse adecuadamente toma más tiempo y esfuerzo.

Por lo general estas bases de códigos tienen décadas en la organización, el saber hacer (*know how*) está concentrado en pocas personas, muchos de los cuales hace tiempo dejaron de trabajar en la compañía y existe documentación insuficiente. Por lo general, los programadores experimentan el tener que lidiar con código de alto grado de acoplamiento y complejo.

No existe una buena recomendación para enfrentar esta situación, otra que considerar no aplicar TDD, o en todo caso, iniciar un proyecto para reemplazar el código legado por un sistema desarrollado de acuerdo a prácticas actualizadas de orientación a objetos.

En el caso de equipo de desarrollo bajo estudio, la gran mayoría de los proyectos son microservicios creados por ellos mismos, por lo que tienen el control de los mismos durante la mayor parte de su evolución, no teniendo que lidiar con código legado.

10.2.5. Sistemas Dependientes de Aplicaciones Externas

TDD puede ser difícil de aplicar cuando existan dependencias con otros sistemas externos, que no están bajo el control del equipo de desarrollo, por ejemplo un Servidor FTP, Sistemas intermedios, Servicios Web, lectura de sensores, otros dispositivos de hardware.

Adicionalmente, es difícil de aplicar cuando para poder probar el software se requieren pruebas de extremo a extremo, por ejemplo interfaces de usuario, programas que usan bases de datos, dependencias de configuraciones de red, etc.

Para enfrentar este reto, se puede optar por colocar la menor cantidad de código en esos módulos y maximizar la cantidad de código que se puede probar, utilizando artificios para representar el mundo exterior (código que devuelve una repuesta predefinida). Esto implica aprender a construir objetos emuladores (*Mocks*) y falsificar las respuestas de sistemas colaborativos, para así permitir pruebas unitarias sencillas.

10.2.6. Desarrolladores Diseñando sus Propias Pruebas

Si las pruebas son diseñadas por el mismo desarrollador, las situaciones o casos no identificados en el código tenderían a ser los mismos siempre, no identificando todos los posibles errores. Aún peor, podrían existir casos de funcionalidad no contemplados. De hecho, ante cualquier mala interpretación, tanto el caso de prueba como el código estarán mal diseñados.

Se puede optar por incorporar a los analistas de calidad para trabajar en pares con los programadores. Bajo este esquema el analista no realiza la inspección después que se ha entregado el desarrollo sino durante y constantemente apoya al programador revisando su trabajo.

10.2.7. Perdida del Foco en las Pruebas tras Resultados Positivos

Cuando se inicia un desarrollo, se ha observado que si en las primeras iteraciones se identifican pocos defectos (los casos de prueba pasan), el equipo podría no hacer tanto énfasis en identificar casos de prueba adicionales o en hacer más pruebas de integración, es decir, tiende a relajarse.

Para afrontar esta situación, debe hacerse énfasis en la aplicación disciplinada de una metodología única para identificar los casos de prueba, independientemente de los resultados iniciales positivos, revisar los casos de prueba. Aquí puede jugar un papel importante el líder de QA, quien bajo este esquema ágil pasa de ser un responsable del área a ser actividades de acompañamiento y *coaching* a todos los programadores y analistas de pruebas.

10.2.8. Constantes Iteraciones y Refactorizaciones

Las constantes iteraciones y refactorizaciones son comunes bajo el desarrollo enfocado a TDD. Está situación implica una alta posibilidad de que alguna funcionalidad que ya había sido probada, luego de una refactorización, falle.

El diseño e implementación de software con el mayor grado de desacoplamiento posible, donde cada pieza de código debe cumplir una responsabilidad única, es muy importante para controlar esta situación. Además, los controles de versiones deben ser automatizados y aplicarse de forma disciplinada.

Una solución también es la automatización de las pruebas de regresión, probar que lo certificado funcione adecuadamente y luego seguir *refactorizando*.

10.2.9. Alto Costo de Errores No Identificados

El costo de errores no identificados es más alto, dado que si una condición no es identificada hasta el final, podrían requerirse modificaciones que obliguen a repetir casos de pruebas, y al haber trabajado de forma iterativa, repetir el proceso tiende a ser muy costoso.

Para minimizar este tipo de situaciones se puede optar por lo siguiente: Incorporar a analistas de prueba especializados desde el principio del desarrollo; Identificar la mayor cantidad de casos de prueba en la primera iteración; Incorporar revisiones del diseño de pruebas; Incorporar revisiones de los casos de prueba en iteraciones posteriores para asegurar que la identificación temprana de todos los casos.

En conclusión, el camino para adoptar TDD puede ser costoso y largo, pero vale la pena los beneficios que se logran, en términos del incremento de la productividad del equipo, minimización del re-trabajo por errores identificados de forma tardía y la posibilidad de entregar productos de software con una menor cantidad de errores.

De todas maneras, si se decide no aplicarlo, se puede seguir adelante con el plan, salvo que no se estaría garantizando disponer de pruebas suficientes, quedando a criterio de los responsables el exigir a sus recursos la creación de los mismos. La metrica de covertura de código puede ser usada en este caso como una linea base, la cual puede ser consultada constantemente para garantizar que la cobertura en el mejor de los casos mejore, o al menos que no empeore.

10.3. Obtener Métricas de Calidad con Inspección Continua

Un gran paso hacia la madurez en el desarrollo de software es dado cuándo se practica la inspección continua del código. Aplicar dicha práctica es lo que proponemos también para nuestro equipo de desarrollo bajo estudio. De qué trata esta práctica y los beneficios que brinda se pueden ver en detalle en la sección *7.4. Inspección Continua*.

La práctica de inspección continua nos permitirá conocer en todo momento la *solidez* de nuestros cimientos, buscando como objetivo principal minimizar la deuda técnica (*7.5. Deuda Técnica*), reforzando así el valor añadido de cada entrega.

Gracias a la inspección continua conseguiremos que el equipo tenga un conocimiento objetivo, unificado y compartido del estado del proyecto, poniendo la deuda técnica bajo control.

10.3.1. Utilizar un Servidor de Inspección Continua

Al igual que con la Integración Continua, para la Inspección Continua debemos valernos del apoyo de herramientas especializadas. Instalar un servidor de Inspección

Continua, es decir, un servidor de análisis estático de código, sería el siguiente paso en nuestro camino hacia el nuevo Entorno Integrado.

10.3.1.1. Elección del Servidor de Inspección Continua

Hasta la fecha, no hay muchas alternativas sin costo en este punto, y sin dudas, ninguna que supera las prestaciones que brinda *SonarQube* en el mundo Java, por lo que esta será la herramienta a usar para lograr nuestro fin.

SonarQube es una plataforma *open source* de gestión de calidad del código, dedicada a analizar y medir continuamente la calidad técnica de los proyectos ofreciendo información visual sobre la salud de los distintos proyectos y su evolución en el tiempo.

SonarQube es un servidor de análisis de código que cubre los aspectos más importantes de la calidad, llamados los 7 ejes de la calidad del código. Ellos son:

- ✓ *Arquitectura y Diseño*:

  Identificar los ciclos entre los paquetes mostrando todas sus dependencias y así poder identificar las no deseadas (diseño espagueti). También dependencias con paquetes de terceros. El Acoplamiento y cohesión del código, etc.

  Incluso se pueden crear reglas de exclusión de llamadas, por ejemplo que no se pueda hacer llamadas entre capas no adyacentes.

- ✓ *Duplicaciones*:

  Detecta las duplicaciones de construcciones de código, favoreciendo la reutilización de métodos y el diseño modular. Básicamente previene el famoso *copy paste*.

- ✓ *Test Unitarios*:

  Se centra en la cobertura que tiene el código en relación a las pruebas, es decir, en qué porcentaje nuestro código está cubierto por test unitarios y/o de integración.

- ✓ *Complejidad*:

  Monitorear la complejidad del código, controlando por ejemplo, la cantidad de líneas de código de un método, clase y/o archivo.

- ✓ *Errores Potenciales*:

  Básicamente realizar controles de todas las situaciones conocidas que pueden resultar en un error futuro, como lo es por ejemplo no chequear por una posible excepción de *null pointer.*

- ✓ *Reglas de Codificación*:

Controlar las convenciones de código según plantillas que se pueden configurar para seguir las mejores prácticas y/o prácticas de la organización en cuanto a las reglas de codificación.

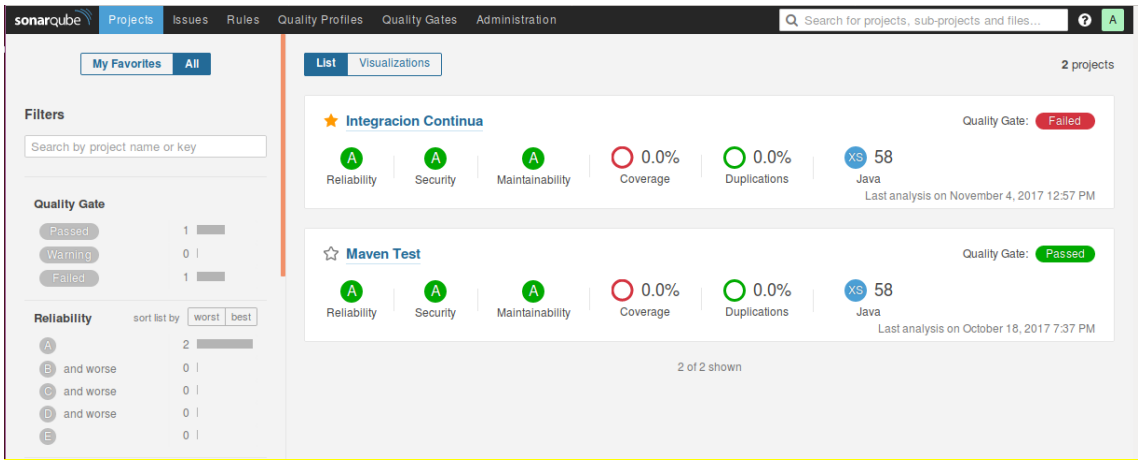
✓ *Comentarios*:

Descubrir métodos y clases no comentadas, perdiendo así información de primera mano en cuanto a la función de las mismas. Incluso también controlar el exceso de comentarios.

Para realizar este análisis, SonarQube se vale de motores de análisis de código conocidos y probados del mercado, como *Findbugs*, *Cobertura*, *PMD*, *Surefire, CheckStyle* entre otros.

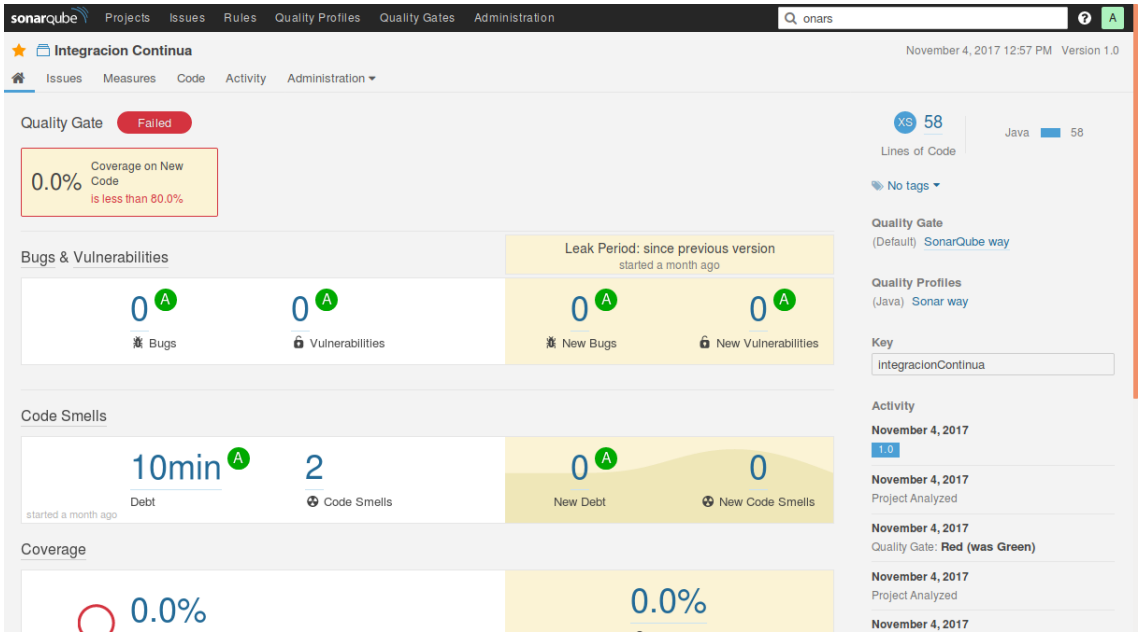
Los resultados de estos análisis pueden ser guardados en una base de datos dándonos la posibilidad, no solo de disponer de la instantánea actual de la salud de nuestro proyecto, sino de la evolución de la calidad de nuestro código en el tiempo.

El servidor tiene una interfaz web donde se pueden configurar muchos aspectos, tal como las reglas, agregando o quitando tantas como se quisiera, el tipo de notificaciones, la información que se quiere ver aparte del control de acceso a la herramienta.



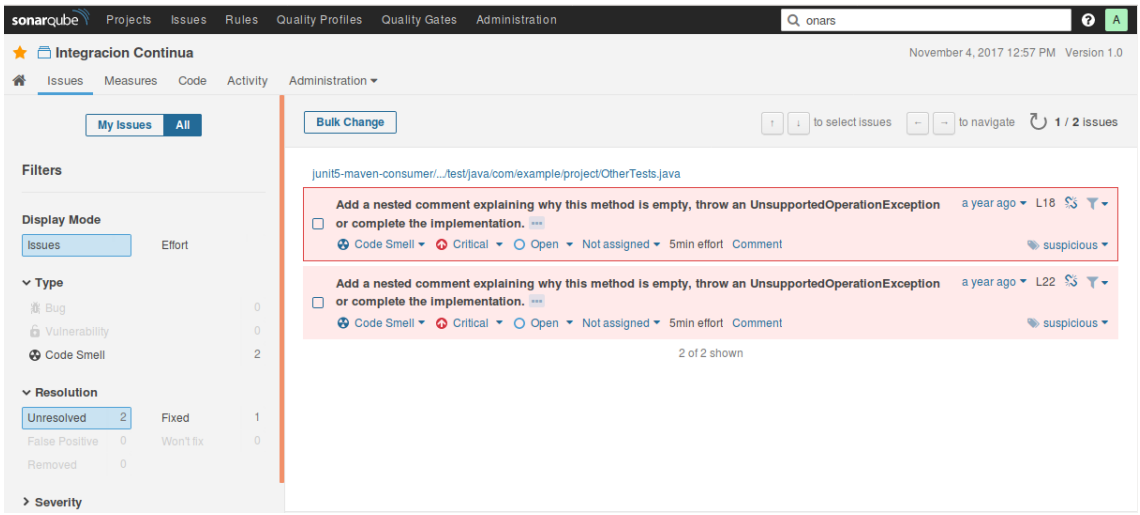
Captura de Pantalla 10.4: Consola de SonarQube (todos los proyectos)

Como se puede ver en la imagen arriba, la consola principal lista todos los proyectos que se estan monitoreando, dando rapidamente el estado general de salud de cada uno. También se tiene la posibilidad de poner el foco en un proyecto a la vez, como la muestra la siguiente imagen, donde se ven las metricas y la salud del proyecto con un poco mas de detalle para el proyecto seleccionado.



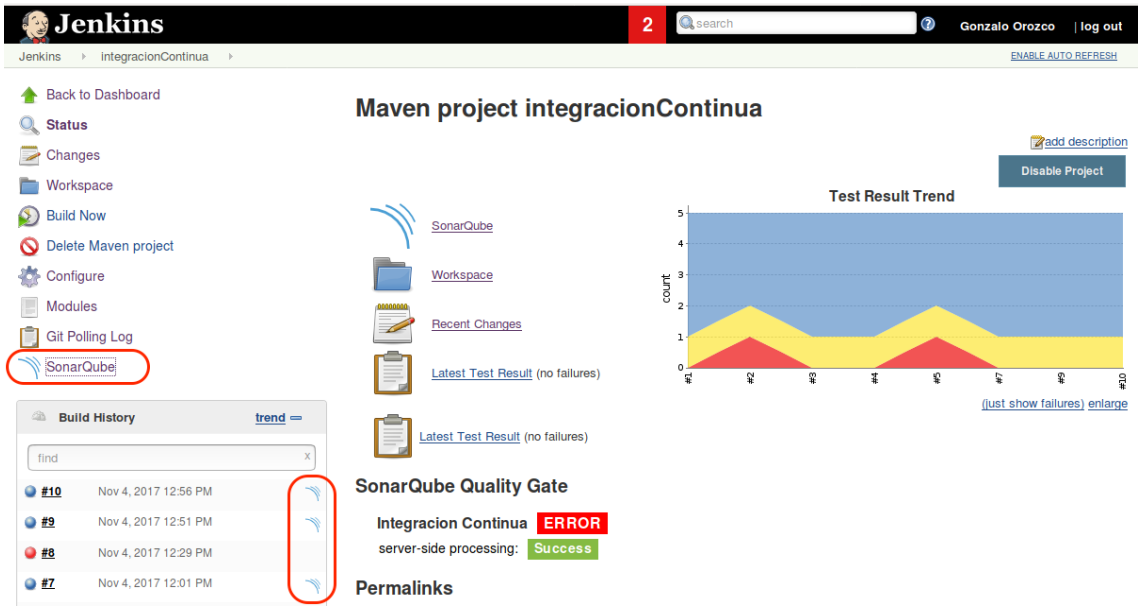
Captura de Pantalla 10.5: Consola de SonarQube para un proyecto

Algo a destacar es que nos permite hacer doble click a cada una de las métricas o problemas potenciales detectados, incluso hasta llegar a la línea de código en cuestión, lo que muestra la potencia y beneficios de contar con esta herramienta.



Captura de Pantalla 10.6: Vista de Issues para el proyecto seleccionado

El acceso al proyecto en SonarQube puede hacerse directamente como vimos e incluso desde el mismo Jenkins, al estar ambas herramientas integradas, en la consola de Jenkins, dentro de cada job, tendremos acceso directo a SonarQube para el proyecto en cuestión.



Captura de Pantalla 10.7: Acceso a las estadisticas del proyecto en SonarQube

La interacción de nuestro servidor de Inspección Continua con el servidor de IC y las demás herramientas la veremos en detalle en la próxima sección. La instalación y configuración de esta herramienta la describimos en el *ANEXO C: Configuración del Nuevo Entorno de Desarrollo Integrado.*

10.4. Nuevo Entorno de Desarrollo Integrado

Ya expusimos las soluciones aisladamente, ahora nos enfocaremos a cómo interactúan entre sí, la infraestructura que las soporta y cuál es su resultado.

Con esta interacción de buenas prácticas, apoyadas en herramientas especializadas, pasamos de la configuración actual (***Figura 8.1*** *- Entorno de Desarrollo Actual*) en donde convivían todos los problemas identificados, los cuales motivaron el presente trabajo, a una nueva configuración en la cual funcionan mancomunadamente las prácticas de Integración Continua, el Desarrollo Guiado por Pruebas y la Inspección Continua. Esta nueva configuración la denominamos **Entorno de Desarrollo Integrado**. El mismo soporta todo el proceso de desarrollo brindando todos los beneficios ya explorados para cada una de estas prácticas.

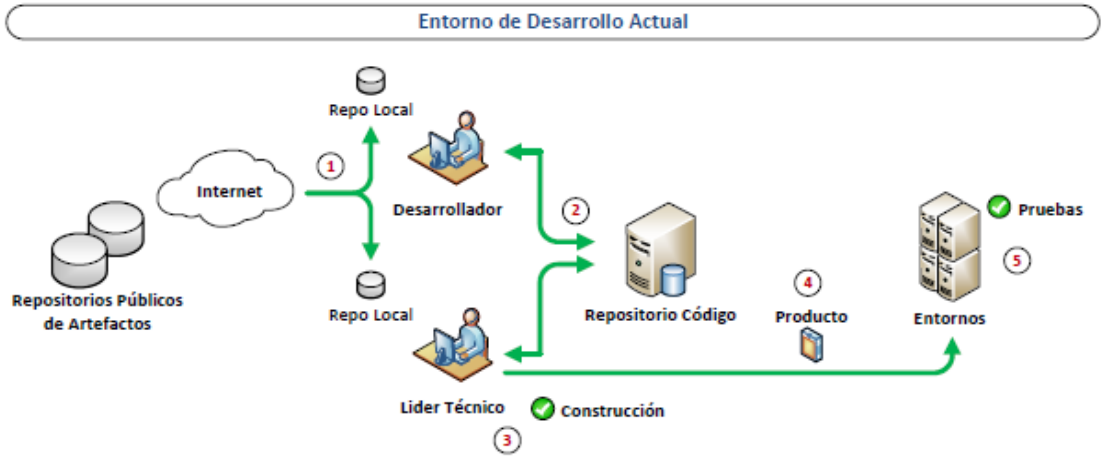


Figura 8.1 – Entorno de Desarrollo Actual

El nuevo Entorno de Desarrollo Integrado debe ser administrado y mantenido por el equipo encargado de la Gestión de la Configuración (*CM – Configuration Manager*). Las tareas de este equipo encargado de la configuración se pueden ver con más detalle en el *Anexo C: Configuración del nuevo Entorno de Desarrollo Integrado*

10.4.1. Arquitectura del Entorno de Desarrollo Integrado

El corazón del nuevo Entorno de Desarrollo Integrado está compuesto por el Servidor de Integración Continua. El mismo es el encargado de monitorear los cambios en el Repositorio de Código para luego disparar las acciones predeterminadas, como construir el código (incluyendo la gestión de dependencia), ejecutar las pruebas unitarias y de integración, ejecutar el análisis de código estático apoyado en el Servidor de Inspección Continua, e incluso desplegar la construcción (probada y estable) a los diferentes ambientes.

Lo que mantiene todo unido y coordinado, el lenguaje común entre todos los elementos que conforman el entorno, es la herramienta de construcción *Maven*. No figura en el gráfico simplemente porque está involucrada en casi todo el trabajo que realiza este nuevo entorno.

Notaran al ver el siguiente gráfico (*figura 10.1*) que la arquitectura del entorno de desarrollo del equipo bajo estudio mutó de manera significativa a otra más sofisticada. Se podría pensar que la instalación, configuración y mantenimiento de este ambiente sería muy costoso, pero todas las herramientas añadidas al mismo son de libre uso y muy fáciles de instalar. Quizás lo más complejo es la configuración de las mismas para que funciones en conjunto, pero expondremos esta configuración en la sección en el Anexo C.

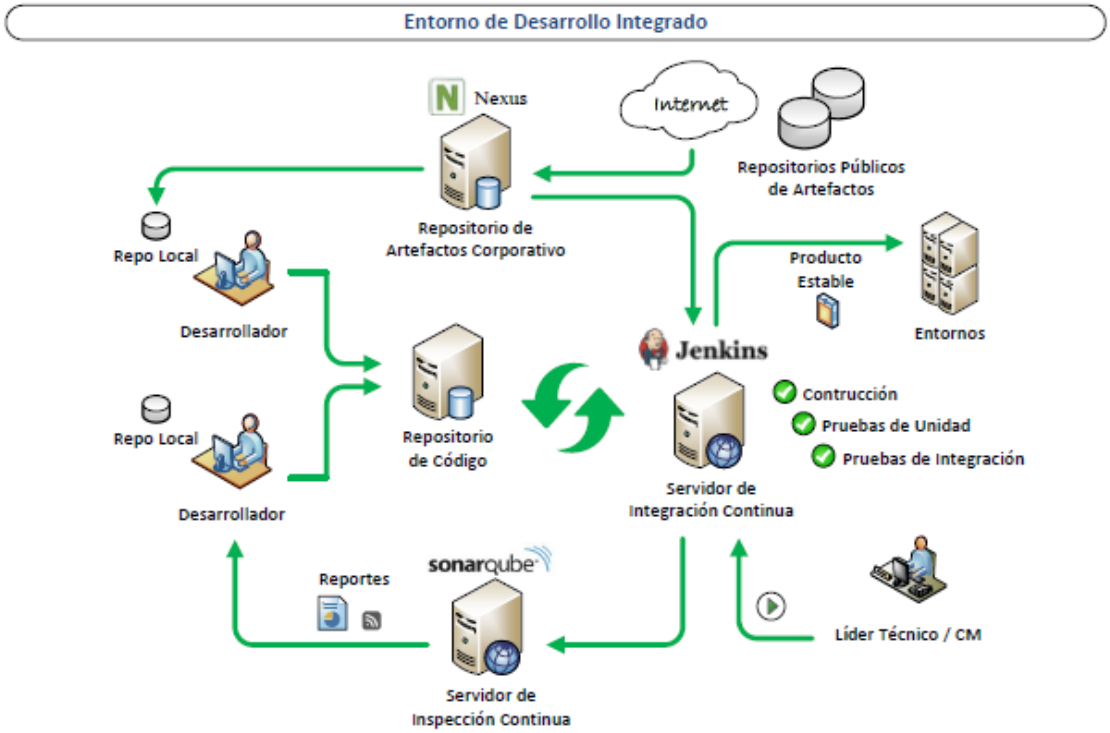


Figura 10.1 - Entorno de Desarrollo Integrado

En el diagrama se puede ver donde juegan su importante papel las distintas herramientas que elegimos, *Jenkins* para el servidor de IC, *Nexus* para el repositorio de artefactos y *SonarQube* para el servidor de Inspección Continua. Cabe destacar que estas herramientas pueden ser reemplazadas por otras en su lugar sin afectar esta nueva arquitectura. Incluso Maven puede ser reemplazado por otra herramienta de construcción, como Gradle o, con un poco más de dificultad, Ant.

10.4.2. Funcionamiento del Entorno de Desarrollo Integrado

Describiremos a continuación los diferentes aspectos del nuevo entorno poniéndose énfasis en como aplica los diferentes conceptos y las buenas prácticas identificadas como solución.

10.4.2.1. Gestión de Dependencia Efectiva

Gracias a la herramienta de gestión de proyectos de software *Maven*, que ya utilizaba el equipo bajo estudio, y sobre todo al Repositorio de Artefactos que incorporamos al nuevo entorno integrado de desarrollo, ahora la gestión de dependencia es mucho más efectiva y controlada, teniendo la posibilidad de poder administrar este proceso con una herramienta visual.

A continuación vemos parte del entorno de desarrollo integrado en donde la gestión de dependencia juega un papel importante y explicaremos cómo se gestiona la misma.

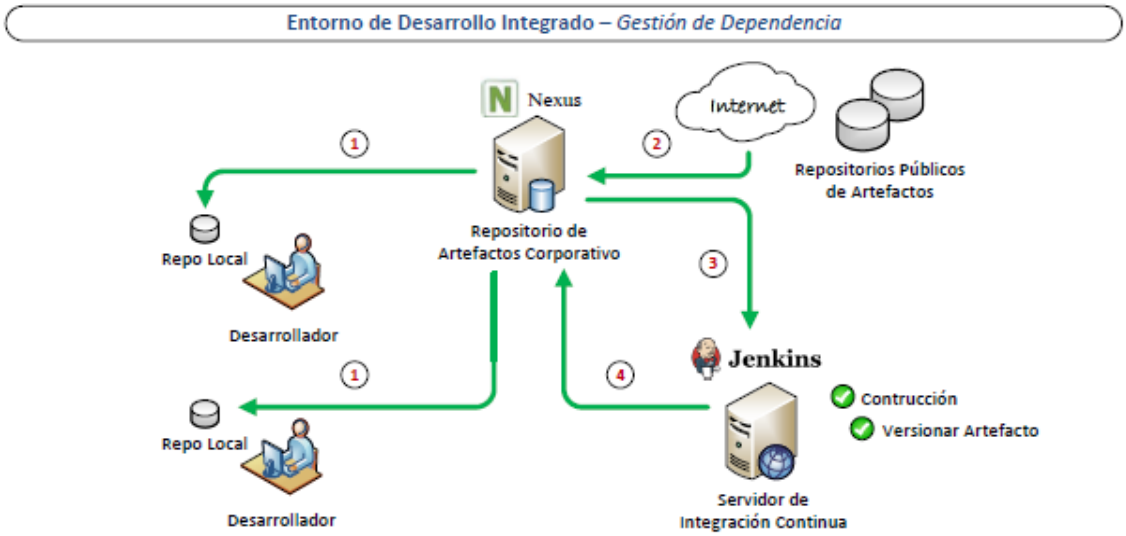


Figura 10.2 – Entorno de Desarrollo Integrado - Gestión de Dependencia

(1) Cada desarrollador, al compilar su código, implícitamente delega a *Maven* la gestión de dependencia, y este al estar configurado para resolver las dependencias a través del Repositorio de Artefactos Corporativo, descarga las dependencias desde ese repositorio, dentro de los límites de la intranet, no teniendo que descargarlo directamente desde repositorios públicos en la Internet. De este modo se tiene total control de las dependencias, por ejemplo en cuanto a licencias y versiones.
(2) El Repositorio de Artefactos Corporativos se conecta a los repositorios públicos solo una vez por dependencia, en el caso de que no la tuviera todavía, y solo si, así lo permitiese el administrador, usando eficientemente de esta manera los recursos de conexión y respetando las políticas de empresa en cuanto a licencias.
(3) Al igual que los desarrolladores, el Servidor de Integración Continua, delega la gestión de dependencia al Repositorio de Artefactos Corporativos para resolver las dependencias en la construcción del código en una integración.
(4) Otra función del Repositorio de Artefactos Corporativo es alojar los artefactos generados por el equipo de desarrollo, por tal motivo, algunas tareas del Servidor de Integración Continua podrían incluir versionar un artefacto generado allí, para luego ser gestionado como cualquier otra dependencia de terceros.

10.4.2.2. Integración Continua en Acción

Aplicar Integración Continua a un proyecto es realmente fácil y con un costo bajísimo, y rápidamente se pueden observar los buenos resultados que brinda.

La base de Jenkins son las tareas (Jobs), donde indicamos qué es lo que hay que hacer para un determinado *build*. Se pueden configurar allí diferentes pasos. Las tareas pueden ser automáticas o manuales, según lo que se pretenda realizar. Por ejemplo:

- Una tarea automática para construir el código y correr las pruebas unitarias al nivel del entorno de desarrollo cuando se compromete código al repositorio.
- Una tarea manual para construir el código, correr las pruebas, y correr el análisis del código estatico al nivel del entorno de QA.
- Una tarea manual para desplegar una nueva construcción a un entorno específico.

Vemos a continuación que papel cumple en nuestro entorno.

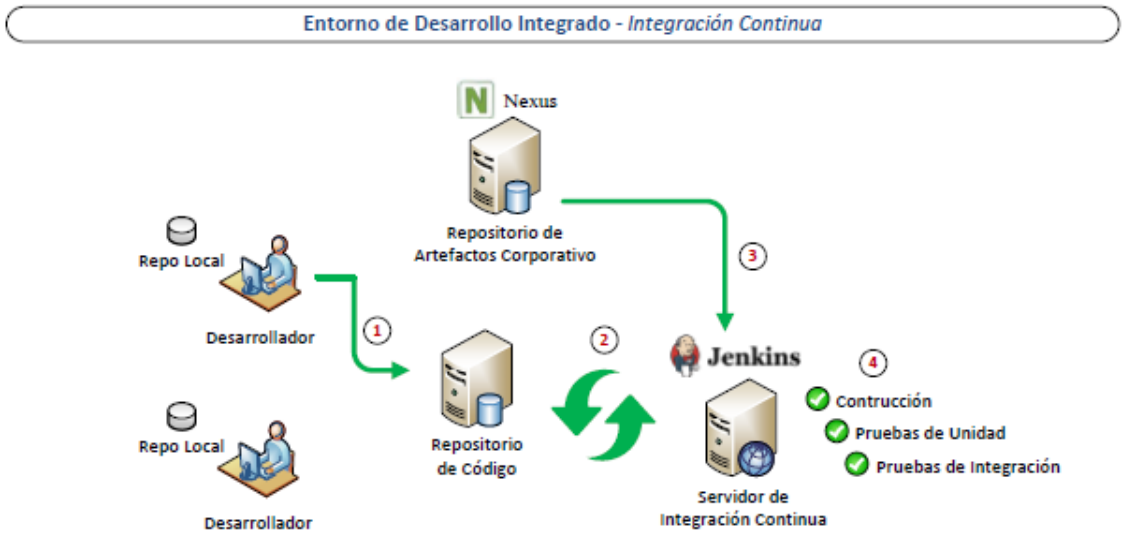


Figura 10.3 – Entorno de Desarrollo Integrado – IC

(1) Para la práctica de integración Continua el punto de entrada es el código que un desarrollador compromete al repositorio de código.
(2) El servidor de IC aquí tiene varias posibilidades de configuración, una de ellas es constantemente (o a intervalos regulares) monitorear el repositorio de código en busca de algún cambio, y al encontrarlo realizar alguna acción preestablecida. Otra opción podría ser manual, es decir, solo ejecutar esa acción a pedido de alguien.
(3) Al igual que los desarrolladores, el Servidor de Integración Continua, delega la gestión de dependencia al Repositorio de Artefactos Corporativos para resolver las dependencias en la construcción del código en una integración.
(4) Es aquí donde se realizan los trabajos previamente configurados, como lo pueden ser la construcción del código, la realización de pruebas de unidad y/o integración, un posterior despliegue, etc.

Un aspecto muy importante de la IC es la comunicación del estado del proyecto, ya sea a través de la interfaz visual del servidor o a través de otros medios como avisos por email, e incluso por mensajes de texto.

Queremos exponer a continuación la comunicación que se realiza en el posible escenario de falla de una tarea del servidor de IC, en este caso, al realizar las pruebas unitarias luego de la construcción del código.

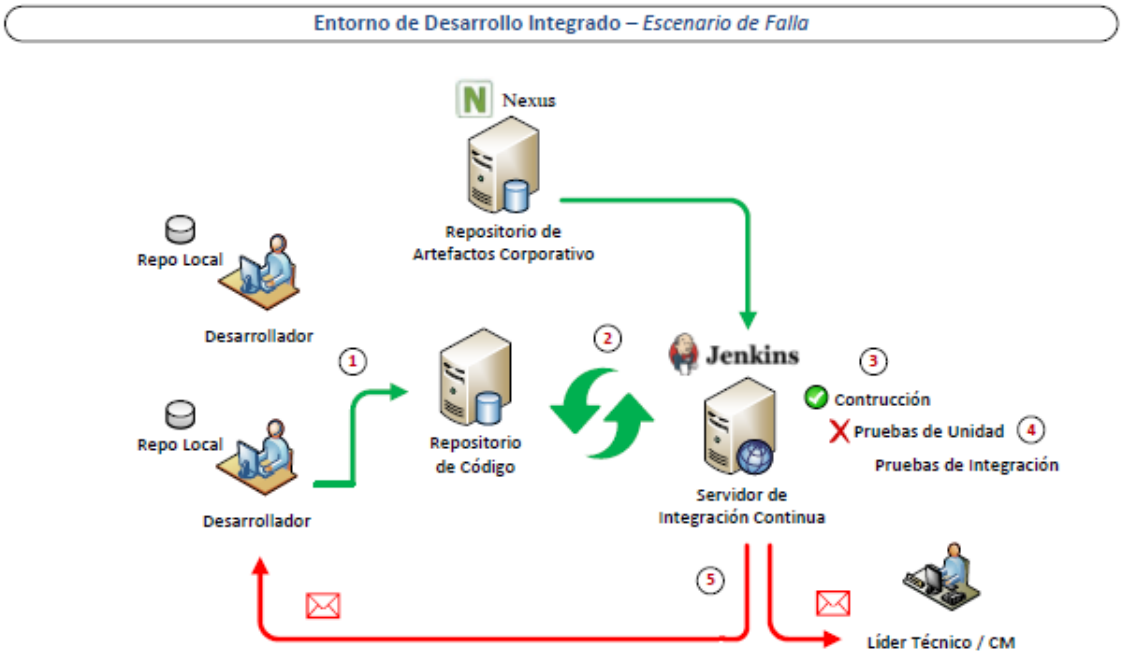


Figura 10.3 – Entorno de Desarrollo Integrado – Escenario de Falla

(1) Otra vez, el punto de entrada es el código que un desarrollador compromete al repositorio de código.
(2) El servidor de IC monitorea el repositorio de código, y al encontrar el cambio, realiza el conjunto de acciones preestablecidas.
(3) Aquí se realizan los trabajos previamente configurados, como lo pueden ser la construcción del código, la realización de pruebas de unidad y/o integración, un posterior despliegue, etc.
(4) Este es el escenario que queremos describir, luego de la construcción del código, comienza a ejecutar las pruebas unitarias, y una de ellas falla.
(5) Al encontrar una falla, o cualquier tarea preestablecida que no finaliza de la forma esperada, el servidor de IC detiene el proceso, lo marca como terminado con fallas y notifica a los involucrados. En el mensaje se identifica a la persona que comprometió el código que hizo fallar la tarea y la línea de código que causa el problema, junto a toda la traza de errores.

Como ya dijimos la comunicación es un pilar importante de la IC, por eso, luego de que un *build* halla fallado, cualquiera haya sido el motivo, la herramienta también nos informará automáticamente una vez que el *build* vuelva a ser estable, es decir, que se construya sin errores y todas sus pruebas hayan pasado exitosamente.

Vemos con mucha claridad en el anterior diagrama (*Figura 10.3*) uno de los grandes beneficios de contar con esta herramienta y entorno, manteniendo al equipo informado de la salud del *build* en todo momento.

10.4.2.3. Inspección Continua de la Calidad del Código

Con la herramienta elegida para la Inspección Continua del código, la calidad del mismo deja de ser un concepto abstracto, pudiendo ahora mesurarla e incluso planificarla pudiendo establecer una linea base y de ahí en más medirse contra ella. Veremos ahora que papel juega el servidor de Inspección Continua *SonarQube* en nuestro nuevo entorno de desarrollo integrado.

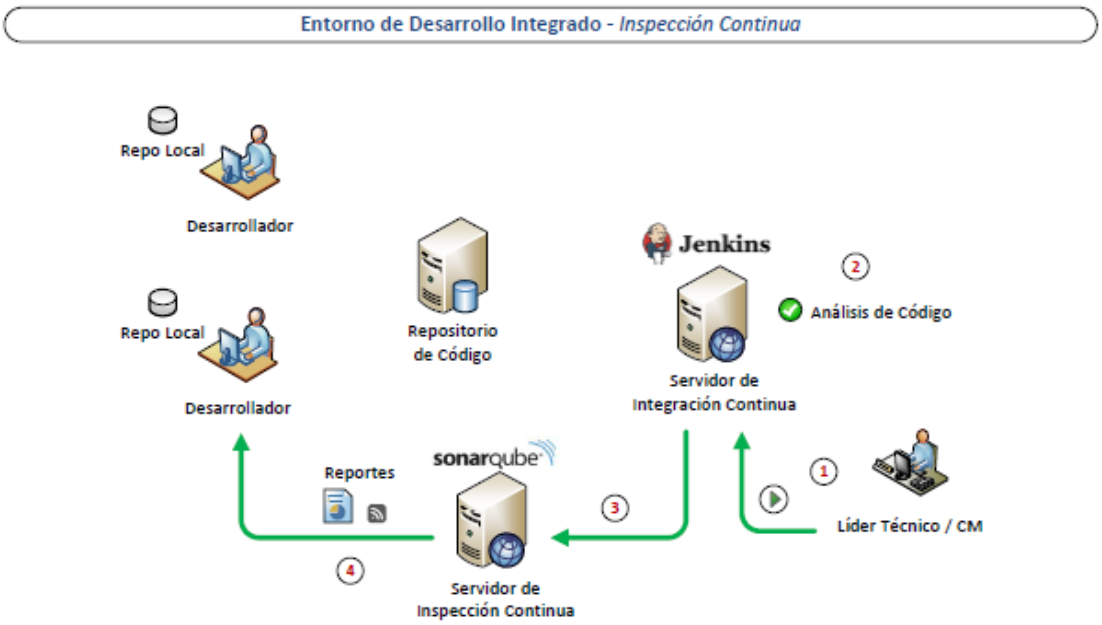


Figura 10.4 – Entorno de Desarrollo Integrado – Inspección Continua

(1) Si bien el análisis de código puede realizarse automáticamente después de la construcción del código, puede considerarse una tarea costosa en tiempo y suele manejarse manualmente en ambientes de desarrollo, por ejemplo una vez al terminar el sprint actual o por reléase, incluso directamente en el entorno de QA.
(2) El servidor de IC ejecuta el análisis de código estático valiéndose en este caso de una extensión de SonarQube para Jenkins que le permite correr la batería de análisis predeterminada.
(3) Una vez terminado el análisis, el servidor de IC envía los resultados al Servidor de Inspección Continua, donde es estructurada, almacenada y comparada históricamente con resultados anteriores.
(4) Todos los resultados están a disposición de los involucrados, ya sea a través de la interfaz del servidor de Inspección Continua o a través de reportes que el mismo servidor genera.

Ya hablamos de lo importante que es en un ambiente ágil administrar la deuda técnica, quizas muy dificil o utópico sin una herramienta como esta, la cual nos permite hacer objetiva la cualificación y cuantificación de la deuda, y sobre todo permitir definir una linea base de calidad y desde ahí planificar como mejorarla constantemente.

10.4.2.4. Despliegue Automático del Producto

Cabe destacar aquí que con despliegue automático nos referimos a la automatización de los pasos, que debería existir en forma de algún tipo de script que será ejecutado desde un *job* dentro de la interfaz del servidor de IC.

Aquí se pueden establecer distintas estrategias, la más simple sería tener un *job* por repositorio y entorno. Es decir, dado los ambientes de *Desarrollo* y *QA* para un repositorio A existirá un un job *A_dev* y uno *A_qa*, donde el job *A_dev* tendría la tarea de buildear el código una vez que alguien lo comprometa a la linea base, correr las pruebas de unidad y de integración y realizar el despliegue ejecutando un script automaticamente del o los artefactos creados al servidor de desarrollo si el build se ejecuta correctamente y todas las pruebas pasan. El job *A_qa* podría tener el mismo comportamiento agregando el analisis estatico del código y exeptuando el despliegue automatico, ya que podría entorpecer las pruebas que el equipo de QA pueda estar haciendo, dejando para otro job tal tarea de despliegue al servidor de QA, que puede ser ejecutado por el mismo equipo a cargo, eligiendo la version deseada.

En el siguiente grafico vemos nuestra propuesta según las necesidades del equipo bajo estudio.

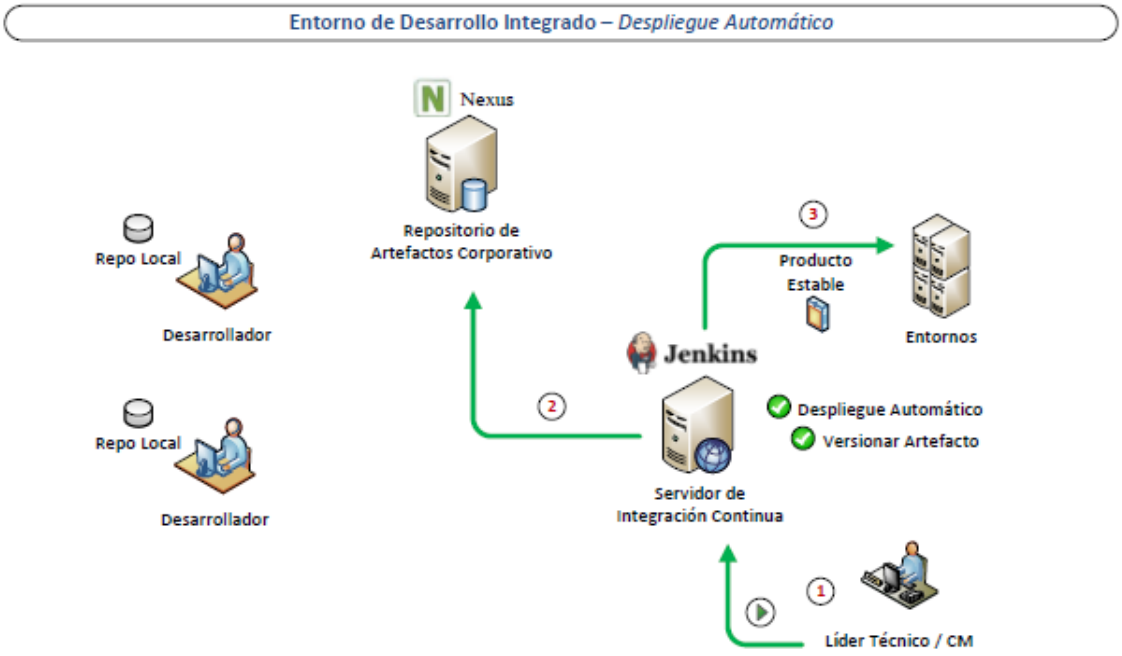


Figura 10.5 – Entorno de Desarrollo Integrado – Despliegue Automático

(1) El líder técnico o el CM encargado ejecuta la tarea correspondiente para desplegar la versión en el entorno en cuestión para que esté disponible para pruebas por parte del equipo de QA o para la aceptación de los cambios por parte del cliente.
(2) El servidor de IC busca la versión correspondiente en el repositorio de artefactos evitando así tener que construirla nuevamente.
(3) El servidor de IC despliega la versión en el ambiente seleccionado según la configuración del *job* seleccionado.

Las posibilidades y convinaciones con esta herramienta son infinitas, y la estrategia de despliegue puede cambiar según cada proyecto e incluso durante la vida del mismo.

Aquí es importante entender lo que logramos con esta práctica de despliegue automatizado. Si disponemos de entornos bajos (desarrollo, QA, etc) similares al entorno de producción, nuestro script de despliegue para cada uno de ellos debería ser tambíen muy similar al utilizado en producción, o aún mejor, el mismo, por lo que estaríamos entrenando el script constantemente, logrando una gran confianza en el mismo y en el proceso de despliegue en general para cuando llegue el momento del despliegue en producción.

11. Resultados

Trataremos en esta sección de describir los resultados que percibimos al aplicar la solución propuesta. Primero se presentó una demostración con un proyecto de ejemplo aplicando la solución completa, obteniendo muy buenos resultados y generando un mayor entendimiento y entusiasmo por parte de los tomadores de decisiones y líderes del equipo bajo estudio. Luego que se aprobara la solución por parte de los responsables de tomar estas decisiones se procedió a realizar una prueba piloto sobre uno de los proyectos de la cuenta.

Ahora bien, retrotrayéndonos a la etapa de relevamiento, quisiéramos recordar los problemas o síntomas que el equipo bajo estudio estaba sufriendo:

- *Excesivo esfuerzo de integración de código luego que dos o más proyectos convergen,*
- *Compilación del código a entregar en ambientes no controlados,*
- *La mayoría del código no está cubierto por pruebas unitarias (pobre cobertura del código),*
- *Mayor detección de errores en entorno de producción (Detección tardía de errores),*
- *Código de pruebas unitarias y de integración que quedan obsoletos,*
- *Falta de gestión de artefactos de terceros u de otros proyectos (Gestión de dependencia pobre)*
- *Falta de confianza en la robustez del código,*
- *Inexistencia de indicadores de calidad del código.*

Para cada uno de estos problemas, expondremos a continuación como nuestra solución mitigo los mismos en la prueba piloto.

11.1. Integración diaria del código

Problema: *Excesivo esfuerzo de integración de código luego que dos o más proyectos convergen*.

Si bien la prueba piloto se realizó sobre solo uno de los proyecto dentro de este equipo, se pudo ver como la solución propuesta respecto a este tópico pudo reducir los tiempos de integración drásticamente entre las diferentes tareas de los desarrolladores involucrados en este proyecto. Como propusimos, se cambió la estrategia de control de versión del código en paralelo a una estrategia lineal, esto es, todos los desarrolladores comprometen su código a la línea base diariamente y no solo a su *branch* personal cuando terminan su tarea, como se hacía anteriormente.

Al momento de llegar a entregar una versión del producto, normalmente sin mucho margen de tiempo, no se tuvo que asignar esfuerzo a la integración (y normalmente, resolución de conflictos producto de la integración) ya que ese esfuerzo se realizó diariamente durante toda la vida del desarrollo del producto, siendo este esfuerzo

menor y siendo la integración mucho más sencilla, ya que el desarrollador solo tiene que integrar lo que realizó recientemente, teniendo el código fresco en la memoria.

11.2. Construcción automática del código en ambiente controlado

***Problema**: Compilación del código a entregar en ambientes no controlados.*

Al disponer ahora de un servidor controlado de IC (S.O, versión de JVM, versión de dependencias de terceros, etc), y siendo todas las construcciones de producto realizadas por el mismo servidor de IC, automáticamente se solucionó este problema.

Ahora podemos estar seguros que siempre es el mismo producto el que se despliega en los diferentes ambientes, y sobre todo, tenemos la seguridad que el producto es repetible, es decir, puede volverse a construir idéntico desde sus archivos fuentes.

11.3. Mayor cantidad de código cubierto por pruebas unitaria

***Problemas**: La mayoría del código no está cubierto por pruebas unitarias (pobre cobertura del código) y Mayor detección de errores en entorno de producción (Detección tardía de errores).*

La solución definitiva de estos problemas se podrá ver cuando la organización y en particular el equipo bajo estudio empiecen a adoptar la *Metodología de Desarrollo Guiado por Pruebas (TDD)*. Como esta parte de la solución depende de una capacitación y tiempo de adopción que quedan fuera del alcance de este estudio, no se pudo recoger resultados definitivos sobre este tópico.

Aun así, se procedió a crear pruebas unitarias para los procesos más importantes, y al ejecutarse estas pruebas diariamente cada vez que se promovía el código, se pudo observar la valía de estas, y no se consideraron una pérdida de tiempo como se lo hacía hasta el momento, ya que si bien se creaban, solo se ejecutaban un par de veces manualmente. Al ejecutar diariamente la batería de pruebas unitarias de que disponen, se aseguran que esa parte del código que está cubierto sigue funcionando correctamente.

Aquí también los reportes sobre cobertura de código que arroja *SonarQube* dan una idea de en qué porcentaje el código del proyecto está cubierto con pruebas unitarias, dando una unidad objetiva para los lideres, quienes pueden alentar cada vez más una mayor cobertura y por ende, una mayor creación de pruebas unitarias.

11.4. Ejecución automática de pruebas unitarias y de integración

***Problema**: Código de pruebas unitarias y de integración que quedan obsoletos.*

Este problema se solucionó al tener la ejecución de pruebas existentes automatizadas gracias al servidor de IC, esto obliga a los desarrolladores a mantener actualizada su batería de pruebas, porque de lo contrario, si el código no pasa las pruebas, la construcción falla.

11.5. Gestión de artefactos de terceros y de otros proyectos

***Problema**: Falta de gestión de artefactos de terceros u de otros proyectos (Gestión de dependencia pobre).*

Al disponer del repositorio de artefactos corporativo (*Nexus)*, ahora las dependencias están completamente controladas por el equipo de configuración y todos los desarrolladores consiguen estas dependencias del mismo lugar, usando la misma version para cada dependencia y no impactando constantemente los recursos de conectividad de la organización. Del mismo modo, los artefactos propios creados se versionan y alojan en este servidor, estando siempre al alcance de quienes lo necesiten, del mismo modo que lo estan las dependencias de terceros.

11.6. Inspección continua de la calidad del código

***Problemas**: Falta de confianza en la robustez del código e Inexistencia de indicadores de calidad del código.*

La confianza en la robustez del producto aumento gracias a la aplicación de todas las medidas propuestas por un lado, e irá en aumento a medida que el porcentaje del código que está cubierto con pruebas unitarias sea cada vez mayor al aplicar la metodología de *TDD*. Por otro lado, al disponer de métricas objetivas, el equipo vio donde está parado y se pudo definir metas de mejoras, es decir, empezar a pagar los intereses de la deuda técnica.

Ahora el producto es repetible y es sometido constantemente a la ejecución de las pruebas unitarias disponibles.

La solución al problema de la inexistencia de indicadores de calidad del código fue quizás el más visible y el que más impacto tuvo en la organización. Gracias a la implementación de la práctica de Inspección Continua con *SonarQube*, se pasó de la ignorancia absoluta sobre la salud técnica del proyecto a tener indicadores de todo tipo sobre la calidad del código, desde el diseño de los componentes, indicadores de cobertura del código en relación a las pruebas, estándares de codificación, y hasta detección de potenciales errores.

Esto permitió trazar una línea base, con la cual se empezará a administrar la deuda técnica adquirida.

12. Conclusión Final

La Integración Continua y las demás prácticas y principios complementarios tratados a lo largo de este trabajo están alineadas bajo un mismo lema: "*el producto siempre está listo para ser liberado*". Esto cambia el paradigma usado en la organización hasta el momento, incorporando cambios de todo el equipo a una modalidad en que el producto va creciendo a partir de cambios incrementales y siempre se encuentra listo para ser desplegado a los distintos ambientes.

La Integración Continua está íntimamente relacionada con la comunicación, cada uno de los involucrados puede fácilmente visualizar el estado del producto y su salud durante todo su desarrollo permitiendo tomar decisiones sobre evidencias objetivas como cobertura de casos de prueba y deuda técnica, algo anteriormente impensado para la madurez de este equipo.

Con la aplicación de la práctica de IC demostramos que es posible, entre muchos otros beneficios, mejorar la productividad y asegurar la calidad del código que este equipo de desarrollo ágil produce. También estamos convencidos que esta misma solución es aplicable a cualquier otro equipo en una situación similar, más allá de las herramientas que se elijan para implementarla. *Un equipo que trabaja con metodologías agiles debe tener un entorno de desarrollo que acompañe esa metodología*.

No estamos ajenos a otras prácticas como la de Entrega Continua (*Continuous Delivery*) y Despliegue Continuo (*Continuous Deployment*), pero ambas son una evolución de la Integración Continua, es decir, no se puede realizar la Entrega Continua sin dominar la Integración Continua, ya que IC forma parte activa de esta metodología. Lo mismo pasa con el Despliegue Continuo, ya que el mismo es una evolución de la Entrega Continua. Cada una de estas metodologías agrega más automatización al desarrollo y entrega de los productos de software. De esta manera la Integración Continua es un primer y gran paso hacia estas otras metodologías si así se lo deseara.

Otro aspecto muy positivo y visible es la relación costo-beneficio que deriva de implementar la solución, al ser todas las herramientas de libre uso, y siendo el ambiente muy automatizable, no debiendo los desarrolladores alocar horas a su funcionamiento, el resultado que arroja es un enorme beneficio, largamente explicitado en este trabajo, a un muy bajo coste. Si bien la solución requiere tiempo de instalación y mantenimiento por parte del equipo de configuración, estamos convencidos que ese tiempo es amortizado rápidamente en forma de beneficios.

Por último y no menos importante, me gustaría resaltar el cambio positivo que hubo en el equipo bajo estudio en relación al involucramiento en el producto y pertenencia al equipo, al descubrir de que manera ellos impactan directamente en la calidad del producto, y como esto motivo al querer mejorar esas metricas día a día.

Habiendo expuesto nuestro trabajo, llevandolo desde la teoria a la práctica a un muy bajo coste, y habiendo documentado y demostrado los resultados y beneficios, solo nos resta preguntarnos: *¿Por qué no aplicar Integración Continua?*

13. Bibliografía

[*Groth, 2004*] Groth, R. (2004). Is the software industry's productivity declining? IEEE Software, 21(6):92-94.

[*Fowler, 2006*] Martin Fowler (2006). Continuous Integration. http://www.martinfowler.com/articles/continuousIntegration.html

[*J.Humble/D.Farley, 2010*] Jez Humble and David Farley (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.

[*Wikipedia - Scrum*] http://es.wikipedia.org/wiki/Scrum.

[*Wikipedia - Technical Debt*] http://en.wikipedia.org/wiki/Technical_debt

[*Rally Software*] https://www.rallydev.com/

[*Atlassian - Jira*] https://www.atlassian.com/es/software/jira

[*Eclipse- IDE*] https://www.eclipse.org/

[*Smart Bear - SoapUI*] http://www.soapui.org/

[*Apache Maven*] http://maven.apache.org/

[*Apache Subversion*] https://subversion.apache.org/

[*Apache Tomcat*] http://tomcat.apache.org/

[*Oracle DB*] https://www.oracle.com/database/index.html

[*Oracle Java*] https://www.oracle.com/java/index.html

[*Oracle-SQLDeveloper*] http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html

[*Spring Framework*] http://projects.spring.io/spring-framework/

14. ANEXO A: Evaluación Comparativa de Servidores de IC

La elección del servidor de Integración Continua se realizo luego de comparar las tres opciones libres más populares al momento de realizar este trabajo, y a priori, las tres opciones más viables. Ellas son: ***CruiseControl***, ***Jenkins*** y ***Continuum***. Los detalles de esta comparación se pueden ver en el siguiente cuadro comparativo.

		Servidores de Integración Continua		
		Apache Continuum	Jenkins	CruiseControl
Criterios	Open Source	SI	SI	SI
	Dificultad de Instalación (1 - 3)	2	1	2
	Licencia	Apache License	MIT License	BSD-style
	Soporta SVN	SI	SI	SI
	Agenda Programable		SI	
	Soporta Maven	SI	SI	SI
	Notificacion		SI	
	Integración con Eclipse		SI	
	Mecanismo de Autentificación	SI	SI	SI
	Posibilidades de Despliegue Automatico		SI	

Claramente se puede ver la mayor versatilidad de Jenkins por sobre sus competidores, pero sobre todas las cosas, es facil de utiliar y el más usado por los desarrolladores y tecnicos de la industria, sumado a que es el que mayor cantidad de plugins tiene, la decisión fue facil.

15. ANEXO B: Evaluación Comparativa de Gestores de Repositorio de Artefactos

Como ya sabemos *Maven* utiliza una matriz de repositorios remotos que le permite localizar y descargar todo lo necesario para generar nuestro proyecto de forma transparente al desarrollador. Además de los repositorios remotos también existe un repositorio local que lo utiliza como caché evitando la descarga en las siguientes generaciones del proyecto y así reducir el tiempo que supondría volver a descargarse todas las librerías.

En determinados entornos, *Maven* puede ser más que suficiente, pero en grandes organizaciones seguramente no. La restricción de acceso a Internet, el control de acceso a los repositorios, la exclusión de ciertas librerías, la reducción del consumo del ancho de banda o la administración de los repositorios de la propia organización son aspectos que quedan fuera del alcance de *Maven*. Debido a estas necesidades aparecieron en el mercado los Gestores de Repositorios *Maven*.

En la actualidad podemos destacar tres: *Archiva*, *Artifactory* y *Nexus*. De los cuales armamos el siguiente cuadro comparativo.

		Gestores de Repositorio de Artefactos		
		Archiva	Artifactory	Nexus
Criterios	Dificultad de Instalación (1 - 3)	2	2	1
	Licencia	Apache License 2.0	Lesser GNU GPL 3.0	Eclipse PL 1.0
	Deployment	Standalone / war	Standalone / war	Standalone / war
	Artifacts storage	Filesystem	Filesystem	Filesystem
	Metadata storage	Filesystem	Filesystem	Filesystem
	Upload Artifacts	SI	SI	SI
	Maven Repository Support (Maven1/Maven2)	SI	SI	SI
	Integración con Eclipse (Update Site Proxying)			SI
	Mecanismo de Autentificación	SI	SI	SI
	Index Format	Lucene	DB metadata indexing	Lucene

Si bién no puede notarse muchas diferencias, la facil instalación de Nexus y su integracion con Eclipse le dieron un plus, sumado a que, en nuestra opinión, es el que mayor y mejor documentación tiene de los tres.

16. ANEXO C: CONFIGURACIÓN DEL NUEVO ENTORNO DE DESARROLLO INTEGRADO

Expondremos en este anexo los detalles más importantes de la configuración del entorno de desarrollo integrado que propusimos, en particular, las configuración de la interrelación entre cada herramienta. No nos detendremos en la instalación de cada una de ellas, ya que esa información está en la web de cada herramienta perfectamente explicada y fácil de ejecutar.

16.1. Pre requisitos

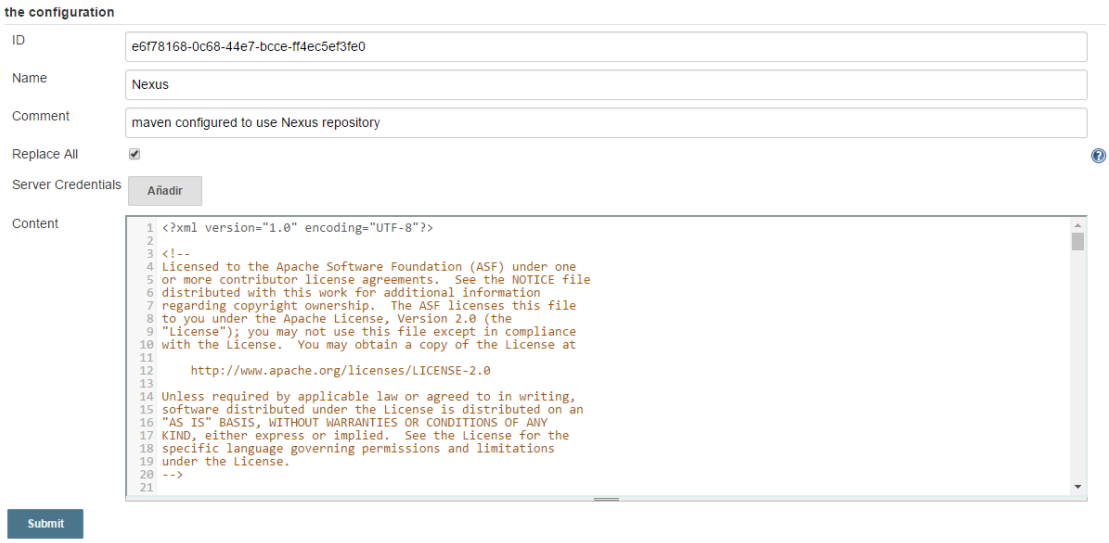
A continuación las herramientas requeridas que se deberán instalar en el servidor elegido para este fin. Inclusive se podría instalar cada uno de los servidores detallados abajo en su propio servidor físico si así se lo requiere.

- *JDK* de java (versión 7 o superior) (https://www.java.com)
- *Apache Maven 3.0* o posterior (https://maven.apache.org/)
- Servidor de IC *Jenkins* (https://jenkins.io/)
- Servidor de artefactos *Nexus Repository OSS* (https://www.sonatype.com/)
- Servidor de Inspección Continua *SonarQube* (https://www.sonarqube.org/)
- PlugIns de Jenkins:
    - *Config File Provider*
    - *SonarQube Scanner for Jenkins*
    - *nexus-jenkins-plugin-1.1.0-05.hpi (desde la pagina de Nexus)*

16.2. Configuración del Servidor de IC Jenkins

Una vez instalado el servidor Jenkins vamos a realizar algunas configuraciones para que pueda interactuar con las demás herramientas.

Primero debemos instalar el siguiente plugin: "*Config File Provider*", con el podremos configurar como debe actuar *Maven*, desde nuestro servidor de IC, con respecto a la gestión de dependencias y a la gestión de artefactos propios. Abrimos la configuración del plugin desde la consola de administración de Jenkins, y agregamos un nuevo archivo de configuración, que se ve como la imagen debajo. Lo importante aquí es el contenido, que será nuestro archivo de configuración de *Maven* (*settings.xml*) adaptado para funcionar con el repositorio de artefactos *Nexus* para usarlo como proxy y como repositorio de nuestros artefactos.



La parte del contenido que cambia con respecto al archivo *settings.xml* por defecto de Maven es la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <mirrors>
    <mirror>
      <id>nexus</id>
      <mirrorOf>*</mirrorOf>
      <url>http://localhost:8081/nexus/content/groups/public</url>
    </mirror>
  </mirrors>

  <profiles>
    <profile>
      <id>nexus</id>
      <repositories>
        <repository>
          <id>central</id>
          <url>http://central</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </repository>
      </repositories>
     <pluginRepositories>
        <pluginRepository>
          <id>central</id>
          <url>http://central</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>

  <activeProfiles>
    <activeProfile>nexus</activeProfile>
  </activeProfiles>

</settings>
```

Explicaremos cada sección de esta configuración:

<mirrors> Aquí se puede configura una lista de espejos (mirrors) que se utilizarán para descargar artefactos de los repositorios remotos de Maven, en el cual declaramos solo uno, la url del contenido público de Nexus (cuya configuración veremos más

adelante). Esto permite configurar Nexus como un proxy, en el cual todas las resoluciones de dependencias serán resueltas por medio de Nexus.

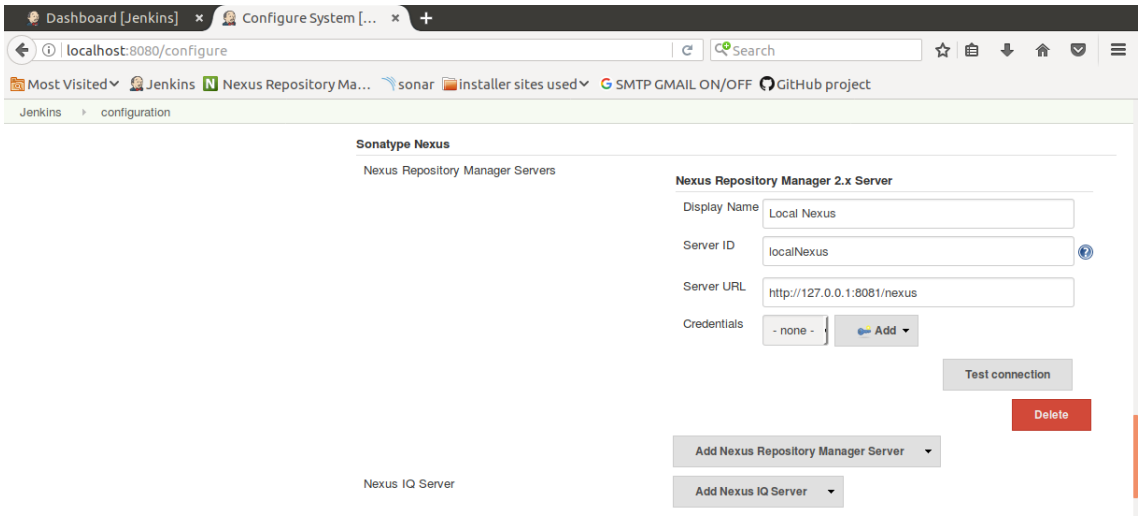
<profiles> Dentro de este tag se encontraran las configuraciones de los diferentes perfiles para construir el código, en nuestro caso, configuramos Maven para que pueda almacenar los artefactos propios en Nexus, ya sea en formato de instantánea (snapshots) o de un *release* versionado.

<activeProfiles> Aquí simplemente le decimos a Maven que el perfil que creamos este activo para todas las construcciones.

16.3. Configuración del Repositorio de Artefactos Nexus

La configuración por defecto es suficiente para empezar, lo importante esta detallado en la sección arriba donde le decimos a Jenkins que resuelva las dependencias utilizando nuestro repositorio de artefactos corporativo Nexus en las secciones "*mirrors*" y "*profiles*" en el archivo de configuracion de Maven *settings.xml*.

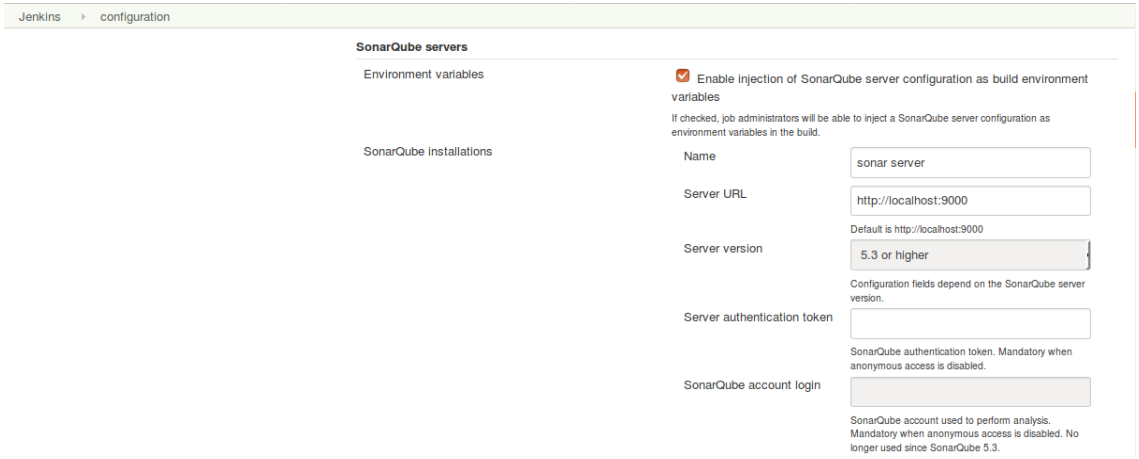
Para que Jenkins lo reconozca debemos primero configurarlo, luego de la intalación del plugIn de Nexus para Jenkins (*nexus-jenkins-plugin-1.1.0-05.hpi*) nos aparecerá la siguiente sección en la pagina de configuración de Jenkins. Allí le colocamos un nombre al servidor, y lo más importante, la URL del mismo.



De este modo, cada vez que Maven necesite resolver alguna dependencia, lo hará a travez del servidor de artefactos corporativo Nexus.

16.4. Configuración del Servidor de Análisis SonarQube

Al igual que Nexus, Para que Jenkins reconozca la instalación de SonarQube, debemos primero configurarlo, para ello, luego de la intalación del plugIn de Sonar para Jenkins (*SonarQube Scanner for Jenkins*) nos aparecerá la siguiente sección en la pagina de configuración de Jenkins. Allí le colocamos un nombre al servidor, y lo más importante, la URL del mismo. Asi de facil ya tenemos configurado SonarQube en Jenkins.

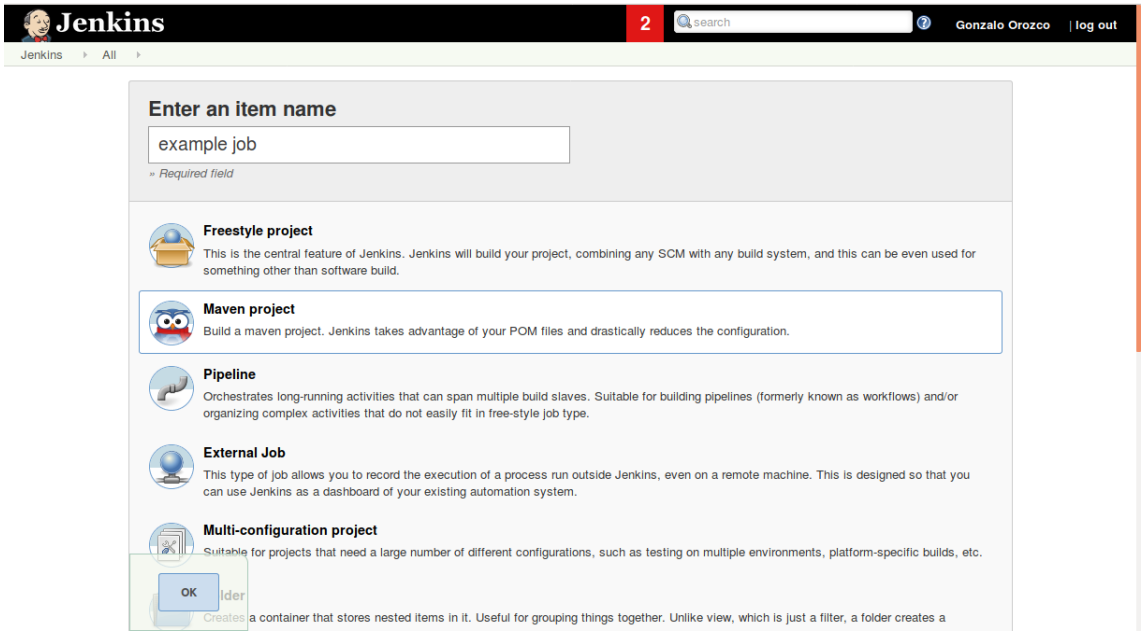


Luego, para utilizar el motor de análicis de código estático que proporciona el plugIn de SonarQube para Jenkins, solo debemos agregar el paso "*post build*" dentro de cada job (como se explica en la siguiente sección) según necesitemos que se corra o no.

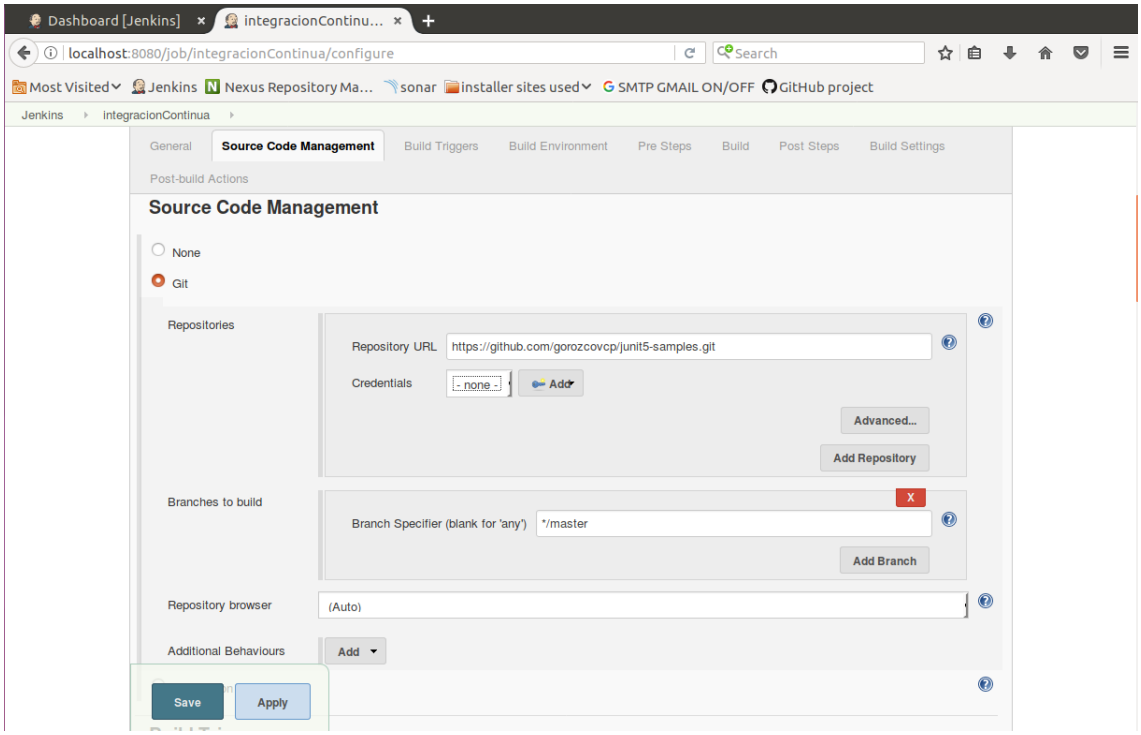
16.5. Crear un Job de Jenkins

La creación de una tarea en Jenkins es muy sencilla e intuitiva, y se podrán crear tantas como se quisiera, aunque lo que se suele usar es una tarea por repositorio de código y por entorno. Vamos a continuación crear una tarea que compile, resuelva las dependencias, ejecute los test y realice un análicis de código estatico cuando se comprometa código al repositorio en cuestión.

Desde la consola de Jenkins creamos un nuevo Job desde "*Create Item*". Se nos presentará el siguiente formulario donde pondremos el nombre del job y el tipo de job que queremos crear, en nuestro caso un "*Maven project*" como muestra la imagen a continuación.

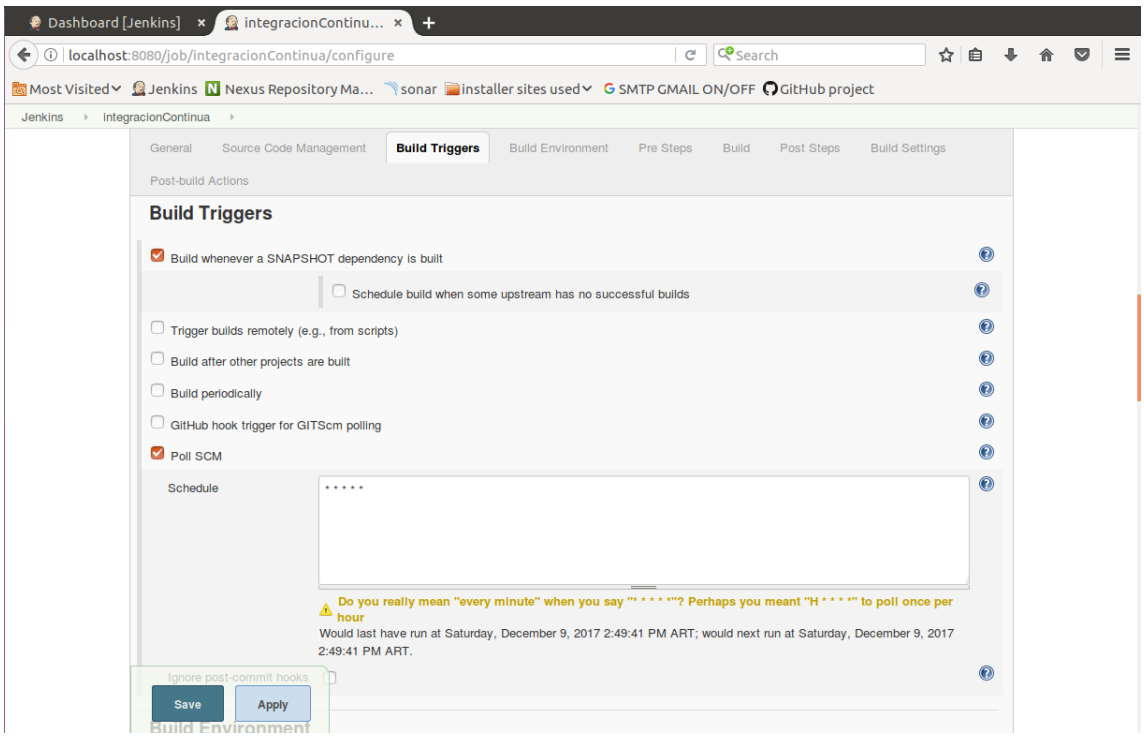


Luego de este paso se nos muestra toda la configuración del job para que podamos adaptarla a nuestras necesidadas. Vamos a resaltar las más básicas e importantes. Comenzamos por decirle al job de donde debe bajar el código. Abajo mostramos como se configura un repo Git de ejemplo.

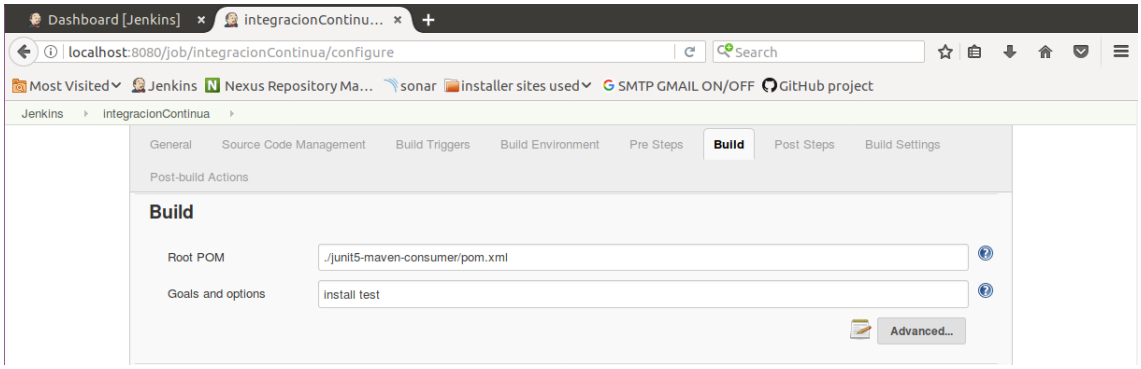


Una vez configurado el repositorio debemos establecer cuando Jenkins debería buscar cambios en el mismo, por ejemplo, podria hacerse una vez cada hora, o día, o

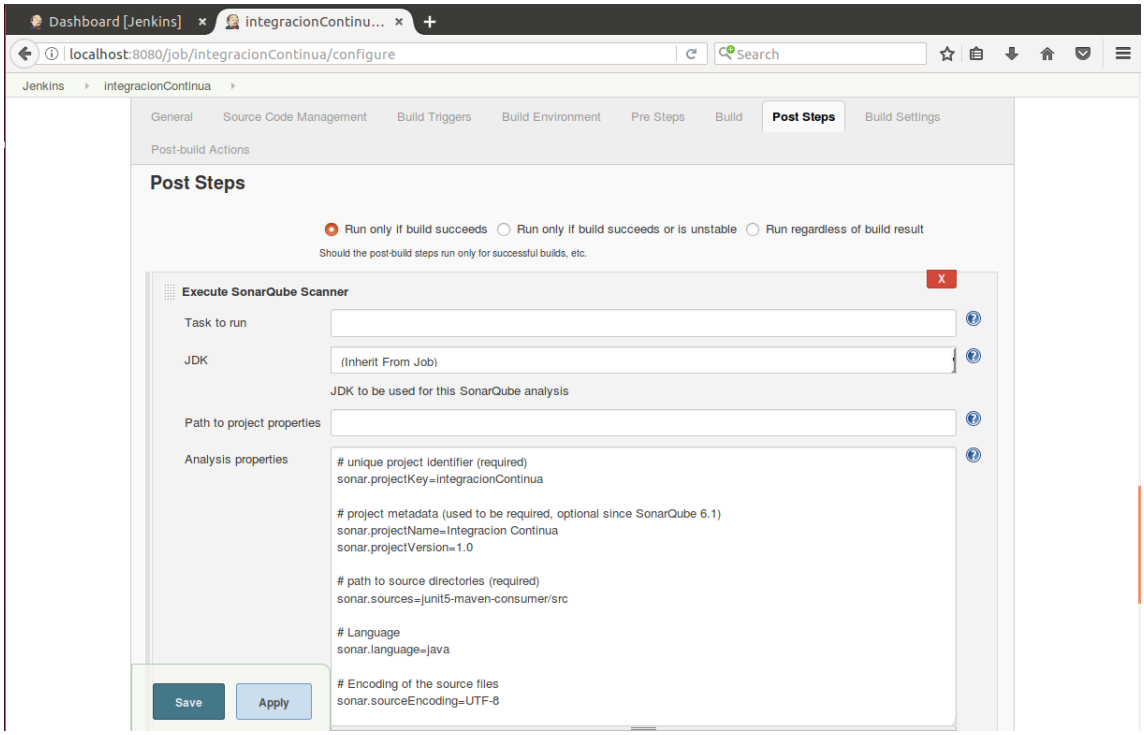
como en el ejemplo de abajo, a cada minuto (* * * * *), lo que se traduce a “cada vez que alguien compromete código al repositorio”.



Ahora debemos decirle a Jenkins que hacer cada vez que encuentre cambios en el repositorio, para eso nos valemos de Maven, y simplemente le ordenamos a Jenkins que que compile, resuelva dependencias y ejecute los test (*install test*) como se ve en la siguiente imagen.



Tambien disponemos de acciones que se ejecutan solo si el paso anterior es ejecutado correctamente (Build), en nuestro caso, queremos que se ejecute el análisis de código estatico con el plugIn de SonarQube. Los resultados luego se podran observar en el dashboard de SonarQube, en el servidor de Inspeccion Continua que instalamos.



Por último vemos algunas configuraciones útiles como la forma de comunicar, en este caso mediante un e-mail cuando el job falla o termina inestable.