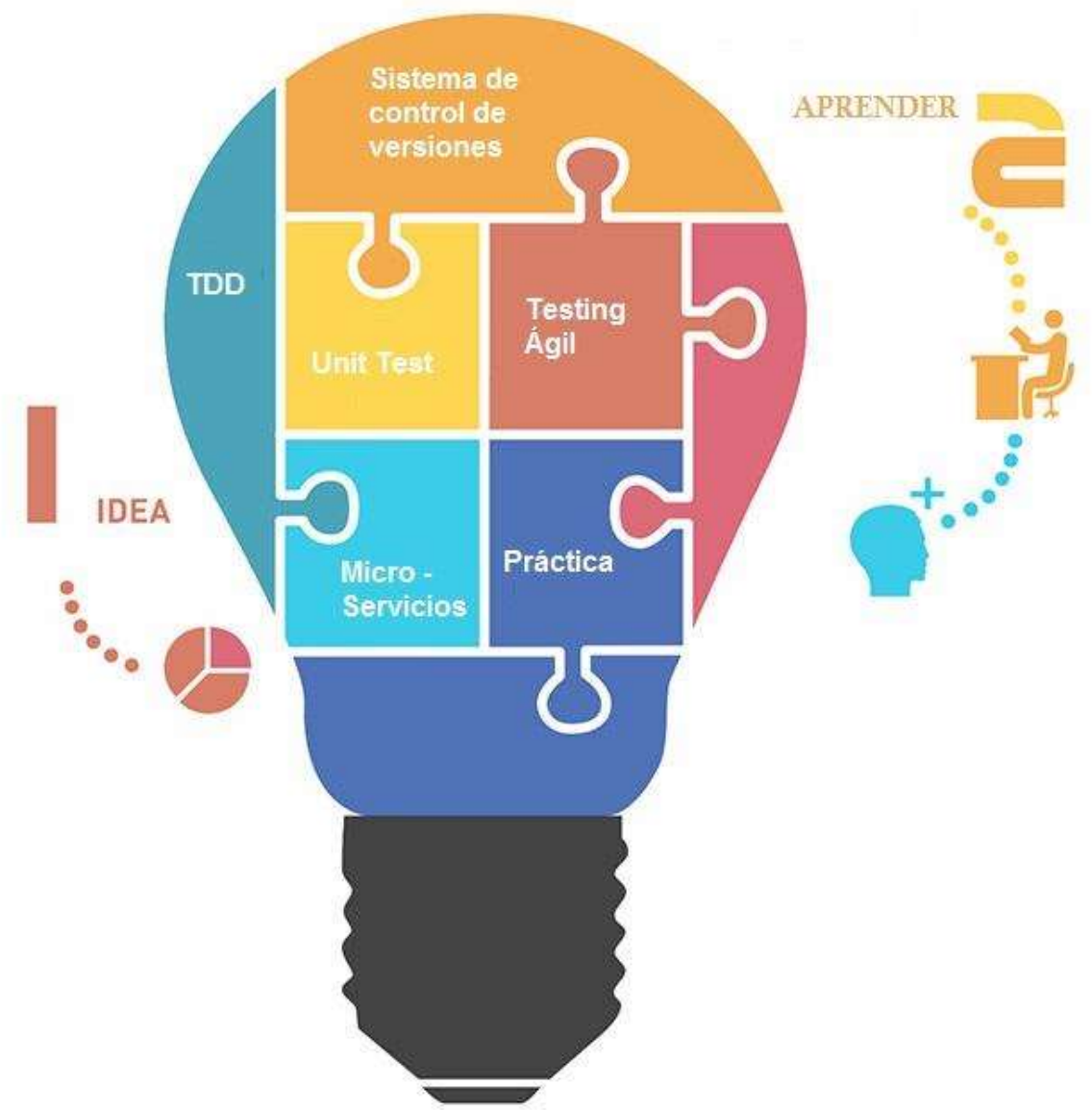
UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

DevOps, Integridad y Agilidad Continúa

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Unidad 2: Prácticas y técnicas de desarrollo de software ágil



Presentación:

La unidad número dos contiene una serie de prácticas, técnicas y herramientas que ayudarán al alumno a comprender los requisitos mínimos que deben tenerse en cuenta para la incorporación de la práctica de Integración Continua, la cual sienta las bases para DevOps.

Estos requisitos vinculan a las personas de los roles de Operaciones, Arquitectura, Desarrollo y Testing los cuales al entrelazarse posibilitará, tal como vimos en la unidad uno, el desarrollo incremental, mejorar la calidad del software y realizar entregas frecuentes a los usuarios.

Objetivos:

Que los participantes:

- Mencionen e identifiquen las principales características de los Sistemas de Versionado de Código, Orquestadores, Microservicios y Testing Ágil.
- Clasifiquen y analicen los componentes de cada propuesta.
- Identifiquen estrategias de versionado.
- Identifiquen los ciclos de TDD.

Bloques temáticos:

1. Versionado y estrategias de código

2.Unit Test & Test-Driven Development

3.Testing Ágil

4.Arquitectura de contenedores / Microservicios

5.Orquestador

Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

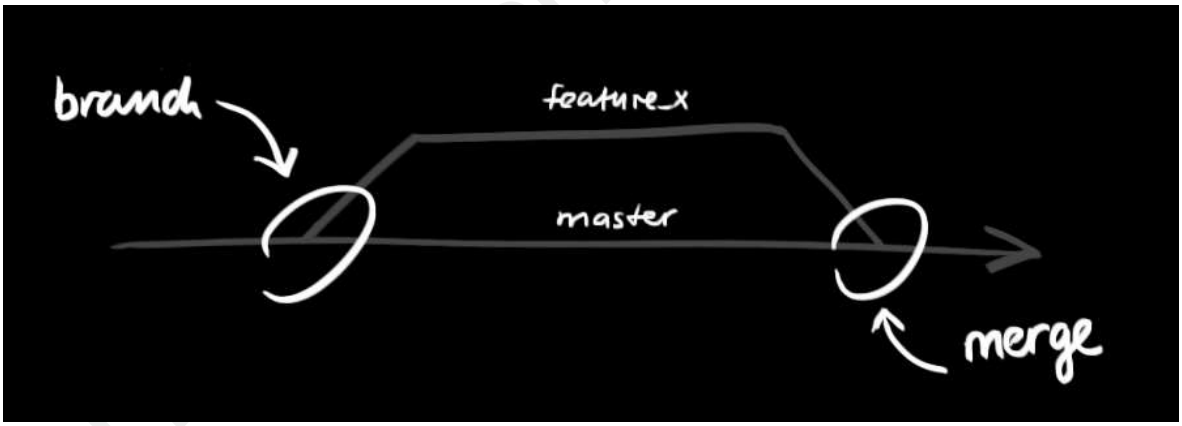
El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1.Versionado y Estrategias de Código:



Introducción al sistema de control de versiones

En esta sección haremos una breve reseña sobre los sistemas más utilizados actualmente para el control de versiones de aplicaciones, ya sea para el código fuente, documentación, base de datos, scripts de deploy, tests y otros materiales que requiera tener un historial de cambios. Además de contar con un historial de cambios, trabajar con esta tecnología habilita a los integrantes del proyecto a trabajar de una manera colaborativa.

Generalmente, una vez que los equipos de desarrollo crecen en cantidad de integrantes, se dificulta el trabajo sobre una misma línea de control de versiones. Veremos las ventajas y desventajas de distintas estrategias para controlar las versiones.

Además, se describirán distintas maneras de abordar estrategias de versionado de código fuente para entender las ventajas y desventajas de la integración (merge) y ramificación (branch) de porciones de código de una aplicación, distinguiendo qué se adecua mejor para la práctica de Integración Continua.

Branch

La operación de crear un branch en un sistema de control de versiones es una operación básica que se suele realizar. Un branch es la realización de una copia, en general, de la línea principal del código de la aplicación y se lo utiliza para operar de una manera independiente de dicha rama posibilitando el desarrollo paralelo.

Hay varias razones por las cuales los desarrolladores quisieran realizar un branch de su código, estas pueden ser por diversas razones, las cuales no quiere decir que cada una de ellas sea excluyente:

- Físicas. Para separar físicamente los archivos y generar independencia entre módulos, subsistemas y sistemas
- Funcional. Para separar incrementos de funcionalidad, configuraciones, bugs, releases, etc
- Ambiente. Separar en un branch el ambiente del sistema operativo. Esto es

especialmente útil para aplicaciones con sistemas operativos distintos, lo que posibilita un mejor manejo de las construcciones (builds) de la aplicación y ejecución de distintas plataformas.
- Organización. Hay ocasiones en que la generación de branch se realiza para organizar los equipos por funcionalidad. Por ejemplo, generar un branch por equipo el cual abordará un conjunto grande de funcionalidades

La creación de ramificaciones de código en muchas ocasiones dificulta la práctica de Integración Continua (práctica fundamental de DevOps), sobre todo cuando los equipos son grandes. Cuando se crea un branch se separa el código de la línea principal y generalmente se desea, dependiendo de la ocasión, que se vuelvan a unir (merge) por lo que la decisión de generar un branch requiere tener definido claramente un proceso y política de cuándo hacer el branch, quién puede operar sobre el mismo y quién lo promueve hacía integración con otra rama.

Hay dos estrategias de ramificación de código

- Ramificación temprana. Consiste crear branches por cada funcionalidad antes de comenzar a desarrollar, pretendiendo evitar grandes cambios en las ramas de código principal, de esta manera lo que se hace es diferir el dolor que causará el merge de cada branch con su origen de línea base.
- Ramificación tardía, al contrario de la anterior. Propone que sólo se tenga que realizar un branch en casos excepcionales, de esta manera se minimiza el dolor de realizar los merge. Lo único a considerar aquí es que se constituya el **hábito de subir los cambios todos los días a la rama principal**.

Merge

Como hemos visto con anterioridad, al crear un branch se desea que esa nueva rama se vuelva a unir a la línea principal de desarrollo, excepto por raras ocasiones en las que se quiere mantener separado. Esta unión de un branch a su línea de origen, en general, no es una tarea sencilla dado que generalmente hay conflictos a resolver.

Los conflictos que hay que resolver provienen cuando hay dos cambios en la misma porción

de código de dos branches distintos que se quieren unir. Cuando ocurre este conflicto se debe decidir qué cambios se desean mantener para resolver el inconveniente y se debe confirmar realizando un nuevo envío al repositorio para que la rama que contenía el conflicto quede en un estado estable. Por lo tanto, **mientras más perdure en el tiempo una rama en donde se esté trabajando más probabilidad de dificultades se tendrán al realizar un merge**. Es decir, que al generar branches menos cercanía se estará de la práctica de Integración Continua, ya que justamente lo que está faltando en ese proceso es la integración de las distintas porciones de una manera continua. Uno de los requisitos para realizar **Integración Continua** es que los desarrolladores realicen un **check-in** contra la rama principal **al menos una vez al día**.

Estrategias de versionado de código

Se podrá apreciar que existe una tensión peculiar entre los sistemas de control de versión distribuidos y la práctica de Integración Continua, ya que los commits se realizan en modo local y recién se realizaría una integración cuando se suban los cambios al repositorio remoto, por ende y por definición no es Integración Continúa hasta que se publiquen los cambios en el servidor (esto es cierto si se utiliza Git). Por lo tanto, **se estará haciendo Integración Continúa si los cambios se publican al menos una vez al día**. En esta sección abordaremos distintas estrategias de versionado de código, mencionando sus características, ventajas y desventajas.

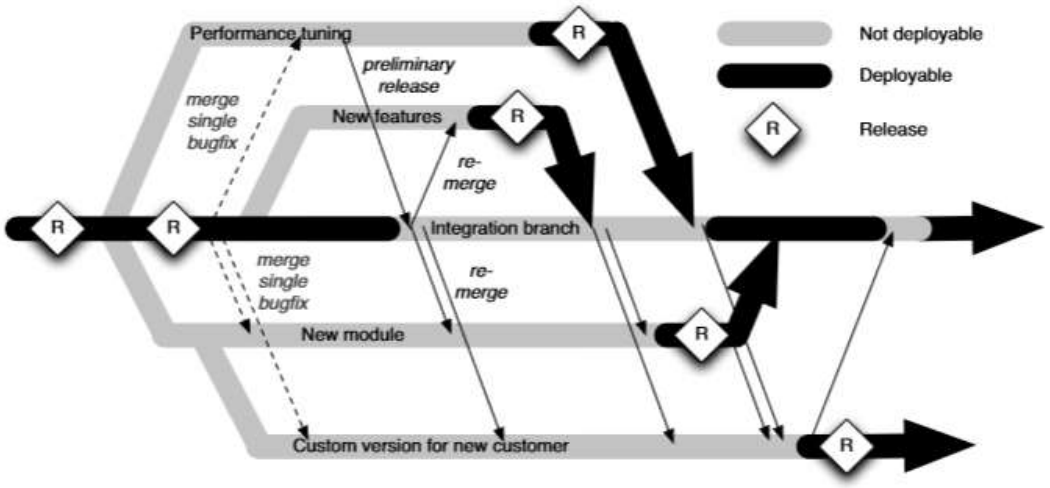
Una de las mayores consecuencias que se observó con la popularidad del uso GIT es lo fácil y rápido con los que se crean los branches y se realizan los merges. Esto provocó que las empresas y sus equipos de desarrollo tiendan a crear grandes cantidades de branches resultando en que cada una de ellas estén en estados no estables por largos periodos de tiempo, lo cual trae aparejado principalmente dos consecuencias.

La primera es que al generar muchas ramificaciones de la funcionalidad se podría pensar que facilitará la construcción de una manera aislada, pero mediante este abordaje no se tiene en cuenta que ese branch luego se debe integrar con la línea principal, por lo cual probablemente habrá al menos dos ramas que evolucionarán en el tiempo, acumulando cambios para una integración. Mientras más tiempo se tiene una rama o branch “vivo” sin integrarlo a la línea principal, más dificultad se tendrá en realizar la integración debido a la

gran cantidad de código acumulado y los posibles conflictos que surjan de modificar las mismas porciones código y/o las posibles dependencias. El impacto de lo mencionado es una gran cantidad de pérdida tiempo y recursos en resolver conflictos de integración.

Por último, mientras más tiempo se demore en realizar un merge de un branch menos se estará practicando la Integración Continua (CI). Es decir que mientras se tenga branches sin integración a la rama principal, en un estado no disponible para implementar, se está imposibilitando lo que la Integración Continua (CI) se supone que resuelve, que es que la aplicación esté lo mayor posible en un estado listo para implementar. Desde el punto de vista de Lean y también de DevOps, los branches en un estado no implementable están generando desperdicio ya que no posibilitan la entrega de valor al cliente.

En la siguiente imagen se muestra una estrategia bastante común en las empresas. No quiere decir que este tipo de estrategia de versionado sea el incorrecto, pero se debe tener en cuenta lo que afecta su implementación. En la siguiente imagen se puede apreciar mejor que casi todas las ramas poseen periodos largos de tiempo en los que no se está integrando y tampoco se encuentran en un estado potencialmente a implementar.



Ejemplo de un pobre control sobre branches. Jez Humble, David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.

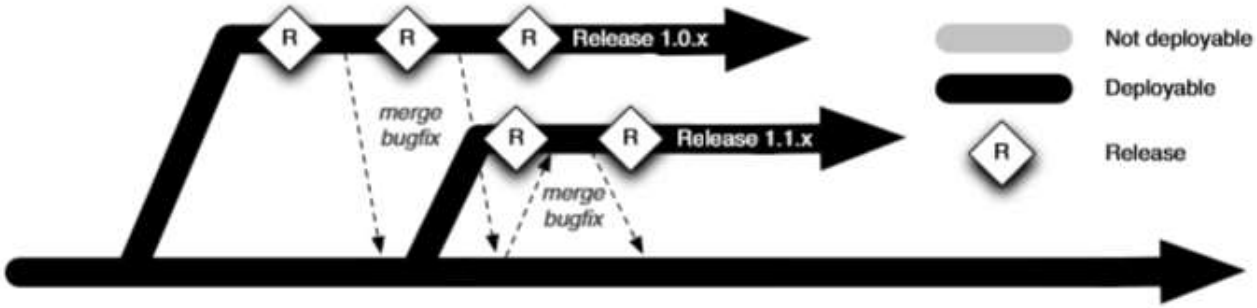
Branch por Release

Una estrategia de versionado fuertemente recomendada en la industria y para poder implementar de una manera menos dolorosa las integraciones es la de mantener ramas vivas y en estado de implementación el mayor tiempo posible. De esta manera se evita que los equipos de desarrollo tengan que desperdiciar tiempo en reparar las integraciones de grandes cantidades de código ya sea en un branch, o al integrar la solución de un defecto, reduciendo riesgos y demoras en los calendarios comprometidos.

Claramente para utilizar una estrategia de versionado de branches por release se requiere una gran coordinación y disciplina por parte del equipo de desarrollo. El no poder hacerlo podría indicar que la aplicación tiene el código fuente poco estructurado, con falta de encapsulación y fuertemente acoplado, imposibilitando un desarrollo orgánico e iterativo.

La idea de realizar un branch para aislar el desarrollo de la funcionalidad podría parecer bastante descabellado, sobre todo en equipos con grandes cantidades de desarrolladores. Lo que hay que tener en cuenta es que las grandes cantidades de ramas con una cantidad considerable de código poseen una alta probabilidad que al integrarse

se tenga que desperdiciar horas o días en realizar una estabilización para reparar la integración, incrementando el riesgo de introducir defectos y demoras en el proyecto. Por esta razón se debe tener coordinación y disciplina con los integrantes del equipo de desarrollo para que al menos una vez al día realicen una integración. De esta manera, al generar un commit se ejecutará un build de la aplicación (la conversión del código fuente a binario) y se ejecutarán las pruebas automáticas necesarias que determinarán si el build recientemente ejecutado se lo considera exitoso.



Ejemplo Release branching. Jez Humble, David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. 1era edición

La recomendación para equipos medianos a grandes es que se realice una división teniendo en cuenta los módulos o componentes de la aplicación, de esta manera se tendrá un diseño de arquitectura menos acoplado posibilitando el desarrollo iterativo y se reduce el riesgo al incorporarse modificaciones. Las integraciones siempre existirán, este abordaje de una aplicación mediante componentes permite que sea menos doloroso el desarrollo concurrente, ya que los inconvenientes en la integración serán independientes en el componente que se esté modificando.

A la hora de implementar CI (Integración Continua, en inglés Continuous Integration) es fundamental distinguir las distintas estrategias de versionado. **La opción que más se recomienda** es la de desarrollar e integrar sobre una rama principal. Por supuesto que se puede optar por branch por features, por equipos, por ambientes u otra personalizada. Cada una de ellas poseen ciertas características que dependiendo de la organización y el equipo que las adopte se podrá obtener más o menos beneficios. Si es importante realizar

un análisis sobre esta decisión y no copiar estrategias porque en otro lado funciona.

Para más información sobre estas estrategias de versionado se recomienda consultar el capítulo 14 del libro de Jez Humble y David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.

Desarrollar sobre una línea principal (develop on mainline)

El desarrollo sobre una línea principal de código en una aplicación es el método que más propicia la práctica de Integración Continua. Además, es una excelente práctica para concebir un desarrollo incremental, ya que los desarrolladores tendrán que ir agregando pequeños cambios al código de manera que no afecte el comportamiento de la funcionalidad actual de la aplicación y al mismo tiempo ejecutar toda la batería de pruebas que correspondan.

Con este tipo de práctica, generar un branch es la tarea menos habitual. En caso de querer realizar grandes cambios se requiere una mayor labor en coordinar y planificar, dado que los cambios deben ser incrementales, pequeños y pasar de capa en capa de la arquitectura de software. Para desarrollar sobre una línea principal se requiere de una buena arquitectura y módulos desacoplados.

Una observación de este patrón es que no todos los cambios que se suban a la línea principal podrán serán implementados, sino que en ciertas ocasiones se debe recurrir a implementaciones de funcionalidad de un modo oculto (feature toggles). Este aspecto posee algunas críticas, aunque nuevamente el mayor beneficio es que todo el tiempo la aplicación contendrá código que fue integrado continuamente, es decir que no sólo no se ha roto, sino que también se ejecutaron las baterías de pruebas.

Muchos equipos que utilizan este patrón también lo complementan con otro llamado **Branch por Release**.

Branch por Feature

Este patrón de creación de branches está motivado para que los equipos puedan desarrollar en simultáneo distintas funcionalidades e integrando su rama al finalizar su trabajo. Por cada **Historia de Usuario**[1] o necesidad se crea un branch sobre el que se desarrollará.

[1] Historias de Usuario. Ver archivo anexo de links

Una vez finalizado el desarrollo se realiza un merge a la rama principal. Este enfoque se popularizó bastante en la actualidad, pero conlleva algunos inconvenientes si no se tiene algunas consideraciones.

Al divergir la rama principal en varios branches se corre el riesgo de que no se integren con la rama principal en un periodo de tiempo largo provocando que el resto de los programadores no obtengan los últimos cambios que realizó el resto del equipo. Este factor vulnera fuertemente el principio de Integración Continúa ya que no se estarían subiendo cambios al menos una vez al día, sino que se tendrían múltiples branches con largos tiempos de vida, los cuales sus cambios no se manifiestan entre sí. La recomendación para que este patrón sea efectivo y no dificulte la práctica de CI, es tener una buena disciplina con las siguientes actividades:

- Cada cambio que se realiza en la línea principal se debe propagar a las ramas.
- Los branches no deben superar los pocos días de vida, nunca más de una iteración.
- En lo posible, sólo se debería realizar un merge de una rama a la línea principal cuando un tester aceptó el cambio.
- Es muy recomendable realizar code review de los cambios a integrarse. Tal vez en forma de **pull request** para tener la capacidad de rechazar el futuro merge.

El patrón de branch by feature (branch por feature) es recomendable para equipos de chicos a medianos, con desarrolladores experimentados. Se recomienda, nuevamente, tener cautela con este enfoque ya que no considerar cuestiones como las anteriormente descritas podría convertirse en un anti-patrón de Integración Continua. No habría: integración todos los días a la línea principal, la integración de todos los branches al momento cercano de lanzar un release podría ser muy dolorosa por la cantidad de conflictos que probablemente surjan en la integración. Se sugiere comenzar con el patrón “Desarrollar en la línea principal” o “Branch by Release” y luego, a medida que se gana más madurez y disciplina intentar con otros patrones. De lo contrario sólo se pretenderá evitar el dolor que genera realizar los merge y probablemente se pierda gran cantidad de tiempo y recursos al realizar una integración tardía.

Branch por equipo

La motivación de utilizar este patrón surge al tener equipos muy grandes, con trabajos en flujos simultáneos de funcionalidad y mantenimiento de la línea principal. Adoptar esta

estrategia, sin los recaudos necesarios, podría provocar los mismos problemas que se mencionaron con anterioridad. La principal diferencia radica en que cuando un merge se realiza en la rama principal este mismo cambio se debe propagar desde la rama principal hacia el resto de los equipos. El merge sólo se debe realizar cuando la rama del equipo están lo suficientemente estable.

Mediante este enfoque se permite trabajar con equipos grandes y además mantener la rama principal siempre en un estado potencialmente implementable, pero no así con las ramas por equipos. El inconveniente principal que se enfrenta aquí es que no todos los equipos terminan al mismo tiempo por ende la rama principal no va a poseer las funcionalidades completas. Los equipos sólo realizan el merge cuando su rama está terminada en términos de la cantidad de funcionalidades, de esta manera se mantiene estable la rama principal.

Para que este patrón tenga efectividad, los equipos se deben dividir con una cantidad pequeña de integrantes. Cada uno de ellos van a realizar los check-in contra la rama del equipo y cada vez que se realice un merge entre los cambios del programador y la rama se deberán correr toda la batería de pruebas (unit test, component test y acceptance test). Debido a esto, cada rama por equipo tendrá su propio pipeline de desarrollo para determinar en qué momento la rama está en un estado estable para que pueda ser integrada a la rama principal. En el caso de poseer defectos, lo más factible es que el equipo mediante la técnica de cherry picking resuelva el bug y lo integre directo a la rama principal. De lo contrario se debería prestar cuidado a que la rama esté estable, ya que posiblemente se avanzó con otras funcionalidades durante la detección del defecto y luego hacer un merge de la rama completa a la rama principal, con todos los riesgos que esto implicaría. En el caso de que no se utilice un sistema de control de versiones que no posea dicha técnica se tendría que evaluar la posibilidad de crear un branch para corregir el defecto. Este tipo de patrón es el menos deseado, ya que se estaría divergiendo toda una rama de un equipo y no de una simple funcionalidad.

2. Unit Test & Test Driven Development

Una de las características que más cambia en un proyecto de software es el código y desde la agilidad se abrazan dichos cambios. Para garantizar calidad en las modificaciones de código, se utiliza Unit Test permitiendo obtener feedback rápido y concreto en el caso de

que algún cambio impacte negativamente, de esta manera se obtiene información exactamente de donde se produjo.

Observemos el común del trabajo de un desarrollador. Trabaja sobre una aplicación agregando o modificando algunas líneas nuevas de código. Revisa los valores de las variables mediante impresiones en pantalla o bien en un debugger. Si encuentra alguna anomalía en las reglas de negocio que acaba de programar, lo arregla y lo vuelve a revisar y descarta las líneas de código que utilizó para probar. Al introducir un cambio en una aplicación ¿Cómo sabe el desarrollador si el nuevo código, ya sea modificado o eliminado, funciona correctamente o no rompe algo en otro lado, al no tener un conjunto de pruebas automatizadas o manuales que lo comprueben? No se tendrá la posibilidad de conocer si introdujo un defecto. Por esta razón la técnica de Unit Test se la considera como una excelente técnica de feedback para los programadores.

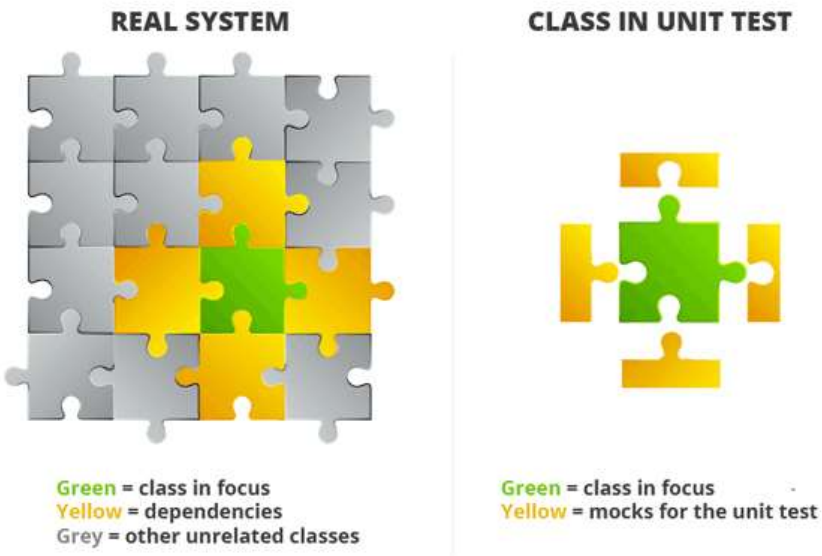
En la agilidad el enfoque es distinto, esas porciones de código que se utilizan para comprobar el código se las aprovecha convirtiéndolas en pruebas y se las automatiza. De esta manera cada vez que se introducen cambios se ejecutarán todas las pruebas que los desarrolladores fueron construyendo, garantizando que los nuevos cambios no introduzcan defectos. En el caso de que las reglas de negocio cambien se deben actualizar las pruebas que las comprueban para mantener la consistencia. **Las pruebas forman parte del producto, pero no del código productivo.**

Para trabajar con Unit Test se debe usar un framework acorde al lenguaje que se está utilizando, por ejemplo: Java, C#, Ruby, etc. El framework contiene un conjunto de utilidades para la creación de pruebas y para su ejecución. Para esta última, en general, se permite definir si se desea ejecutar todas las pruebas o bien alguna porción en particular. Para cada lenguaje hay frameworks de Unit Testing que se los conoce como la familia xUnit Test.

Unit Test se lo puede considerar como una fuente de información de rápido acceso para diagnosticar problemas cuando se lo incorporar a la práctica de integración continua.

Si un desarrollador introduce un cambio y falla, es más probable que lo solucione en menor tiempo si la falla es sobre el cambio que acaba de introducir y no sobre resto del código existente. Al fallar el código antiguo, el programador se tiene que tomar el tiempo de buscar en dónde se encuentra el error y corregirlo, y además verificar que esa corrección en el antiguo código no produzca nuevas fallas.

Una de las dificultades que más se pueden apreciar a la hora de definir qué incluye un Unit Test es la diversidad de definiciones que se encuentran para enmarcar cuánta porción de código se debe probar.



Al existir esta diversidad no hay una respuesta concreta al momento de decidir qué se incluye en el test y qué no. Lo que se debe tener en cuenta a la hora de construir es que las pruebas unitarias deben ser rápidas, para que el desarrollador pueda obtener feedback instantáneo y actuar con mayor información ante inconvenientes. Si la ejecución no es rápida, los desarrolladores tienden a dejar de escribir las pruebas. Incrementar el número de tests unitarios incrementa el tiempo de espera para saber si el Build se ejecutó correctamente, por lo cual realizar la práctica de CI conlleva mantenimiento por parte del equipo.

Teniendo en cuenta el aspecto de la velocidad, se suele determinar que cada Unit Test debe ser aislado de su entorno y no tener acceso a dependencias externas, tales como acceso a base de datos, lectura o escritura en disco o bien consumir un servicio. Esto no quiere decir que no se deban realizar pruebas contra estas dependencias, sino que esos tipos de test se debieran utilizar para comprobar el comportamiento en las distintas interacciones de nuestro código.

Las pruebas unitarias son comprobaciones de porciones aisladas de código y de una única responsabilidad (por ello emerge un buen diseño y código legible), de esta manera el código y el sistema se puede probar sin tener que conectarse a recursos externos. En el caso de necesitar establecer una relación con alguna dependencia para poder evaluar algún funcionamiento particular se puede utilizar la técnica de Mock[2] y Stub[3] de objetos. Mediante la técnica de Mocking y Stub lo que se logra es aislar la porción de código del exterior y simular el comportamiento que tendría que tener, además de ganar velocidad en la ejecución.

Lo antes visto lo podemos resumir con un principio que se recomienda que sigan la conformación de las pruebas de Unit Test. Este principio se lo conoce con el acrónimo F.I.R.S.T

Principio F.I.R.S.T

Este principio hace referencia a un conjunto de reglas que mejoran la escritura de Unit Test. (Fast, Independent, Repeatable, Self-validating, Timely)

Fast: Las pruebas deben ejecutarse rápido. Si la ejecución de las pruebas es lenta provoca que se tienda a no ejecutarlas y si no se ejecutan entonces no se podrán hallar los problemas de una manera rápida para solucionarlos, por lo que también se tenderá a no escribir pruebas y así no se tendrán suficientes pruebas para tener más cobertura de código (cantidad de código productivo que está bajo la protección de Unit Test).

Independent (independientes): Las pruebas no deben depender unas de las otras, es decir que la ejecución de una prueba no debe condicionar la ejecución de la próxima. Es decir, deben poder ejecutarse en cualquier orden. Cuando hay dependencia en la ejecución, esta se manifiesta en cadena al fallar la prueba, dificultando el diagnóstico del problema.

Repeatable (repetibles): Las pruebas se deben poder ejecutar en cualquier ambiente. Es decir que se deben poder ejecutar en el ambiente del desarrollador, en el ambiente de QA, en un ambiente de producción o bien sin conexión de red. Si la prueba no se puede ejecutar

[2] Mocks: Ver archivo anexo de links
[3] Diferencia entre Mock y Stub: Ver archivo anexo de links

en cualquier ambiente siempre se va a tener una excusa para no ejecutarlos o no hacerlos.

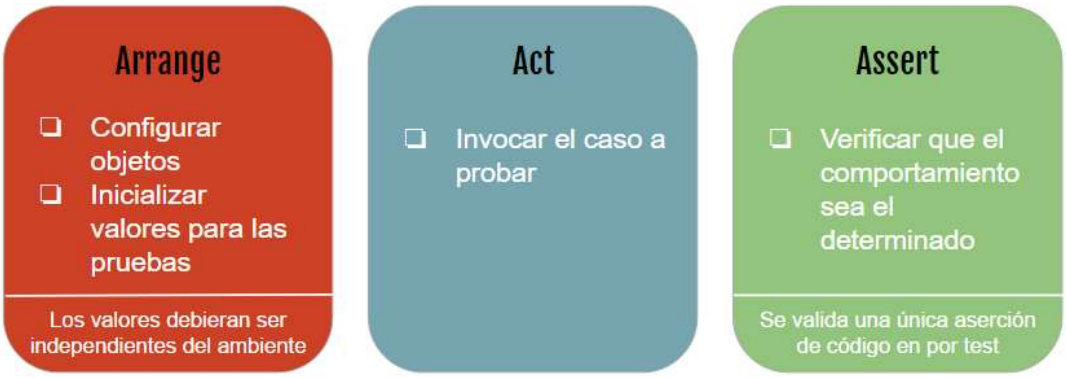
Self-Validating (autovalidación): Las pruebas deben tener un valor a probar que sea binario, falso o verdadero. La prueba por sí misma debe indicar el error, no se debiera tener que revisar grandes registros del sistema para determinar dónde se encuentra el error. Si la prueba no se valida por sí misma entonces requiere el análisis de una persona por ende el resultado de la prueba será subjetivo. Esto es muy importante a la hora de escribir una buena prueba.

Timely (de manera temprana): Las pruebas deben ser escritas lo más temprana posibles, justo antes de la escritura del código que será productivo y deben poder cubrir lo más posible los escenarios.

F AST	➢ Se deben ejecutar rápido
I NDEPENDENT	➢ Deben aislar el resultado y correr de manera independiente (ya sea en suite o individual)
R REPEATABLE	➢ Se debe poder repetir la ejecución sin importar el orden. Determinístico
S ELF-VALIDATION	➢ El desarrollador debe tener la disponibilidad de validar el test por sí mismo
T HOROUGH AND TIMELY	➢ Debe cubrir los escenarios de manera completa y de ser posible escribir la prueba antes que el código

Patrón AAA

Se recomienda que la escritura de código de Unit Test esté basada en ciertos patrones (se mencionan algunos de ellos en la sección de TDD) y principios.
Para que el código de Unit Test sea legible para otros desarrolladores es conveniente que posea una estructura común. Dicha estructura corresponde al patrón AAA (por sus siglas en inglés: Arrange, Act, Assert). Este patrón facilita la legibilidad de la prueba, así como también su mantenimiento. Al estar escrito de una manera que es estándar posibilitará a otros desarrolladores que necesiten menor tiempo en comprender y mantener la prueba.



Los principales beneficios de la utilización de Unit Testing son los siguientes:

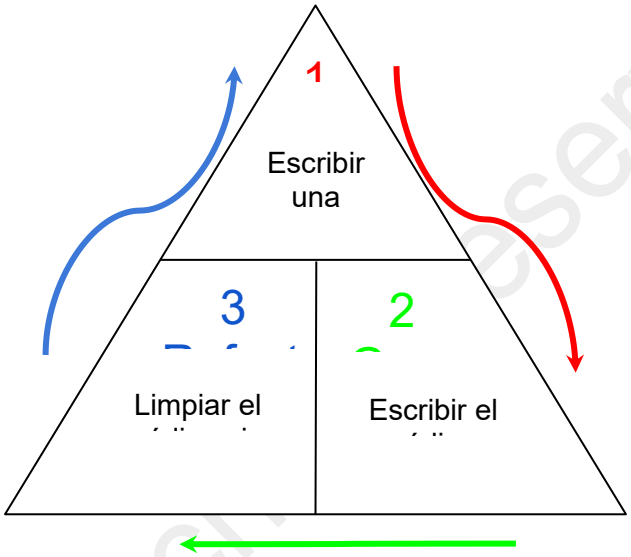
- Provee feedback casi instantáneo. El código se prueba reiteradas veces a medida que se introducen nuevos cambios. Se identifican y se solucionan problemas rápidamente.
- Hace el código más robusto. Se piensa en el desarrollo basado en la prueba que se tendrá que hacer teniendo en cuenta las pruebas exitosas, las que no y también casos de prueba de excepciones.
- Incrementa la confianza en el sistema que se está desarrollando, ya que al ejercitar las pruebas en cada cambio se está en conocimiento si hay alguna falla en las pruebas que ya fueron escritas.
- Se utiliza para documentación de casos de prueba dentro del código, así como también por ejemplo para el contrato de servicio con una API.

TDD - Test Driven Development

Las siglas TTD refieren al desarrollo guiado por las pruebas (test driven development por sus siglas en inglés), es una práctica que utiliza la técnica de Unit Test y distintas implementaciones de automatización para su realización. TDD permite generar código de buena calidad y legibilidad, ya que resulta en mejores diseños debido a que se deben

pensar pequeñas porciones de código que se deben probar. Además, proporciona documentación de pruebas al proyecto directamente en el código fuente.

TDD tiene como objetivo no sólo introducir las pruebas lo más temprano posible en el ciclo de desarrollo de software, sino que tiene como objetivo el desarrollo emergente del diseño. Veamos el flujo del proceso que define esta práctica.



El ciclo, comienza escribiendo una prueba para un determinado requerimiento que debe ser desarrollado (utilizando un framework de la familia xUnitTest), la cual tiene que fallar. Si la prueba no falla es que ya existe código de producción (no de prueba) que está provocando que la ejecución sea exitosa. Hasta aquí vemos el objetivo más claro de que primero asegurarnos de tener escrito la prueba antes de comenzar a desarrollar, pero veremos que este no es todo el beneficio. Al escribir solamente el código de prueba sin escribir el código productivo lo que nos va a **guiar** es a escribir el código faltante para que la prueba sea exitosa. Es aquí en donde vemos el beneficio más importante, el **diseño** de cómo tiene que funcionar nuestro código lo hemos definido en la prueba que especificamos primeramente y no en el código final. Hay que destacar que, al escribir primero la prueba y luego el código productivo, el diseño que emerge hará que nuestro software tenga el diseño adecuado para que pueda ser probado, ya que no todos los diseños de solución poseen la misma facilidad para ejecutar pruebas. Por otro lado, esta

técnica cambia el enfoque en que los programadores piensan el software, primero la prueba y luego la solución que satisface dicha prueba.

Subyacente a la posibilidad de que se ejecuten pruebas eficientes existen patrones de diseño de software que facilitan esta tarea (generalmente en programación orientada a objetos). Uno de los patrones de diseño que recomiendo utilizar el de **Inyección de Dependencias**, otro con el cual se pueden obtener buenos resultados en cuanto a calidad de código y arquitectura es el patrón **S.O.L.I.D.** (Robert C. Martin) que es un acrónimo de sus siglas en inglés: Single Responsibility, Open Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) y por último el patrón **DRY** (no te repitas, por sus siglas en inglés: don´t repeat your-self), que hace referencia a reconocer que cada parte del sistema debe tener la autoridad de una sola y unívoca representación (Andy Hunts)

Resumiendo, primero escribimos la prueba y luego la ejecutamos para que falle, de aquí el color rojo de la imagen.

El segundo paso es escribir el código que necesita la prueba para que su ejecución sea exitosa. Pero escribir dicho código no significa, necesariamente, que complete satisfactoriamente la totalidad de la prueba en la primera vez, sino que se abordara algunos puntos que fallaron en la prueba. Por ejemplo, si al crear una prueba se encontraron nueve puntos de fallo, (podría ser por ejemplo la inexistencia de una clase o la invocación de un atributo o de un método que no existe, etc), la propuesta de TDD es que se aborde punto por punto escribiendo sólo el código necesario para que la ejecución de la prueba sea exitosa. Si la prueba no es exitosa se debe reescribir el código de implementación. Para más información sobre estos términos puede consultar sobre Programación Orientada a Objetos (POO o OOP por sus siglas en inglés)

Por último, una vez que se tiene la prueba escrita sólo con el código necesario para que dicha prueba se ejecute correctamente se comienza con la refactorización. Este último punto consiste en cambiar el código escrito en pos de que sea más legible, mejorar lo suficiente el diseño, su robustez, pero sin modificar su comportamiento exterior. El diseño inicial del código ya se generó en la prueba que se escribió en el primer paso (por ejemplo, la clase y la firma de sus métodos, no sólo su invocación), en el segundo nivel de la práctica de TDD se generó el código de implementación necesario para que la prueba pase (por ejemplo, se creó la clase y la firma del método). Y en este último paso se utilizan las técnicas apropiadas de encapsulación, eliminación de código redundante, y

demás mejoras sobre el código de implementación teniendo en cuenta que al realizar todas estas modificaciones la ejecución de la prueba tiene que pasar exitosamente.

Una vez embellecido el código y que todas las pruebas son satisfactorias (“luz verde”) se toma una nueva funcionalidad y se comienza con el proceso. Más adelante veremos que las pruebas que se generan con la práctica de TDD mezclan Pruebas Unitarias (Unit Test) y Pruebas de Componentes.

Los beneficios de utilizar esta práctica son variados y muy poderosos.

- Se introducen las **pruebas al inicio** en que se comienza con el desarrollo.
- El diseño de la **arquitectura emerge** de un código diseñado para ser probado.
- El **código** es altamente **reutilizable**.
- **Aumenta la calidad** del software, ya que el número de defectos disminuye.
- La **documentación** del proyecto está **embebida** en el código, ya que las pruebas formadas mediante TDD son la representación de los requerimientos.
- Las pruebas se agregan al proyecto de una manera **incremental**.
- **Motivación** de los desarrolladores. Se tiene más grado de certeza que lo que se programó funciona.

Las principales dificultades que se pueden observar al querer incursionar en esta práctica son las siguientes:

- La curva de aprendizaje. Dominar esta práctica lleva un largo tiempo, no es algo sencillo.
- Poca cantidad de desarrolladores en el mercado que la dominan.

3.Testing Ágil

Testing Ágil es un concepto que surge de la necesidad de abordar la actividad de probar el sistema o el producto de una manera temprana para acompañar el ciclo de desarrollo de software iterativo e incremental, es decir que al finalizar una iteración se debe entregar el delta de desarrollo y el resultado de la ejecución de las pruebas. La diferencia más notoria que se puede apreciar es que este abordaje difiere bastante del paradigma tradicional en el que las pruebas se ejecutan una vez que finaliza el desarrollo.

En Testing Ágil las pruebas se introducen lo más rápido posible dentro del ciclo de desarrollo de software. En una cultura DevOps si se desea entregar rápidamente software de valor al cliente/usuario se debe asegurar un mínimo de calidad, por lo que la naturaleza incremental del desarrollo nos llevará a probar hacia atrás (prueba de regresión) lo que ya hemos probado anteriormente, lo cual nos asegura que los nuevos cambios no están dañando el sistema actual. Para sustentar el desarrollo iterativo e incremental se requieren ciertas prácticas y técnicas de desarrollo, algunas de ellas son las que vimos en esta unidad sobre Unit Test y TDD, y otras las veremos en las siguientes unidades, como por ejemplo los distintos tipos de despliegues.

Nuevamente, el enfoque que se propone para la realización de la actividad de testing es que sea temprano y continuo, propiciando un ambiente de colaboración que tendrá como principal actor al tester en relación directa (siempre que sea posible) con el usuario o el negocio y con los desarrolladores en post de proporcionar más calidad al producto que se elabora.

Algunos equipos ágiles no distinguen entre sus integrantes si son testers y desarrolladores enfatizando tener la mirada en el cliente y en el negocio para poder escribir pruebas que satisfagan sus necesidades y agreguen el valor necesario en esta actividad. Tal vez no haya personas con dichos rótulos de tester y en ese caso el equipo entero es responsable de entregar producto con buena calidad.

Las personas involucradas en las pruebas del producto, incluyendo las actividades de relevar las necesidades del negocio para incluirlas en los escenarios de prueba, se involucran lo más temprano en el proyecto, marcando una gran diferencia de lo que se suele observar en proyectos que poseen un enfoque más tradicional al estilo cascada (waterfall) en que las pruebas son una fase post mortem al desarrollo. Además, cabe resaltar que los testers tampoco son los únicos responsables en agregar calidad al producto, tal como hemos visto anteriormente existen diferentes técnicas y prácticas que permiten a los programadores incorporar calidad en etapas tempranas del desarrollo. Por

otro lado, al tener un equipo en el cual se desarrollará o se evoluciona un producto de software mediante un enfoque de DevOps se debe tener especial atención en la calidad y en la automatización, dado que, al hacer iteraciones e incrementos (desarrollo ágil), cada porción nueva de software que se desarrolle debe ser probada antes de su lanzamiento a producción y además asegurar que ese nuevo cambio no introdujo defectos en la construido. Aquí el análisis sobre riesgos vinculados a la calidad y confiabilidad juegan un papel importante, motivo por el cual la automatización obtiene mayor relevancia.

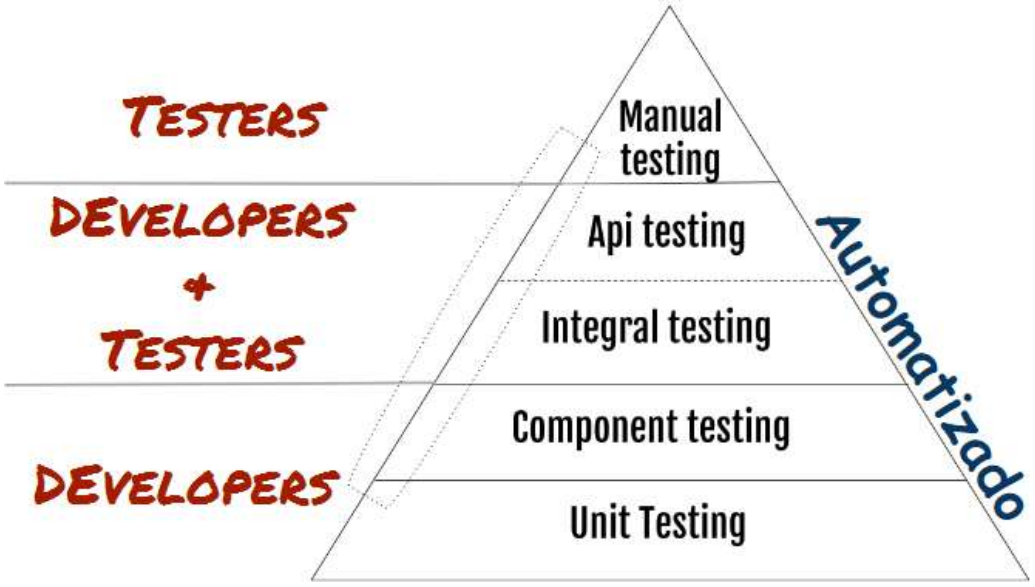
En ciclos de vida de desarrollo ágiles se busca enfatizar que las pruebas se realicen lo más temprano posible, gestionándolos con marcos empíricos, las pruebas se deben basar en el aprendizaje obtenido mediante la observación y evolución del producto. De nada serviría congelar las pruebas junto con los requerimientos en un enfoque ágil y de DevOps, estos van a cambiar y por ende lo hará el software.

En lugar de tener un enfoque predictivo en el cual se desean controlar los cambios que requiere el negocio, se propone que las pruebas (sea el proceso, la documentación, los datos y la manera de ejecutarlas) evolucionen a lo largo del tiempo de vida el producto. Es decir, que los testers escriban los escenarios al comienzo de la iteración en colaboración con los desarrolladores para ajustar lo necesario, y automaticen las pruebas necesarias para poder evitar la repetición manual de dichos casos en iteraciones venideras.

Lo que se enfatiza en un enfoque ágil del testing es que no sólo el desarrollo esté “infectado” con testing, sino que los testers tengan una participación distinta a la que suelen tener en un enfoque tradicional en el cual suceden al desarrollo. En un enfoque ágil, el tester debe estar “infectado” de creatividad, se debe dar un ámbito de colaboración y participación al inicio del desarrollo y no sólo al final para asegurar cierta calidad. Es mejor buscar incrementar el valor que se le da a la calidad cuando el software aún no fue construido o está siendo construido que cuando ya se finalizó. El tester debería trabajar en colaboración con el negocio para obtener un mayor entendimiento de sus necesidades aprendiendo de este para obtener mayor y mejor contexto de la funcionalidad que se deberá probar y validar si realmente esa funcionalidad al fin y al cabo es útil. Esto último es primordial. En una cultura DevOps se querrá entregar valor lo más rápido y con el menor riesgo posible, midiendo si eso realmente fue así.

La siguiente imagen representa un posible esquema de testing en una iteración de un proyecto con un enfoque ágil. El enfoque piramidal se da para poder tener una visión de las actividades y conceptos vinculados a las personas que se encuentran en un equipo de

elaboración ágil. Dicho enfoque remarca que el producto no sólo debe poseer pruebas unitarias (Unit Tests) sino que las pruebas deben ir creciendo a lo largo de las iteraciones mediante un corte transversal a la pirámide. De esta manera se puede observar claramente que las actividades que realizan los desarrolladores y los testers dentro de un equipo ágil en una iteración o bien en el despliegue de una nueva funcionalidad requiere mucho más esfuerzo del sólo hecho de realizar el código productivo del requerimiento o escribir el escenario. El hecho de que existan determinadas actividades que son realizadas por roles específicos es sólo orientativo y no significa que se deban realizar todas o bien que un rol realice sólo alguna de las que tiene demarcadas. Además, la pirámide representa de abajo hacia arriba en donde se debería poseer más cobertura de pruebas, y tal como se puede apreciar las pruebas manuales (también las pruebas de interfaz de usuario, GUI) son los que tendrían que estar en menor cantidad, ya que son las más caras de mantener.



El enfoque de testing ágil en un ámbito de DevOps, va más allá de probar el código que puede desarrollar el equipo desarrollo (para lo cual, a continuación, veremos algunas prácticas), se trata además de hacer foco en probar continuamente la fragilidad que tiene la infraestructura tanto una vez que está el software en producción como su proceso de traspaso en ambientes. El enfoque de testing ágil sobre la infraestructura y arquitectura será dado por la reproducción de las posibles fallas que se puedan dar en las mismas

comprobando la capacidad que tiene todo el sistema de responder de una manera acorde. Para ello, es necesario tener un enfoque de integración continua, poseer un sistema de versionado para el código, la infraestructura y los datos.

BDD - Behavior Driven Development

La técnica de Desarrollo Guiado por el Comportamiento (Behaviour Driven Development BDD, por sus siglas en inglés) surgió como necesidad de complementar la enseñanza y práctica de TDD, implementada por Dan North. BDD es la evolución de TDD. Primero escribir la prueba y luego el código para que pase la prueba. BDD se puede considerar como un proceso de desarrollo de software que acerca al negocio y a los desarrolladores a colaborar y validar las pruebas que se considerarán necesarias para que un requisito sea dado como terminado. Se dice que es la evolución de TDD dado que, en lugar de escribir las pruebas unitarias, se escribirán las pruebas que verifiquen el código desarrollado desde el punto de vista del negocio y en colaboración con este. El enfoque complementa no sólo a TDD, sino que también nos acercará a otro tipo de técnica la cual se la conoce con el nombre de ATDD (por sus siglas en inglés, Acceptance Test Driven Development). En este curso haremos una breve reseña y no ampliaremos más sobre estas dos técnicas ya que se consideran fuera del alcance del mismo, pero si consideramos que es útil conocerlas e identificar cuándo utilizarlas.

Como mencionamos anteriormente las pruebas de BDD son escritas antes de que se desarrolle el código de la necesidad del negocio. Dicha necesidad, generalmente, se la captura mediante lo que se conoce como *historias de usuario*. Las historias de usuario capturan las necesidades del negocio mediante un lenguaje ubicuo que acerca a los usuarios y a los desarrolladores. En el caso de los desarrolladores, la incorporación del lenguaje del dominio en la aplicación posibilitará una mejor documentación a través del código fuente (ya sea en las pruebas y en el código productivo), dando la ventaja de una mejor mantenibilidad a mediano/largo plazo. Luego de escribir las pruebas de aceptación y ejecutarlas, al estilo TDD, se prosigue a automatizarlas y para ello, comúnmente, están escritas en un lenguaje llamado *Gherkin*.

El lenguaje Gherkin es el más utilizado para la práctica de BDD ya que es entendible/legible por la mayoría de las personas y también por las computadoras para realizar la automatización, así mismo esto habilita a que la documentación del comportamiento se

aloje en la aplicación de una manera viva, y dando mantenimiento a la misma también se le estaría dando mantenimiento a las pruebas. Para comenzar con BDD sólo es necesario comprender cinco instancias de este lenguaje para poder hacer efectiva la prueba (Feature-Scenario-Given-When-Then):

Feature: es el nombre de la funcionalidad de negocio. El nombre debe ser unívoco y explícito. Muchas veces, se escribe en el formato de las historias de usuario: Como...Quiero...Para...

Scenario: identifica uno de los criterios de aceptación de funcionalidad. Para un determinado feature, se tendrá al menos un escenario de prueba.

Given: provee contexto. Aquí se detallan los prerrequisitos del sistema para poder probar el escenario en cuestión

When: especifica el conjunto de acciones / eventos que se realizarán en la ejecución de la prueba. Son las interacciones que se realizarán con el sistema.

Then: especifica el resultado esperado de la prueba. Son los cambios observados en el sistema.

En la actualidad hay una gran variedad de herramientas para BDD. Su uso dependerá del lenguaje de programación y del atractivo funcional que se encuentre en ellas. Algunas de las herramientas que se pueden utilizar para la automatización son: Cucumber, SpecFlow, Behave, JDave.

Las principales ventajas que posee BDD las podemos enumerar de la siguiente manera:

- Colaboración entre el negocio y el equipo de desarrollo.
- Mejora el entendimiento de las reglas de negocio en el equipo de desarrollo.
- Expansión de la práctica de TDD (genera un mejor entendimiento en los desarrolladores).
- Aumenta la mantenibilidad de la aplicación.
- Documentación viva en la aplicación.
- Se descubren problemas de usabilidad de manera temprana
- Reduce la cantidad de bugs
- Automatización de pruebas.

ATDD - Acceptance Test Driven Development

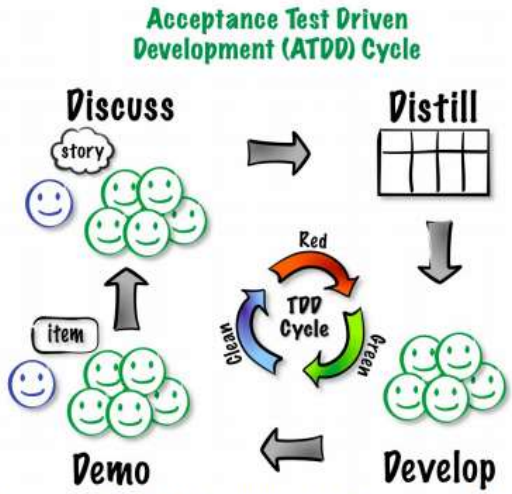
El desarrollo guiado por las pruebas de aceptación es una evolución de BDD en el cual ya no es en sí una técnica sino más bien una metodología en la cual los casos de prueba de aceptación son elaborados por el equipo completo de desarrollo inclusive el negocio/usuario. Los desarrolladores, los testers, los expertos en usabilidad, analistas de negocio y los usuarios conversan sobre qué restricciones debería poseer un requisito para que sea considerado por todos como algo terminado e implementado.

ATDD, es una práctica ágil avanzada que requiere cambios en el ámbito de la organización para que pueda ser posible, se necesita sustancialmente confianza y responsabilidad. En una primera mirada, el equipo debe ser lo suficientemente maduro, autoorganizado y multidisciplinario para posibilitar las conversaciones necesarias para determinar de una manera valiosa las pruebas de aceptación. Y, además, ese nivel de madurez en agilidad lo debe tener el equipo y la organización que lo soporta, ya que en el mismo lugar y momento en el que se discuten los requisitos de aceptación estará participando el negocio.

ATDD pone énfasis en las pruebas y en la colaboración en todo el ciclo, no sólo escribiendo las pruebas unitarias al inicio sino también en la aceptación, dando como resultado al final de la construcción del requisito una suite de pruebas de regresión más completa.

La práctica de ATDD posee 4 etapas: discusión, desagregación, desarrollo y demostración.

The Acceptance Test Driven Development (ATDD) Cycle



(ATDD cycle model developed by James Shore with changes suggested by Grigori Melnick, Brian Marick, and Elisabeth Hendrickson.)

La imagen original, está explicada a continuación

En una reunión de conversación (**Discuss**), primero se debate, desde el punto de vista del usuario, el comportamiento del sistema necesario para considerarse aceptado por todo el equipo, inclusive el negocio. Lo valioso de la conversación sobre los requerimientos en conjunto con el usuario es poder establecer las expectativas adecuadas sobre lo nuevo que se quiere construir. Tal vez se de en ese espacio de debate que el requerimiento en cuestión no sea prioritario en ese momento, que se sea demasiado grande para poder implementar de una sola vez.

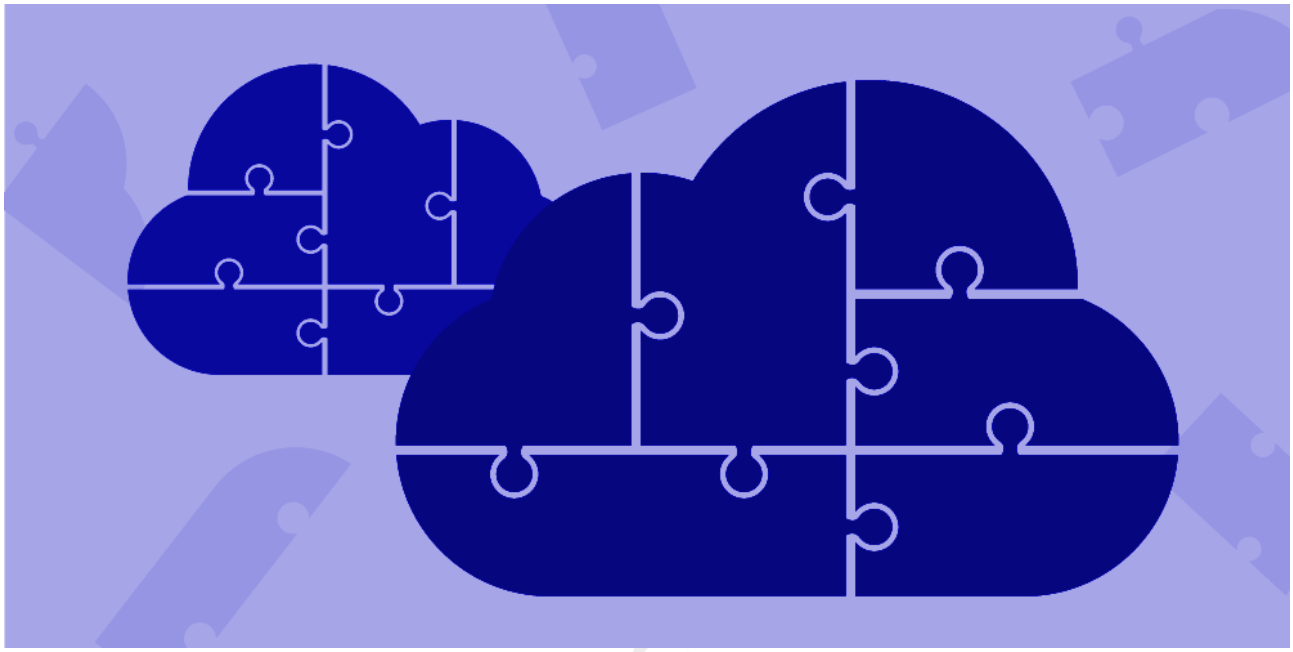
En la desagregación (**Distill**), ya habiendo entendido qué se deberá desarrollar se procede a escribir las pruebas de aceptación de manera de capturar el conjunto de pruebas ejecutables que se deberán ejecutar en nuestro framework de automatización. En esta etapa no hay que (pre)ocuparse de escribir la prueba en el formato aceptado para la automatización, sino que lo que se debe lograr es mostrar las intenciones que tendrán las pruebas, es decir los escenarios posibles y su conjunto de datos para que se puedan dar.

Desarrollar (**Develop**), el código. En esta etapa, el equipo desarrollará las pruebas antes elaboradas en un lenguaje común al framework de automatización y se utilizará el enfoque de TDD/BDD en el cual se suceden estos eventos:

- Primero se escribe la prueba
- La prueba falla
- Se escribe el código de implementación hasta que la prueba pase
    - Para pruebas unitarias
    - Y para pruebas de aceptación
- Se refactorea el código buscando perfeccionarla y eliminar casos duplicados

Finalmente, en la demostración (**Demo**) el equipo junto con el stakeholder observan que las pruebas cumplan las expectativas de comportamiento del sistema que se establecieron en un principio y además se suelen ejecutar pruebas exploratorias sobre el código implementado para detectar posibles riesgos que no se hayan identificado con antelación. Estas últimas pruebas son manuales.

4.Arquitectura de contenedores / Microservicios



La tecnología de contenedores no es nueva ni se formó con el software Docker. Su aplicación comenzó en el mundo Linux el cual desde, aproximadamente, el 2009 contuvo una versión del Kernel llamada LXC que proveía una interfaz de virtualización distinta a lo que se conoce como máquina virtual. En realidad, lo que se virtualiza, el contenedor, son una serie de procesos y recursos en el entorno original. Cuando se virtualiza se posee dos

ámbitos: la máquina host que es la cual posee los recursos en los cuales se configura la abstracción que se desea y por otro lado los contenedores, que serán entornos virtuales aislados de la máquina host y del resto de los contenedores, posibilitando compartir los recursos y su visibilidad mediante una interfaz establecida. Si bien los contenedores no poseen el mismo nivel de aislamiento que una máquina virtual, su abstracción y su ligereza los convierten en una solución muy versátil al utilizarla con una arquitectura orientada a servicios.

Los contenedores se basan en imágenes de sistemas operativos, a su vez la ejecución de estos contenedores virtualiza ciertos procesos de la máquina host y son manejados por su kernel. Es decir que los contenedores no son sistemas operativos completos, sino que conviven a través de una capa de virtualización orquestada por el kernel de la máquina host, mientras comparten los recursos que sean necesarios.

Lo anterior, el porqué es más barato a nivel recursos tener virtualización por contenedores que por sistemas completos. Por ejemplo, un desarrollador podría tener varios contenedores ejecutándose en simultáneo en su entorno sin ningún tipo de dificultad y sin necesidad de incurrir a tener costosos entornos. Además, al virtualizar en una determinada capa posibilita que los distintos contenedores, a su vez, virtualicen otras capas subyacentes de manera que cada uno de ellos pueda aprovechar la virtualización sobre la que está el otro. Cabe resaltar que muchas veces los contenedores se utilizan para definir la infraestructura de la aplicación, pero no así para almacenar sus datos. Es decir que lo que se almacene en un contenedor no será persistente dentro de él, al apagarlo toda la información que no fue definida en su configuración de la imagen no será persistida (stateless). Esta última característica es lo que transforma a los contenedores en **infraestructura inmutables** y posibilita que su infraestructura sea más confiable, ya que si se desea realizar un cambio se debe hacer a través del archivo de configuración (que debiera estar en un sistema de control de versiones de código fuente) y luego al compilar la imagen con el nuevo cambio queda impactado en ella.

Cuando se instala el software Docker, se elige / completan las configuraciones necesarias en el archivo correspondiente, por ejemplo, qué sistema operativo se quiere, puertos expuestos, etc. Luego se ejecuta la compilación de la imagen que se descargó en conjunto con la configuración que se aplicará a dicha imagen. También se podría hacer un commit sobre la imagen, compilarla y subirla a un repositorio de imágenes.

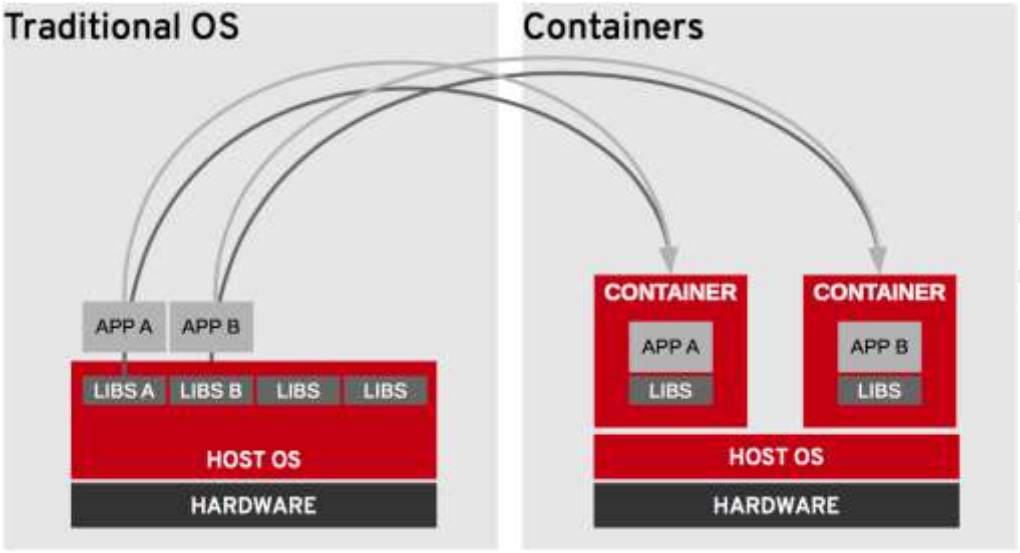
Lo visto hasta aquí en esta unidad es un conjunto de prácticas y técnicas más utilizadas en la industria del software y en las organizaciones basadas en DevOps, lo cual no quiere decir que sean las únicas.

En el caso de querer optar por una infraestructura en versionado de código (Infrastructure as Code, IaC por sus siglas en inglés) con mayor flexibilidad y posibilidad de automatizarla, es en donde aparece el interrogante de qué arquitectura utilizar. Generalmente, al querer implantar una aplicación y aprovechar los beneficios que nos proveerá la virtualización en contenedores, la arquitectura más utilizada es la de orientada a microservicio o a servicios.

Las aplicaciones basadas en servicios utilizan interfaces (API) bien definidas para cada servicio de manera que la comunicación de los clientes que la consumen sea específica. Dentro de una arquitectura de contenedores, el objetivo al que se apunta es que cada contenedor posea un servicio (*un contenedor, una responsabilidad*), que esté a su vez tendrá una especificación de la API por la cual se comunican los clientes (el código desarrollado).

Además, lo que posibilita el contenedor es que su servicio escale y crezca de una manera independiente, ya que cualquier modificación que se realice no debiera afectar al resto de la aplicación, no sólo a nivel código sino a nivel despliegue en producción.

Esto significa que mientras la especificación de la API se mantenga, la aplicación puede crecer y escalar de manera horizontal sin afectar a la totalidad de la aplicación, ya que se podrían hacer despliegues individuales a un servicio específico sin la necesidad de detener todo el sistema. Además de estas ventajas antes descritas, el uso de contenedores más la arquitectura basada en servicios nos da la posibilidad de probar las fallas del sistema a un bajo costo, ya que podremos simular distintas fallas de ambientes y probablemente luego se automaticen las pruebas de esas fallas incorporándolas a la suite de pruebas dentro del ciclo de vida de desarrollo.



Cada contenedor debería representar un único servicio y a su vez, cada servicio debería representar una única unidad de negocio de manera que posea un interfaz con foco en una determinada a una problemática de negocio.

Por otro lado, la arquitectura de microservicios permite tener la flexibilidad de implementar soluciones de servicios en distintas tecnologías, dado que cada contenedor podría tener diferentes sistemas operativos (alojados en hosts distintos) ya que posee la característica de ser desacoplados. Por ejemplo, se podría tener un servicio desarrollado en Ruby, C++, en NodeJS, en .NET y cada uno de ellos tendrá un desarrollo y una implementación independiente, sin afectar el resto. Esto permite tener flexibilidad para utilizar la tecnología adecuada a cada necesidad que se requiera sin estar a sujeto a alguna en particular por ser el núcleo de toda la aplicación. Por otro lado, crecer a gran escala con este tipo de enfoque puede tener consecuencias adversas en su mantenimiento, ya que conlleva una gran demanda en la administración de cada máquina host y contenedor. Igualmente, para ello hay otras herramientas que permiten y facilitan la automatización de esto sin incurrir en tareas repetitivas para los administradores y que no sean propensas a error.

5.Orquestador

Implementar una aplicación implica manejar diferentes ambientes y recursos, ya sea servicios, datos, configuraciones y/o directorios. Esto causa complicaciones debido a que para implementar una aplicación en un ambiente hay que seleccionar qué ambiente se

requiere, qué configuración aplicará a la aplicación y al ambiente en cuestión y los datos que se consuman. Los tiempos de complicación y pruebas aumentan aún más si el ambiente seleccionado para implementar es compartido con otras aplicaciones, ya que hay que controlar y evitar que los cambios introducidos en el ambiente no afecten al resto de las aplicaciones si es que está compartido.

El orquestador de implementación es el responsable de **coordinar la secuencia de pasos** requeridas para poder hacer el despliegue de una aplicación en un ambiente, **de una manera automatizada**. El ambiente podría ser pruebas para desarrolladores, pruebas de integración, pruebas de aceptación o bien producción. Lo recomendable es que el proceso y los scripts que se utilicen sean los mismos para todos los ambientes. De esta manera se mantiene probado reiteradas veces tanto el proceso de implementación como los scripts necesarios para realizarlo. La secuencia de pasos a seguir del orquestador para la implementación varía según la organización, la necesidad del negocio y el equipo sobre la aplicación desarrollada.

Por ejemplo, un orquestador para una aplicación podría realizar pasos semejantes a los siguientes:

- Detectar qué rama de código realizó el cambio
- Ejecutar la batería de pruebas automatizadas
- Ejecutar análisis de código estático
- Generar los archivos binarios si corresponde a una aplicación compilada (Build)
- Preparar el ambiente a implementar (servidores, aprovisionamiento de software, etc)
- Transferir la aplicación al ambiente
- Configurar las variables requeridas en el ambiente
- Proveer datos al ambiente
- Si existen, realizar pruebas de integración hacia otros sistemas
- Realizar un *Smoke Test*[4] sobre el ambiente desplegado
- Registrar mediciones e informes de errores y advertencias en cada caso

Como hemos visto, estos pasos que realizará un orquestador dependen de la organización en que se desarrolle la aplicación y del ambiente en que se quiere desplegar la aplicación.

[4] Smoke Test. Ver archivo anexo de links

Hay que recalcar que el conjunto de la secuencia de pasos que realiza el orquestador es el proceso o el pipeline de desarrollo y despliegue de la aplicación. Esto quiere decir que el orquestador representa el proceso de una manera automatizada, mostrándolo de una manera explícita a la organización, y cada actividad de ese proceso son las tareas (con la expectativa que la mayoría sean automatizadas) que lo conforman. La visibilidad de tener esto nos permite optimizar el proceso de despliegue.

Mientras las actividades que realiza el orquestador estén más automatizadas y haya cada vez menos pasos manuales para el flujo desde el desarrollo hasta la entrega del software en los ambientes que correspondan (Pipeline), más rentable será para la empresa. Al automatizar los distintos pasos, midiendo y detectando posibles errores, las implementaciones son menos dolorosas, por ende, podrían ser más frecuentes y menos riesgosas. En lugar de días o semanas pueden llegar a durar de un par de minutos a algunas horas. Generalmente lo que se suele ver en las organizaciones es que el proceso de despliegue en ambientes suele estar automatizado hasta los ambientes de prueba (QA) y no punta a punta, desde el desarrollo hasta la entrega en producción. Este último paso para desplegar la aplicación en un ambiente productivo en términos generales involucra la interacción con otras áreas y pasos manuales para que posibiliten la implementación en producción.

Bibliografía utilizada y sugerida

- Alex Williams. The Docker & Container Ecosystem. The New Stack.
- Andrew Hunt & Venkat Subramaniam. Practices of an Agile Developer: Working in the Real World. 1era edición. EE:UU: Pragmatic Bookshelf; 2006
- Jez Humble, David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. 1era edición. EE:UU: Addison Wesley; 2010
- Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. 1era edición. EE:UU. Pearson Education; 2009
- Tobias Mayer y Alan Cyment. Por Un Scrum Popular: Notas para una Revolución Ágil (Spanish Edition). Dymaxicon; 2014.

Lo que vimos:

En esta unidad dimos un recorrido por los conceptos de las principales prácticas y técnicas de desarrollo que se utilizan en ambientes de DevOps, desde la perspectiva de la codificación, las pruebas, la arquitectura y la infraestructura.

El contenido de esta unidad es fundamental para comprender que las herramientas actuales están basadas en conceptos de más alto nivel y que para su utilización es necesario tener presente el impacto que puede provocar en los equipos y en la organización.

Observamos las principales ventajas y desventajas de algunas de estas prácticas, brindando el conocimiento necesario para identificar la conveniencia de su utilización en el equipo u organización al que se desee aplicar.

Lo que viene:

En la siguiente unidad nos adentraremos concretamente en la práctica que prescribe la Integración Continua. Abordaremos los principales conceptos y requisitos que se recomiendan tener para implementarla.