



Análisis y Diseño de Sistemas 2

2015

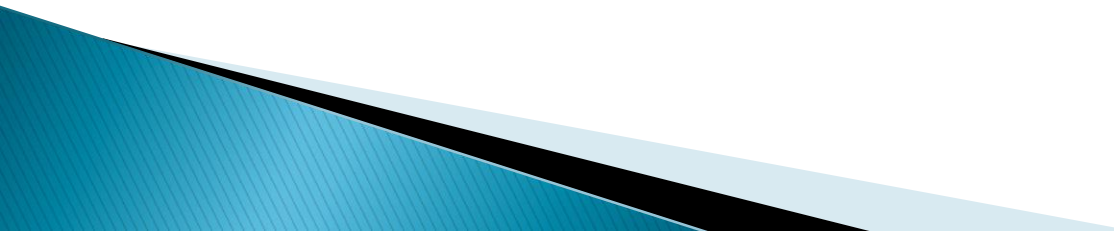
Ing. Luis Alberto Arias Solórzano

Unidad 1

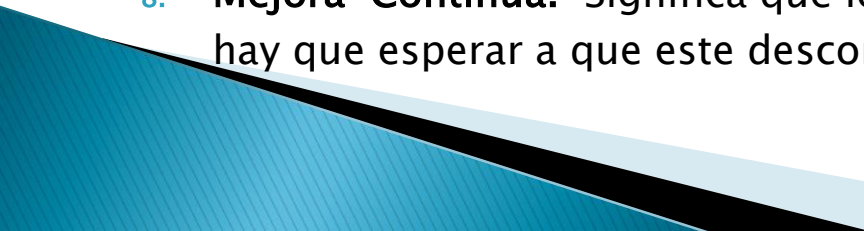
Continuous delivery

- ▶ Continuous Delivery es la entrega continua de código a un ambiente, una vez el desarrollador sienta que dicho código esta listo para ser enviado.
- ▶ Esto puede ser a UAT, Staging o directamente a Producción. La idea es entregar código a un usuario base, ya sea del equipo de Quality Assurance (QA) o al usuario final que revisa continuamente.
- ▶ Es muy similar a Continuous Integration, pero este capta también las pruebas de la lógica del negocio. Las pruebas de unidad son limitadas y no captan toda la lógica del negocio esta fase puede ayudar a captar esos detalles.
- ▶ También se utiliza para revisión de código, este se marca para *release* una vez pase por QA y UAT.
- ▶ La base de la entrega continua es crear grupos pequeños o batches pequeños de trabajo o código que se envían constantemente a revisión y facilitan encontrar errores con antelación.

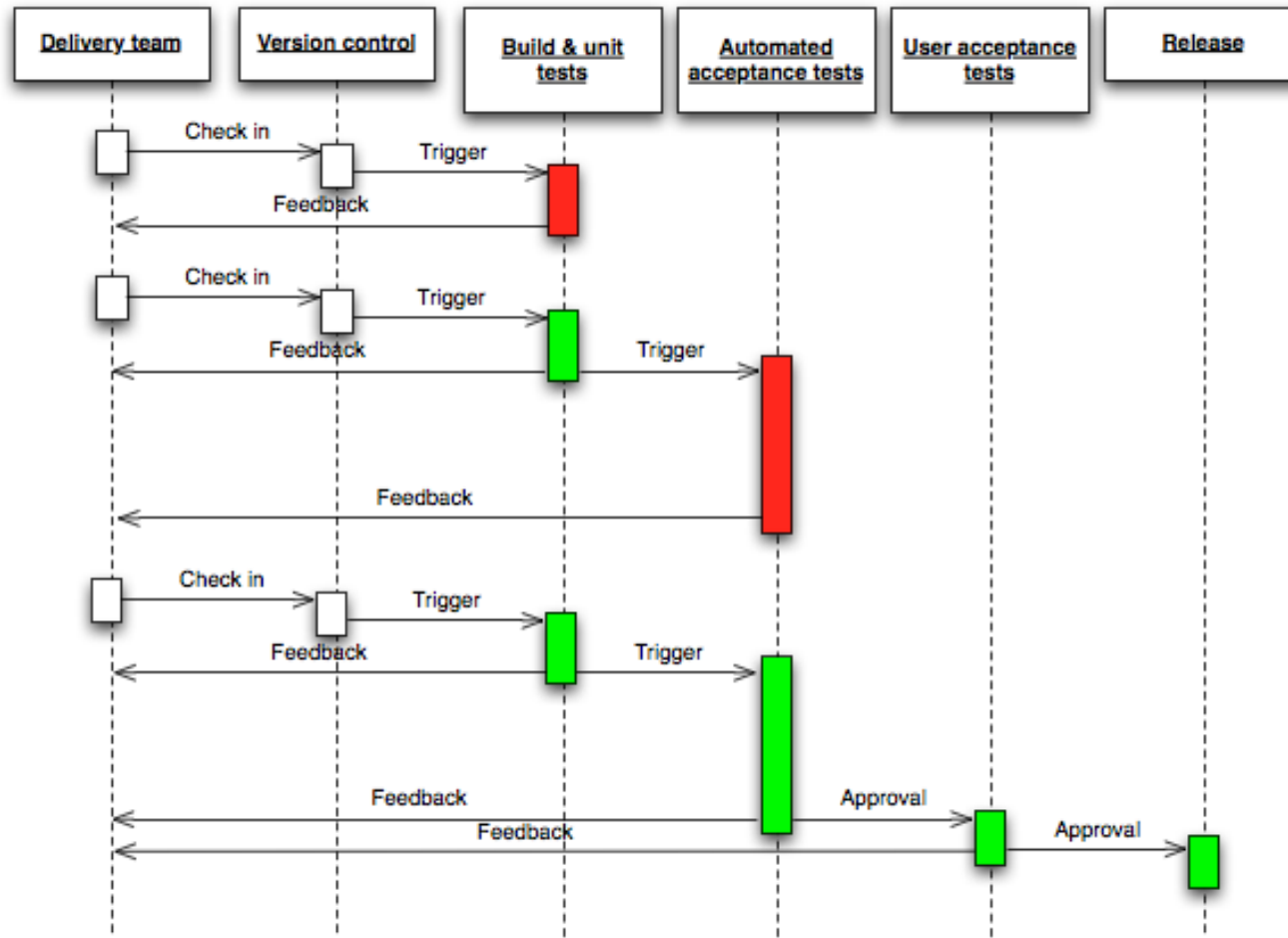
Continuous delivery – Principios

1. El proceso de release o deploy del software debe ser repetible y confiable.
 2. **Automatizar todo:** Un proceso manual no puede considerarse siempre como confiable y menos repetible. Se deben automatizar las tareas repetitivas esto nos conducirá a la confiabilidad.
 3. **Si algo es difícil y tedioso, hazlo mas a menudo.** Puede sonar tonto, pero esto básicamente significa que hacer las actividades dolorosas o tediosas nos llevara a mejorarlas, probablemente a automatizarlas y hacerlas mas sencillas.
 4. **Mantener todo bajo control de versiones.** Puede sonar obvio pero pueden existir sistemas que aun no están en un lugar seguro y ordenado.
- 


Continuous delivery – Principios

5. **Terminado significa liberado “released”.** Esto implica que la propiedad del proyecto se encuentra en manos del usuario y trabaja correctamente. No deben existir preocupaciones o situaciones en las que no se haya revisado algo. Es común tener personas que entregan código directamente a producción y monitorean su funcionamiento, no es considerado una buena practica.
 6. **Desarrollo de Calidad:** Toma un tiempo para implementar métricas que midan la calidad de tu desarrollo, e implementa estándares que te ayuden a mantener la calidad en el software.
 7. **Todos son responsables en el proceso de “release”.** Un sistema que solo corre en la maquina del desarrollador no genera dinero a la empresa. Desarrolladores, managers, testers, todos en el equipo deben trabajar para que el producto llegue a manos del cliente.
 8. **Mejora Continua.** Significa que los sistemas siempre están evolucionando, no hay que esperar a que este descontinuado y sea mas difícil de mantener.
- 

Continuous delivery – Principios

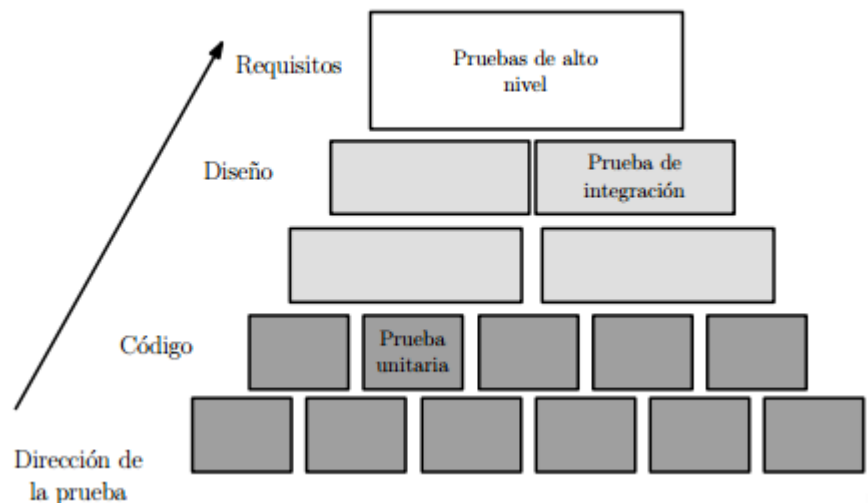


Continuous delivery – Practicas

1. **Construir “Build” las librerías y binarios solo una vez.** Constantemente nos vemos en la necesidad de recompilar el proyecto completo, incluyendo las dependencias, una buena practica es mantener los binarios separados y evitar el recompilar las piezas que se usan comúnmente.
 2. **Utilizar el mismo mecanismo para implementar en todos los ambientes.** Suena obvio, pero hay casos donde para liberar un software en el ambiente de QA es automático y en producción es manual (copiando archivos) y también casos donde si es automático en los ambientes pero utilizando un método distinto.
 3. **Realizar pruebas de humo.** Una vez entregado no dejes el producto por su cuenta, escribe un plan de pruebas de humo y asegúrate que funcionen ya colocado en el ambiente deseado, se debe hacer un buen diagnostico.
 4. **Si cualquier cosa falla, hay que detenerse.** Debemos iniciar el proceso nuevamente, no parchemos, ni truquemos el proceso. Si es necesario hay que hacer rollback y solucionar el problema antes de iniciar de nuevo.
- 

Estrategias de pruebas

Una estrategia de prueba de software proporciona una guía que describe los pasos que deben realizarse como parte de la prueba, cuando se planean y se llevan a cabo dichos pasos, y cuanto esfuerzo, tiempo y recursos se requieran. Por tanto, cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de casos de prueba, la ejecución de la prueba y la recolección y evaluación de los resultados. Una estrategia de prueba de software debe ser suficientemente flexible para promover un uso personalizado de la prueba. Al mismo tiempo, debe ser suficientemente rígida para alentar la planificación razonable y el seguimiento de la gestión conforme avanza el proyecto



Criterios para completar las pruebas

Cada vez que se analiza la prueba del software, surgen una preguntas clásicas:

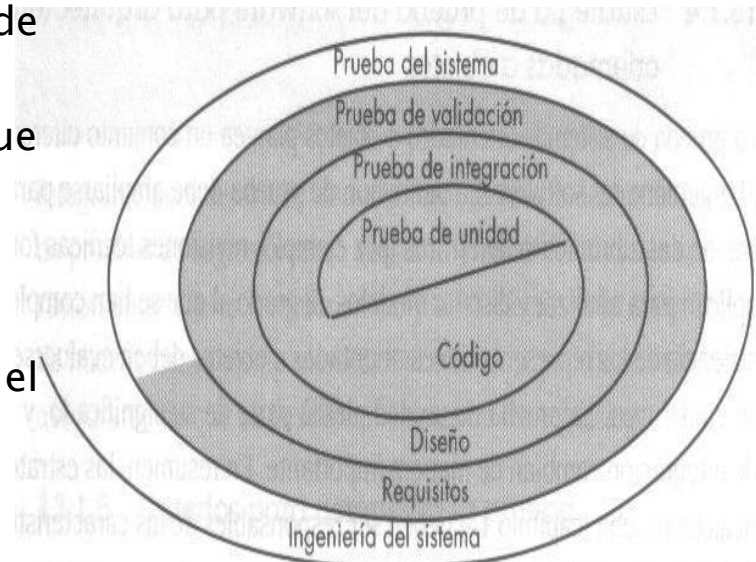
¿Cuándo terminan las pruebas ?

¿Cómo se sabe que se ha probado lo suficiente?.

Lamentablemente , no hay una respuesta Definitiva a estas preguntas , pero existen algunas respuestas pragmáticas e intentos tempranos a manera de guía empírica.

Una respuesta a la pregunta es: “nunca se termina de probar; la carga simplemente pasa de usted (el ingeniero de software) al Usuario final”. cada vez que el usuario ejecuta un programa de computo, el programa se pone a prueba.

Otra respuesta un tanto cínica, mas no obstante precisa es: “ las pruebas terminan cuando se agota el tiempo o el dinero”.



Estrategias de pruebas

1. **Especificar los requerimientos del producto en forma cuantificable mucho antes de comenzar con las pruebas.** Aunque el objeto predominante de una prueba es encontrar errores, una buena estrategia de prueba también valor otras características de la calidad , como la portabilidad, el mantenimiento y la facilidad de uso. Esto debe especificarse en una forma medible, de modo que los resultados de las pruebas no sean ambiguos.
2. **Establecer de manera explicita los objetivos de las pruebas.** Los objetivos especificados de las pruebas pueden enunciarse en términos medible. Por ejemplo, la efectividad de las pruebas, su cobertura, el tiempo medio antes de aparecer una falla, el costo por descubrir y corregir defectos. La densidad de defectos restantes o la frecuencia de ocurrencia, y las horas de trabajo de prueba deben enunciarse dentro del plan de la prueba.

Estrategias de pruebas

3. **Conocimiento general del software.** Entienden a los usuarios del software y desarrollan un perfil para cada categoría del usuario. Los casos de uso que describen el escenario de interacción para cada clase de usuario pueden reducir el esfuerzo de prueba global al enfocar las pruebas en el uso real del producto.
4. **Desarrollar un plan de prueba que enfatice “prueba de ciclo rápido”.** Es recomendable que un equipo de software “aprenda a probar en ciclos rápidos (2 por ciento del esfuerzo proyecto) de cliente– Utilidad al menos la ‘comprobabilidad’ en campo, los incrementos de funcionalidad y/o la mejora de la calidad”. La retroalimentación Generada a partir de estas pruebas de ciclo rápido puede usarse para controlar niveles de calidad y las correspondientes estrategias de prueba.

Estrategias de pruebas

5. **Construir software “robusto” que esté diseñado para probarse a si mismo.** El software debe diseñarse en forma que use técnicas anti errores es decir, el software debe poder diagnosticar ciertas clases de errores. Además el diseño debe incluir pruebas automatizadas y pruebas de regresión.
6. **Usar revisiones técnicas efectivas como filtro previo a las pruebas.** Las revisiones técnicas pueden ser tan efectivas como probar para descubrir errores. Por esta razón, las revisiones pueden reducir la cantidad del esfuerzo de pruebas que se requieren para producir software de alta calidad. Esto es como filtro previo a las pruebas. Las revisiones técnicas pueden ser tan efectivas como probar para descubrir errores. Por esta razón, las revisiones pueden reducir la cantidad del esfuerzo de pruebas que se requieren para producir software de alta calidad.

Estrategias de pruebas

7. **Desarrollar un enfoque de mejora continuo para el proceso de prueba.** La estrategia de pruebas debe medirse. Las métricas recopiladas durante las pruebas deben usarse como parte de un enfoque de control de proceso estadístico para la prueba del software.




Estrategias de pruebas

Desarrollar un conjunto de pruebas completas de nuestro sistema es una tarea cuya complejidad dependerá del entorno de ejecución de nuestra aplicación y de las interacciones con sistemas externos. Un conjunto de pruebas completas de nuestro sistema implicaría tener diferentes conjuntos de pruebas:

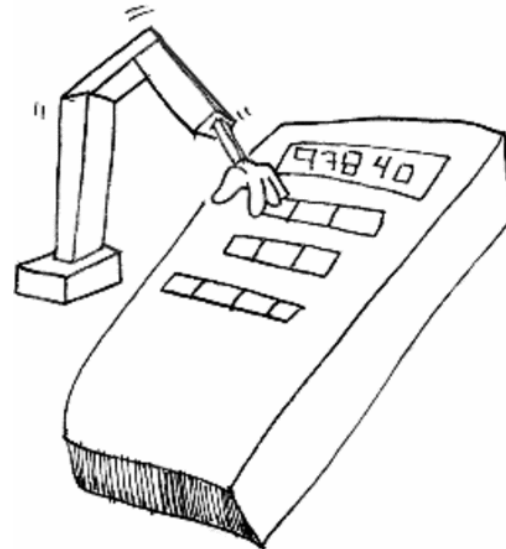
- Pruebas Unitarias
- Pruebas de Rendimiento
- Pruebas de Integración , interna y con sistemas externos
- Pruebas de Diseño (Accesibilidad)
- Pruebas Funcionales
- Pruebas de regresión
- Pruebas de Aceptación (UAT)

En el modelo tradicional de desarrollo este tipo de pruebas se realizaban una vez finalizado el desarrollo. No obstante este enfoque no suele funcionar en la mayoría de los proyectos de desarrollo y es realmente poco efectivo.



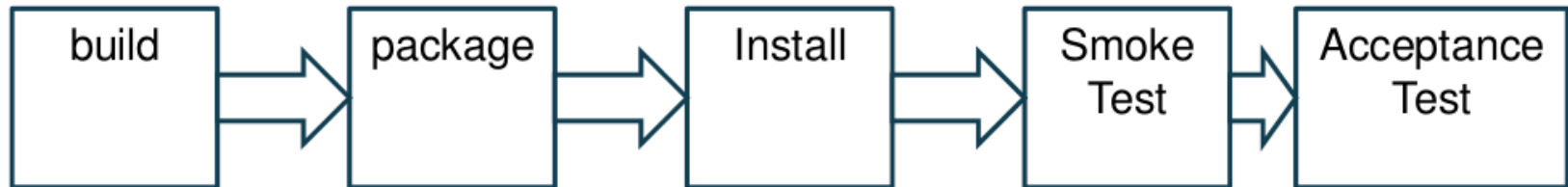
Pruebas automaticas

Desarrollar pruebas automáticas nos permite comprobar en cualquier momento y ejecutando la 'suite' de pruebas si nuestro sistema cumple con la funcionalidad implementada hasta el momento. Esta facilidad es tremendamente útil durante el desarrollo, puesto que nos permitirá desarrollar cambios con seguridad, sin tener que pagar el esfuerzo de volver a probar todo. Dejar las pruebas para el final del desarrollo nos privará de esta ventaja.



Deployment Pipeline

El pipeline es un flujo de tareas que se ejecutan secuencialmente, iniciándose cuando se hace una entrega al repositorio SCM, y si todo se ejecuta acaba con unas pruebas de aceptación pasadas correctamente.

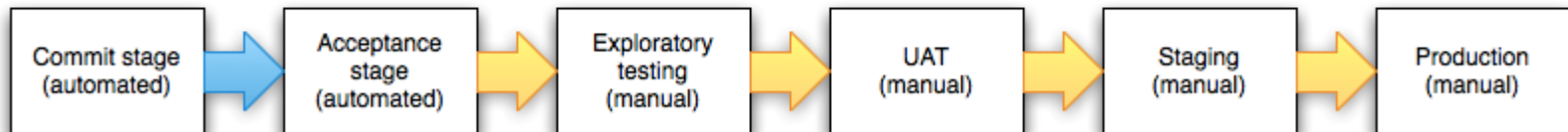
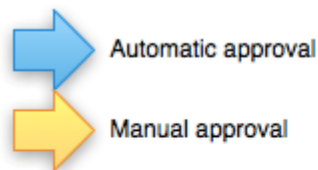


Uno de los retos de un entorno de compilación y pruebas automatizadas se requiere ser rápido, de modo que se puedan obtener respuestas rápidas, pero las pruebas integrales tardan mucho tiempo en ejecutarse.

Un **deployment pipeline** es una manera de lidiar con esto mediante la ruptura de su construcción en etapas. Cada etapa proporciona el aumento de la confianza. Las primeras etapas pueden encontrar la mayoría de los problemas de rendimiento de retroalimentación más rápida, mientras que las etapas posteriores proporcionan más lento y más a través de sondeo, esto es una parte central de Continuous Delivery.

Deployment Pipeline – Etapas

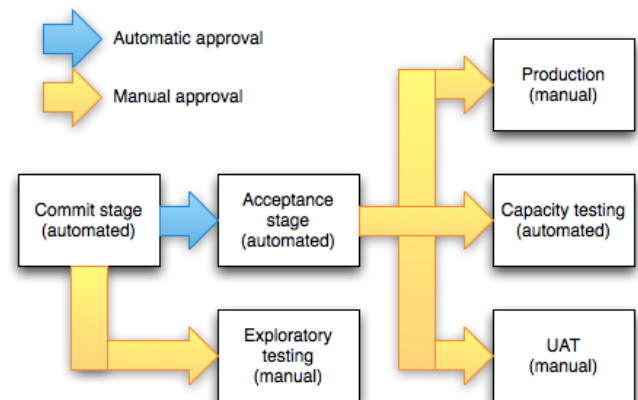
- ▶ Por lo general, las primeras etapas se hará cualquier recopilación y se proporcionaran los binarios para las etapas posteriores.
- ▶ Las etapas posteriores pueden incluir comprobaciones manuales , tales como las pruebas de que no se pueden automatizar.
- ▶ Las Etapas pueden ser automáticas o requerir una autorización para proceder humano , pueden ser paralelas sobre muchas máquinas para acelerar la construcción.
- ▶ La implementación en producción es por lo general la etapa final.



Deployment Pipeline – Proposito

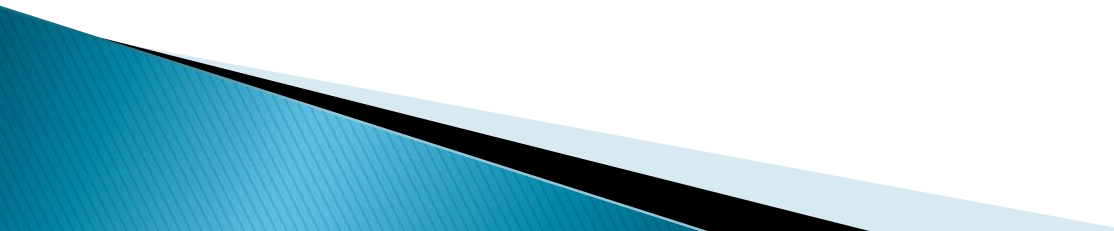
En términos más generales el trabajo de la deployment pipeline es detectar los cambios que conducirán a problemas en la producción . Estos pueden incluir el rendimiento , la seguridad , o problemas de usabilidad . Una tubería de despliegue debe permitir la colaboración entre los distintos grupos que participan en la distribución de software y ofrecer a todos la visibilidad sobre el flujo de los cambios en el sistema, junto con una pista de la auditoría completa.

Una buena forma de presentar la entrega continua (Continuous Delivery) es modelar su proceso de entrega actual como una deployment pipeline, a continuación, examinar esta para encontrar los cuellos de botella , las oportunidades para la automatización y puntos de colaboración.



Tarea – Investigar

Conceptos:

- Pruebas Unitarias
 - Pruebas de Rendimiento
 - Pruebas de Integración
 - Pruebas de Diseño (Accesibilidad)
 - Pruebas Funcionales
 - Pruebas de regresión
 - Pruebas de Aceptación (UAT)
- 

Gracias