Estrategia de pruebas

Exported on Aug 1, 2018 4:22 PM

Table of Contents

1 Introducción

Muchos proyectos dependen enteramente de pruebas funcionales manuales para verificar los requerimientos funcionales y no funcionales de las aplicaciones. Otra veces, incluso cuando las pruebas automatizadas existen, éstas están desactualizadas y sin mantenimiento.

El testing es una actividad crosfuncional que involucra a todo el equipo y debe realizarse desde el principio. Construir con calidad desde el principio significa que debemos implementar pruebas automatizadas en multiples niveles (unitarias, de componentes y de aceptación). Además, estas pruebas deben ejecutarse cada vez que se introduzcan nuevos cambios en la aplicación (código fuente, configuración o ambiente).

Las pruebas manuales también son escenciales pero deberían enfocarse a otro tipo: Pruebas de usabilidad, pruebas exploratorias, demostraciones del software.

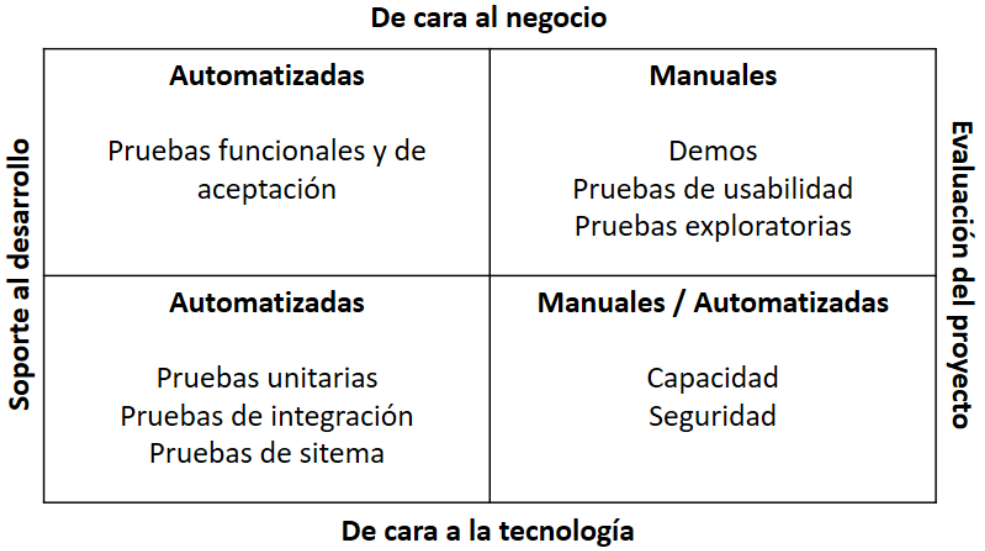
En un proyecto ideal, testers colaboran con desarrolladores y usuarios para construir pruebas automatizadas desde el principio del proyecto. Es decir, las pruebas se escriben antes de que el equipo comience a trabajar en la implementación de los requerimientos. El resultado es una especificación ejecutable del comportamiento del sistema y cuando las pruebas pasen la funcionalidad estará implementadad correcta y completamente.

Si las pruebas se aplican no solo a los aspectos funcionales del sistema, pueden reforzar el cumplimiento de otros requerimientos no funcionales como rendimiento y seguridad. Por ejemplo, si implementar una nueva funcionalidad incrementa el tiempo de ejecución de las pruebas de manera considerable, puede corregirse o refactorizarse de manera temprana.

Este mundo idea puede alcanzarse por completo en proyectos que adopten de manera temprana y con disciplina los princpios y estratégias apropiados.

2 Tipos de pruebas

Existen varios tipos de pruebas. Brian Marick propuso uno de los modelos más usados sobre los tipos de pruebas que los proyetos deben tener para asegurar la entrega de aplicaciones de alta calidad:



En el modelo, las pruebas se categorizan de acuerdo a si son de cara al negocio o a la tecnología y si soportan el proceso de desarrollo o sirven para evaluar el proyecto.

2.1 Pruebas de cara al negocio que soportan el proceso de desarrollo

Estas pruebas son comunmente conocidas como pruebas funcionales o de aceptación y aseguran que el criterio de aceptación de una historía se cumpla. Estas pruebas son críticas porque ayudan a los desarrolladores a responder la preguta ¿Cómo sé que he terminado? y ¿He obtenido lo que quería? a los usuarios.

Herramientas como Cucumber, JBehave, Concordion y Twist, ayudan a separar los scripts de pruebas de la implementación de la aplicación a la vez que proporcionan una forma de mantenerlos sincronizados.

En general, hay un camino aceptado que el usuario seguirá en su interacción con la aplicación conocido como "happy path" y suelen escribirse en la forma de ***Given/Dado*** [que el sistema se encuentra en un determinado estado] ***When/Cuando*** [el usuario realiza una serie de acciones] ***Then/Entonces*** [un conjunto de nuevas características o estado se obtendrá como resultado]. A esto suele hacerse referencia como el modelo **GIVEN-WHEN-THEM** de las pruebas.

Las pruebas de aceptación (manuales o automatizadas) deben ser ejecutadas cuando el sistema se encuentra en un ambiente y estado idéntico al de producción.

2.2 Pruebas de cara a la tecnología que soportan el proceso de desarrollo

Estas pruebas son creadas y mantenidas exclusivamente por desarrolladores y se dividen en tres grupos: pruebas unitarias, pruebas de componentes y pruebas de despliegue.

- **Pruebas unitarias:** Prueban una parte aislada del código. Por esta razon suelen simular otras partes del sistema usando mocks, stubs o doubles. Las pruebas unitarias no deben incluir llamados a base de datos, filesystem, ni otros componentes externos. De este modo

las pruebas unitarias debrían ejecutarse de modo rápido y deberían cubrir la gran mayoría del código. La desventaja de estas pruebas rápidas es con ellas no siempre se pueden detectar problemas que derivan de la interacción de varios componentes.

- **Pruebas de integración o componentes:** Son pruebas mas grandes que se enfocan en grupos de funcionalidades. Suelen ser más lentas ya que suelen realizar invocaciones a sistemas externos (bases de datos, servicios, etc).
- **Pruebas de humo o despliegue**: Son pruebas que se ejecutan siempre que se realizan despliegues de la aplicación y permiten verificar que los sistemas instalados funcionen y estén configurados correctamente. Esto incluye verificar que los sistemas externos sean accesibles y respondan apropiadamente.

2.3 Pruebas de cara al negocio que permiten evaluar el proyecto

Estas pruebas suelen ser manuales y permiten validar que la aplicación va a entregar a los usuarios el valor que se espera. No solo se trata de verificar que la aplicación cumpla con las especificaciones sino también de verificar que las especificaciones sean correctas.

- **Demos:** Los equipos ágiles realizan demostraciones a los usuarios al final de cada iteración para demostrar la nueva funcionalidad que se ha añadido. Sin embargo, la funcionalidad también debe mostarse a los usuarios lo más pronto posible para asegura que cualquier malentendido o problema con las especificaciones sea encontrado lo más pronto posible.
- **Pruebas exploratorias:** El testing exploratorio podría describirse como un proceso de aprendizaje creativo que no solamente permite encontrar bugs, sino también identificar posibles conjuntos de pruebas que pueden automatizarse e incluso nuevos requerimientos para la aplicación.
- **Pruebas de usabilidad:** Se realizan para descubrir qué tan fácil es que los usuarios cumplan sus objetivos con ayuda del software. Hay diferentes formas de realizar pruebas de usabilidad, desde cuestionarios y preguntas hasta métodos que implican grabar a los usuarios realizando tareas con la aplicación.

Finalmente, hay un tipo de pruebas que son realizadas directamente por los usuarios finales: las pruebas beta. De hecho las organizaciones con más visión del futuro entregan nuevas versiones de sus productos constantemente a grupos de usuarios de prueba sin que estos incluso, algunas veces, se den cuenta. Una buena estrategia, por ejemplo, es usar canary release y recopilar métricas que determinen cómo las nuevas características son usadas y si es que éstas funcionan correctamente y entregan suficiente valor.

2.4 Pruebas de cara a la tecnología que evalúan el proyecto

Las pruebas de aceptación vienen en dos categorías: funcionales y no funcionales. Como no funcionales entendemos a los atributos de calidad no incluidos en la funcionalidad (capacidad, usabilidad, disponibilidad, seguridad, etc.). Aunque los usuarios casi nunca emplean tiempo para especificar características de seguridad y capacidad,es más que seguro que una perdida en sus datos o la no disponibilidad de la aplicación por problemas de capacidad les sería muy molesto. Por este motivo, los requerimientos no funcionales deben ser especificados de la misma manera que los requerimientos funcionales al principio de los proyectos.

Las pruebas que se usan para evaluar estos criterios de aceptación no funcionales, así como las herramientas con las que se llevan a cabo, suelen ser diferentes y requieren ambientes especiales, conocimiento especializado y otros recursos considerables. Por este motivo, la implementación de estas pruebas tiende a aplazarse o incluso, cuando ya están implementadas, no son ejecutadas con frecuencia. Sin embargo, en los útlimos años han ido apareciendo herramientas mas simples de usar y las que eran complejas han mejorado también.

Es recomendable empezar con algunas pruebas no funcionales básicas e ir incrementando su complejidad a medida que el proyecto crezca y lo necesite. Siempre debe considerarse invertir un poco de tiempo investigando cómo implementar este tipo de pruebas.

3 Pruebas Unitarias

3.1 Contenido

3.2 Introducción

> Cualquiera puede escribir código que una máquina pueda entende. Los buenos programadores escriben código que humanos pueden entender. – Martin Fowler

Antes de empezar a usar herramientas complejas, patrones o APIs, debemos ser capaces de verificar que el software que construimos funciona o no. Debemos configurar nuestro ambiente de desarrollo de manera que nos entregue feedback forma rápida y sencilla. Las pruebas unitarias automáticas nos ayudan a verificar el funcionamiento correcto y a detectar efectos no deseados de manera rápida.

Una prueba en un salón de clases es una verificación de nuestros conocimentos que ayuda a determinar si estamos listos para el siguiente grado o nivel. En cuanto a software, las pruebas son la validación de los requerimientos funcionales y no funcionales antes que éstos sean entregados a los clientes.

Las pruebas unitarias de código permiten evaluar si el resultado de realizar una operación es correcta o no de manera rápida. Una prueba unitaria es una verificación simple en la que observamos si el material producido es coherente. Las pruebas unitarias son una práctica común en TDD.

Por lo general, todas las pruebas están incluidas en el mismo proyecto pero en directorios diferentes al código de producción. Algunos frameworks disponibles para realizar pruebas unitarias en Java son:

- SpryTest
- Jtest
- JUnit
- TestNG

3.3 JUnit 4+

JUnit es un framework de pruebas unitarias para Java que tiene gran popularidad. JUnit tiene, además, soporte para runners personalizados y frameworks de mocking como Mockito.

JUnit es un framework basado en anotacione por lo que cualquier clase puede ser una clase de pruebas.

3.3.1 Anotaciones

Anotación	Ámbito	Descripción
@Test	Método	Convierte en una prueba a cualquier método público anotado con @Test
@Before	Método	Se ejecuta antes de cada prueba. Permite la preparación de los datos antes de la prueba.

Anotación	Ámbito	Descripción
@After	Método	Se ejecuta despues de cada prueba. Normalmente se usa para regresar el estado de la aplicación a como se encontraba antes de realizar los cambios requeridos por la prueba.
@BeforeClass	Método	Código que se ejecuta solo una vez por cada clase.
@AfterClass	Método	Similar a @BeforeClass, puede usarse en cualquier método público, estático y que no retorna parámetros.

3.3.2 Aserciones

La aserción es una herramienta para verificar una asumpsión usando el output de un algoritmo implementado. En un ejemplo clásico, un programador espera que la suma de dos número enteros tenga como resultado otro número positivo, por lo que podría escribir una prueba con una aserción del resultado.

El paquete org.junit.Assert provee un conjunto de métodos estáticos sobrecargados para realizar aserciones entre valores esperados y reales de tipos primitivos, objetos y arrays.

Assert	Descripción
assertTrue(condition) assertTrue(failure message, condition)	Lanza un error del tipo AssertionError cuando la condición no es verdadera. Se puede proveer un mensaje.
assertFalse(condition) assertFalse(failure message, condition)	Similar a assertTrue.
assertNull()	Verifica que el objeto sea nulo, caso contrario lanza un error de tipo AssertionError.
assertNotNull()	Verificaque el objeto no sea nulo.
assertEquals(string message, object expected, object actual) assertEquals(object expected, object actual) assertEquals(primitive expected, primitive actual)	Verifica que los parametros primitivos sean iguales o ejecuta una comparación usando el método equals() para parámetros de tipo Object.
assertSame(object expected, object actual)	Sólo soporta objetos y verifica que las referencias de ambos parámetros sea la misma usando el operador ==
assertNotSame(object expected, object actual)	Similar a assertSame.

3.3.3 Manejo de excepciones

El manejo de excepciones es importante para probar condiciones de error. Por ejemplo, supongamos que deseamos probar un API que recibe tres parámetros. Si uno de los tres parámetros es nulo, el API debería retornar una excepción.

La anotación @Test puede recibir un parámetro para indicar la clase de la excepción que se espera. Por ejemplo:

```
@Test(expected=RuntimeException.class)
public void exception() {
```

```
    throw new RuntimeException();
}
```

3.3.4 Test runners y la anotacion @RunWith

Los test runners ejecutan las pruebas de JUnit. Algunos IDEs como eclipse tienen runners gráficos por defecto que permiten ejecutar las pruebas directamente.

Cuando anotamos una clases con @RunWith o extendemos una clase anotadada con @RunWith, JUnit invoca a la clase referenciada para ejecutar las pruebas en lugar de usar el runner por defecto.

La anotación @RunWith cambia la naturaleza de las pruebas. Podriamos de esa manera, correr las pruebas como si fueran pruebas parametrizadas, pruebas de spring o incluso pruebas de Mockito con objetos Mock inicializados antes de cada prueba, etc.

3.4 Spock Framework

Spock es un framework de pruebas y especificaciones para Java y Groovy orientado a BDD. Spock está basado en JUnit sin embargo hace uso de características de groovy para ampliar sus capacidad. Además puede integrarse fácilmente con SpringBoot.

3.4.1 Estructura de una prueba de Spock:

```
class FirstSpecification extends Specification {
 def "two plus two should equal four"() {
    given: "two operators left and right equal 2"
        int left = 2
        int right = 2

    when: "the operators are added"
        int result = left + right

    then: "result must be 4"
        result == 4
 }
}
```

Para mayor información visite el sitio de spock en: http://spockframework.org/

4 TDD: Test Driven Development

4.1 Índice

4.2 Introducción

El desarrollo guiado por pruebas (TDD) es un enfoque de desarrollo evolutivo que tiene como principal idea escribir primero las pruebas y escribir luego el código de producción con el objetivo de satisfacer esas pruebas.

La simple idea de escribir las pruebas primero reduce el esfuerzo extra de escribir las pruebas uniarias después del desarrollo. Los mocks, stubs y doubles son usados extensamente para simular dependencias externas. Mockito y JMockit son dos de los frameworks más populares para este fin y se utilizan en conjunto con los frameworks de pruebas unitarias como JUnit.

4.3 Ciclo del TDD

4.3.1 Crear una nueva prueba

En el ciclo de TDD, cada nueva característica comienza con la creación de una nueva prueba. La prueba define o mejora una característica. Es importante que antes de escribir la prueba, el desarrollador comprenda claramente los detalles de la especificación y los requerimientos.

4.3.2 Ejecutar todas las pruebas y comprobar que la nueva prueba falle

Esto permite verifica el funcionamiento de la bateria de pruebas. La nueva prueba no pasará sin que se escriba nuevo código para el requerimiento relacionado a ésta.

4.3.3 Escribir nuevo código

El siguiente paso es escribir nuevo código con el único objetivo de que la prueba pase correctamente. En este punto el código puede no ser muy elegante ni optmizado.

4.3.4 Ejecutar nuevamente las pruebas

Si todas las pruebas funcionan correctamente, el desarrollador puede estar seguro que su código cumple con los requerimientos sin romper funcionalidad existente. Si las pruebas fallan, el código debe mejorarse.

4.3.5 Refactorizar el código

En TDD, el codigo fuente debe limpiarse y mejorarse frecuentemente. El código debe moverse de dónde originalmente fue puesto (para pasar las pruebas) a algun lugar dónde mejor corresponda (paquetes, funciones, etc.). A su vez, el refactor debe aprovecharse para hacer el código mas leíble y mantenible.

4.3.6 Repetir

Con una nueva prueba, el ciclo se repite para empezar a crear nueva funcionalidad. Cada paso debe ser corto. Una medida eficiente es alrededor de 10 ediciones por cada ejecución de las pruebas. Si es que el código no satisface la prueba rápidamente, el desarrollador debe deshacer sus cambios en lugar de realizar depuraciones complejas.