Formación pruebas software

Guía didáctica por CRC IT

0. Prefacio

Sobre la ilustración de la cubierta

"Queda, por último, la Música. ¿Qué mayor inutilidad que unir unos ruidos con otros ruidos que no expresan directamente nada y que pueden ser interpretados de mil distintas maneras según el estado de ánimo de quien los escuche? ¿A quién alimenta eso? ¿A quién abriga? ¿A quién cobija? ¡A nadie! La Música es la más inútil, biológicamente hablando, dé todas las Artes y, por ello, por su pavorosa y radical inutilidad, es la más grande de todas ellas; la menos irracional, la más intelectual, la más espiritual, la más humana, en tanto que esto signifique superación de los seres inferiores. Porque lo cierto es que hay quien entiende, ¡equivocadamente, claro está!, por "humano"...

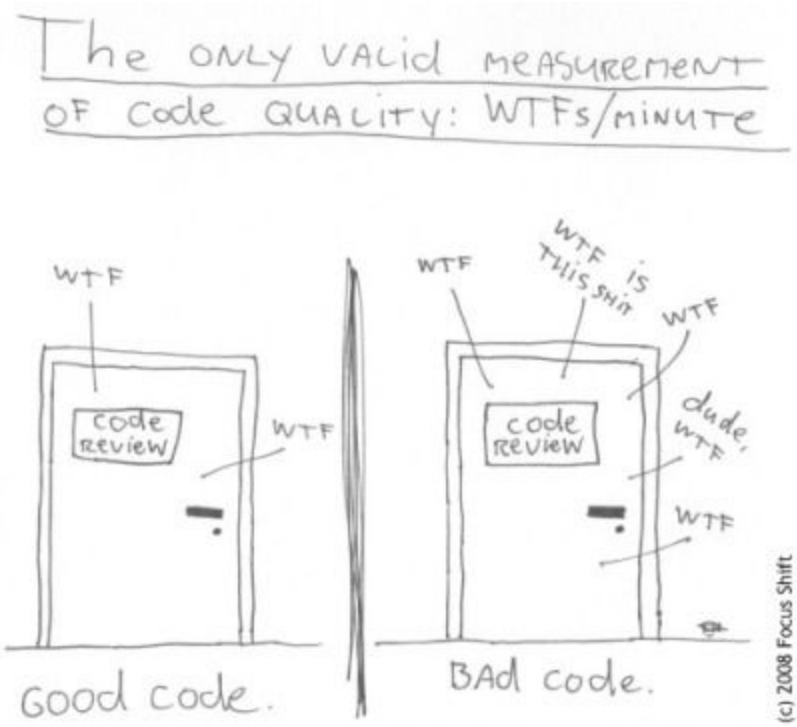
Alice Gould, Los renglones torcidos de Dios (Luca de Tena)

La ilustración de la portada es un dibujo diseñado por Sergio Valero García, con esta imagen quiero transmitir una idea inherente a la temática de esta guía y su objetivo: **La calidad del software es una forma de artesanía.**

Como cualquier otra forma de ARTEsanía el objetivo puede ser puramente práctico, terminar cuanto antes la tarea y pasar a otra cosa, podemos negar que hagamos lo que hagamos siempre nos estamos expresando.

Pero con cada pequeño detalle, cada nombre, cada pieza que encaja de una determinada manera, nos estamos expresando, y esa expresión, esa transmisión, esa parte de nosotros que queda inmerso en todo lo que hacemos es la diferencia entre artesanía y un proceso industrial.

1. Introducción



¿Qué es un test software?

Según la Wikipedia:

> *Las pruebas de software (en inglés software testing) son las investigaciones empíricas y técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad del producto a la parte interesada o stakeholder. Es una actividad más en el proceso de control de calidad.*

El objetivo de realizar pruebas software será asegurar el correcto funcionamiento de un sistema durante todo su vida útil.

¿Por qué probar?

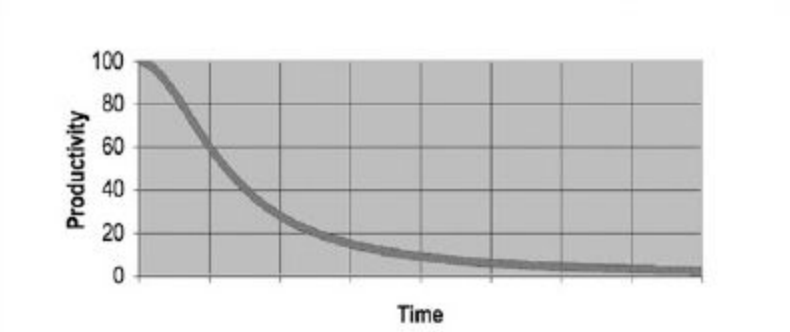
El coste total de un desastre

Si eres programador desde hace dos o tres años, habrás sufrido los desastres cometidos por otros (y por ti, en más medida) en el código (Si tienes más experiencia loS habrás sufrido en más medida), este grado de sufrimiento puede ser significativo. En un periodo de uno o dos años, los equipos que avancen rápidamente al inicio de un proyecto pueden acabar a paso de tortuga. Cada cambio en el código afecta a dos o tres partes del

mismo. Ningún cambio es trivial. Para ampliar o modificar el sistema es necesario comprender todos los detalles, efectos y consecuencias, para de ese modo poder añadir nuevos detalles, efectos y consecuencias. Con el tiempo, el desastre aumenta de tal modo que no se puede remediar. ES IMPOSIBLE.

Al aumentar este desastre, la productividad del equipo disminuye y acaba por desaparecer. Al reducirse la productividad, el director hace lo único que puede:

1. Retrasa la entrega de cada iteración cada vez más
2. Ampliar la plantilla del proyecto con la esperanza de aumentar la productividad. Pero esta nueva plantilla no conoce el sistema, ni la diferencia entre un cambio adecuado y otro que destroce el diseño. Y a más presión, y más prisa se cometen más errores y la productividad se acerca cada vez más a cero:



¿Alguna vez has tenido que superar un desastre tan grave que has tardado semanas en lo que normalmente tendrías que haber tardado horas? ¿Has visto un cambio que debería haberse realizado en una línea, aplicado en decenas de módulos distintos? Son síntomas demasiado habituales.

Hablemos de cifras

Standish Group revela en 1994 los siguientes resultados:

- Porcentaje de proyectos que son cancelados: 31 %
- Porcentaje de proyectos problemáticos: 53 %
- Porcentaje de proyectos exitosos: 16 % (pero estos sólo cumplieron, en promedio, con el 61 % de la funcionalidad prometida)

Atendiendo a estos resultados poco esperanzadores, durante los últimos diez años, la industria invierte varios miles de millones de dólares en el desarrollo y perfeccionamiento de metodologías y tecnologías. Sin embargo, en 2004 los resultados seguían sin ser alentadores:

- Porcentaje de proyectos exitosos: crece hasta el 29 %.
- Porcentaje de proyectos fracasados: 71 %.

Según el informe de Standish, las diez causas principales de los fracasos, por orden de importancia, son:

1. Escasa participación de los usuarios.
2. Requerimientos y especificaciones incompletas.
3. Cambios frecuentes en los requerimientos y especificaciones.
4. Falta de soporte ejecutivo.
5. Incompetencia tecnológica.
6. Falta de recursos.
7. Expectativas no realistas.
8. Objetivos poco claros.
9. Cronogramas irreales.
10. Nuevas tecnologías.

Código incorrecto - código limpio

¿Por qué el código de calidad se transforma tan rápidamente en código incorrecto? Nos quejamos de que los requisitos cambian comprometiendo el diseño original, que los plazos son ajustados… *Quizás una posible respuesta* es que no estamos siendo suficientemente *profesionales.* Compartimos gran parte de la responsabilidad en especial si tiene que ver con **código incorrecto.** Nuestro trabajo es defender el **código limpio** con intensidad.

Imagina que eres médico y que un paciente te exige que no te laves las manos antes de la operación porque no le parece importante y/o se pierde demasiado tiempo…

Pruebas de software y código limpio

No es fácil definir que es código limpio, ya que existen tantas definiciones como buenos programadores, pero preguntando a experimentados programadores ¿Qué es para ti código limpio?, tenemos las siguientes respuestas:

- **Dave Thomas (fundador de OTI y padrino de la estrategia Eclipse) :** El código limpio se puede leer y mejorar por parte de un autor que no sea su autor original. **Tiene pruebas de unidad y de aceptación.** Tiene nombres con sentido. [...]
- **Ron Jeffries (autor de numerosos libros como Extreme Programming Installed):** El código limpio **pasa todas sus pruebas.**
- **… Actualmente en la industria software las pruebas son un estándar de CALIDAD.**

El manifiesto ágil

Estas ideas del código limpio fueron revisadas cuando en 2001, 17 representantes de nuevas metodologías convocados por kent Beck dijeron ¡basta ya!, y se reunieron para discutir sobre el desarrollo software. Estos profesionales, con una dilatada experiencia como aval,

llevaban ya alrededor de una década utilizando técnicas que les fueron posicionando como líderes de la industria del desarrollo software. Conocían perfectamente las desventajas del clásico modelo en cascada donde primero se analiza, luego se diseña, después se implementa y, por último (en algunos casos), se escriben algunos tests automáticos.

- **Individuos e interacciones** sobre procesos y herramientas.
- **Software que funciona** sobre documentación exhaustiva.
- **Colaboración con el cliente** sobre negociación de contratos.
- **Responder ante el cambio** sobre un seguimiento de un plan.

Estos fundamentos desembocan en la **programación extrema** o ***eXtreme Programming***, y las metodologías ágiles como un camino para producir software de CALIDAD.

2. Pruebas y código limpio

Introducción

Según la programación extrema (**XP**), las metodologías ágiles y los nuevos estilos modernos de programación los test quedan lejos de ser fragmentos de código con un `@Test` encima utilizados como suplemento en la comprobación del funcionamiento de un sistema.

Así en el marco **XP** entre las responsabilidades de los test ahora contaríamos con:

- Generar documentación dinámicamente.
- Documentar el código (me dicen cómo y para qué funciona).
- Son pieza fundamental para la REFACTORIZACIÓN (**patrones**, mejoras en el código, regla del Boy Scout: **Dejar el campamento más limpio de lo que te lo has encontrado vs ¡Voy a tocar lo mínimo posible para no prepararla!**).
- Son la única manera realista de acometer **CAMBIOS** ágilmente.
- ¡Son la única manera inteligente de acometer **CAMBIOS** ágilmente!.
- ¡¡Son la única manera de acometer **CAMBIOS** ágilmente!!.
- Prueban que el correcto funcionamiento del código.
- Mediante un sistema de IC pueden ser la interfaz para una comunicación realista del desarrollo de un proyecto.
- **LOS TEST SIRVEN PARA CREAR CÓDIGO LIMPIO. CALIDAD DE SOFTWARE**

Código limpio

Perfecto, queda claro que los test nos sirven para escribir código de calidad y evitar las tragedias relatadas en el capítulo anterior, pero ¿qué es exactamente este *código de calidad?*

Mal código

- **Naming: "Debes poner el mismo cuidado nombrando a una variable que nombrando a tu primogénito",** uso de nombres que revelen las intenciones, evitar la desinformación, usar nombres que se puedan pronunciar, que se puedan buscar fácilmente, evitar " *utils ", " *Manager ", " *Info ",..

```
int d; // tiempo transcurrido en días
int elapsedTimeInDays;
```

- **Duplicidad**
- **Complejidad**: Equilibrio entre duplicidad y complejidad, mejor tener duplicidad a una abstracción inadecuada.
- **Ausencia de principios de diseño => Principio de mínima sorpresa** (WTFs/minute).

Un parámetro de lo chapucero que es un código es cuando no sabemos (o podemos) realizar test.

Buen código

Las cuatro reglas de Kent Beck son para muchos (Robert C Martin, Martin fowler, Extreme programming), fundamentales para crear un software bien diseñado. Según Kent, un diseño es sencillo si cumple las siguientes reglas (por orden de importancia):

1. **Ejecuta todas las pruebas.**

Un diseño debe generar un sistema que actúe de manera prevista. Ha de existir una forma sencilla y rápida de comprobar que realmente funciona de manera esperada. **La creación de pruebas conduce a obtener mejores diseños,** ya que afecta al cumplimiento por parte de nuestro sistema de los principales objetivos de la programación de bajo acoplamiento y elevada cohesión.

2. **No contiene duplicados.**
3. **Expresa la intención del programador.**
4. **Minimiza el número de clases y métodos.**

Según Robert C. Martin las reglas 2 a 4 abarcan el proceso de REFACTORIZAR, una vez creadas las pruebas debemos mantener el código limpio.

Principios SOLID

Es un acrónimo mnemónico introducido por Robert C. Martin a comienzos de la década del 2000 que representa cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar con el tiempo. Los principios SOLID son guías que pueden ser aplicadas en el desarrollo de software para eliminar código sucio provocando que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible. Debe ser utilizado con el desarrollo guiado por pruebas o TDD, y forma parte de la estrategia global del desarrollo ágil de software y XP

- **Single Responsibility** : Una clase sólo ha de tener una razón para ser modificada. Para contener la propagación de los cambios hay que separar las responsabilidades=> Una clase con más de una responsabilidad es más propensa a cambios => **Conocer el negocio para entender/anticipar los cambios.**
- **Open-Closed:** Las entidades de software han de estar abiertas a su extensión y cerradas a su modificación.
- **Liskov substitution:** Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas (Si **S** es un subtipo de **T,** entonces los objetos de tipo **T** en un sistema software pueden ser sustituidos por objetos de tipo **S** sin alterar ninguna de las propiedades deseables de ese programa).
- **Interface segregation:** Muchas interfaces cliente específicas son mejores que una interfaz de propósito general.
- **Dependency Inversion:** Depender de Abstracciones. No depender de concreciones (Inyección de dependencias: Patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase quien cree el objeto).

XP Programing

Para definir rápidamente el ya nombrado Xtreme Programing se podrían nombrar cuatro pilares importantes:

- Feedback.
- Comunicación.
- Respeto.
- Simplicidad

3. JUnit

xUnit

xUnit es una aglutinación de frameworks de pruebas para diferentes lenguajes y plataformas de desarrollo que cumplen el mismo objetivo, aspecto y filosofía. El impulsor de estos frameworks fue su versión para Java:Junit. Este framework fue escrito (prácticamente) en un vuelo hacia Atlanta por Kent Beck y Eric Gamma, ya que Kent quería aprender Java y Eric quería ahondar en las librerías de pruebas de Kent escritas para *Smalltalk*.

Entre otras características, aquellas que hacen de JUnit un framework idóneo para la realización de pruebas destacamos que es gratuito, está ampliamente utilizado por lo que existe mucha documentación (http://junit.org/), existen plug-ins en numerosos IDEs que facilitan su uso, es ligero, extremadamente sencillo de usar y muchas herramientas de funcionalidades añadidas (como el análisis de cobertura) utilizan JUnit como base.

Los pasos para usar JUnit en nuestros proyectos serían:

1. Incluir las librerías (http://mvnrepository.com/artifact/junit/junit).
2. Crear una clase que contendrá los métodos de prueba o *suite* (por convención terminan por Test, lo que facilita la búsqueda de suites de test a diferentes herramientas adicionales como los sistemas de integración continua).
3. Dicha clase de prueba extenderá la clase `TestClase.`
4. Crear un método público que devuelve void para cada prueba que se quiera crear y que estará anotado con *@Test.*
5. Los casos de prueba serán exitosos o fallidos dependiendo de la ejecución de una sentencia que incluirán estos métodos y que es de tipo assert. En la

documentación podemos ver todos los tipos de asserts o comprobaciones de que disponemos para validar el código a probar (http://junit.sourceforge.net/javadoc/org/junit/Assert.html).

La ejecución de las pruebas se efectuará típicamente desde el IDE que estemos usando ya que tendremos una opción para ejecutar directamente la suite de test creada. También lo podemos hacer por consola con la sentencia:

```
java org.junit.runner.JunitCore <Mi clase de pruebas>
```

API

Assertions

JUnit proporciona una serie de métodos de comprobación para objetos y primitivos. El parámetro expectativa (expected) es evaluado contra el parámetro actual, opcionalmente el primer parámetro puede ser un mensaje (String) para la salida en caso de fallo.

Un método de aserción significativamente diferente es assertThat que en lugar de un parámetro actual tiene un objeto Matcher.

No tiene ningún sentido “estudiarse” la API de los assert ni de los assertThat debido al acceso instantaneo al APi desde cualquier IDE moderno y al sencillo acceso a la funcionalidades debido a la acertada expresividad de los métodos del framework.

A modo de ilustración a continuación se lista un ejemplo de assert de diferentes tipos:

```
public class AssertTests {
  @Test
  public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    org.junit.Assert.assertArrayEquals("failure - byte arrays not same",
expected, actual);
  }

  @Test
  public void testAssertEquals() {
    org.junit.Assert.assertEquals("failure - strings are not equal", "text",
"text");
  }

  @Test
  public void testAssertFalse() {
```

```
    org.junit.Assert.assertFalse("failure - should be false", false);
  }

  @Test
  public void testAssertNotNull() {
    org.junit.Assert.assertNotNull("should not be null", new Object());
  }

  @Test
  public void testAssertNotSame() {
    org.junit.Assert.assertNotSame("should not be same Object", new Object(),
new Object());
  }

  @Test
  public void testAssertNull() {
    org.junit.Assert.assertNull("should be null", null);
  }

  @Test
  public void testAssertSame() {
    Integer aNumber = Integer.valueOf(768);
    org.junit.Assert.assertSame("should be same", aNumber, aNumber);
  }

  // JUnit Matchers assertThat
  @Test
  public void testAssertThatBothContainsString() {
    org.junit.Assert.assertThat("albumen",
both(containsString("a")).and(containsString("b")));
  }

  @Test
  public void testAssertThathasItemsContainsString() {
    org.junit.Assert.assertThat(Arrays.asList("one", "two", "three"),
hasItems("one", "three"));
  }

  @Test
  public void testAssertThatEveryItemContainsString() {
    org.junit.Assert.assertThat(Arrays.asList(new String[] { "fun", "ban", "net"
}), everyItem(containsString("n")));
  }

  // Core Hamcrest Matchers with assertThat
  @Test
```

```
  public void testAssertThatHamcrestCoreMatchers() {
    assertThat("good", allOf(equalTo("good"), startsWith("good")));
    assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
    assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
    assertThat(7, not(CombinableMatcher.<Integer>
either(equalTo(3)).or(equalTo(4))));
    assertThat(new Object(), not(sameInstance(new Object())));
  }

  @Test
  public void testAssertTrue() {
    org.junit.Assert.assertTrue("failure - should be true", true);
  }
}
```

Agregación de test en suites

Suite como runner (La clase o clases que se encargan de ejecutar los métodos de test) permite manualmente construir una suite con test contenidos en diferentes clases. Para usar esto se ha de anotar un clase con `@RunWith(Suite.class)` y `@SuiteClasses(TestClass1.class,..)`. La ejecución de esta clase será la de toda la suite configurada.

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
  TestFeatureLogin.class,
  TestFeatureLogout.class,
  TestFeatureNavigate.class,
  TestFeatureUpdate.class
})
public class FeatureTestSuite {
}
```

Expected Exceptions

Evaluar que el código cumple su flujo de ejecución normal es importante pero estar seguro de la generación de excepciones en los casos límites será adecuado también, en JUnit la sintaxis para esto es muy sencilla:

```
@Test(expected= IndexOutOfBoundsException.class)
  public void empty() {
```

```
        new ArrayList<Object>().get(0);
    }
```

Matchers and assertthat

Jow Walnes construyó un nuevo mecanismo de aserción ya comentado mediante el método assertThat. Este nos brinda más flexibilidad y refinamiento y nos ahorrará código a la hora de realizar validaciones, la sintaxis de dicho método es:

```
assertThat([value], [matcher statement]);
```

Otra ventaja de assertThat es la expresividad: poder pensar en términos de "sujeto", "verbo" y "predicado " de forma natural (assert "x is 3" en lugar de assert "equals 3 x"). Como podemos ver comparando las siguientes pruebas:

Assert vs assertThat:

```
assertTrue(responseString.contains("color") ||
responseString.contains("colour"));
// ==> failure message:
// java.lang.AssertionError:
```

```
assertThat(responseString, anyOf(containsString("color"),
containsString("colour")));
// ==> failure message:
// java.lang.AssertionError:
// Expected: (a string containing "color" or a string containing "colour")
//      got: "Please choose a font"
```

El método `assertThat` coje un objeto y la implementación de un matcher. Es el matcher el que determina si el test pasa o no pasa, es decir, es el encargado de realizar la comparación y la comprobación.

JUnit viene con una colección de matchers (de la librería Hamcrest) que podremos utilizar:

Core		
	any()	Coincidencia en algún elemento.
	is()	Coincide si un objeto es igual (equals) a otro.
	describedAs()	Añade una descripción a un Matcher.
Logical		
	allOf()	Para un array de matchers todos los objetos deben coincidir con el objeto objetivo.
	anyOf()	Igual que el anterior pero sólo requiere la coincidencia de algún elemento.
	not()	Negación de un matcher anterior.
Object		
	equalTo()	Igualdad de dos objetos dados.
	instanceOf()	Matcher de instaceOf.
	notNullValue() + nullValue()	Evalúa si un objeto es null o not null.
	sameInstance()	Evalúa si un objeto es exactamente la misma instancia que otro.

Otras funcionalidades

Y sin detenernos mucho más, ya que será la práctica la que realmente nos puede permitir un conocimiento real del framework, se reúnen otras herramientas que nos serán útiles en el desarrollo de los test:

- **@Before y @After**: Anotaciones que provocan que los métodos anotados se ejecuten antes y después de cada prueba respectivamente.
- **@BeforeClass y @AfterClass**: El método anotado se ejecutará una sola vez antes y después de todas las pruebas de la clase (suite).
- **Assumtions**: Al utilizar las asunciones estableceremos una condición previa antes de un assert, es decir (,) si la asunción no se cumple no se evalúe el assert y la prueba pasa.
- **@Rules**: Nos servirán para establecer condiciones comunes a diferentes objetos en diferentes suites.

4. Pruebas software: Teoría

Antes.. Ahora?

En 1997 nadie había oído hablar del desarrollo guiado por pruebas (TDD), las pruebas eran pequeños fragmentos de código desechable que se creaba para asegurar que los programas funcionaban. Se escribían clases y métodos y después código had hoc para probarlos.

Actualmente a nadie con suficiente conocimiento del tema se le ocurriría hacer esto, las pruebas forman parte del desarrollo de un sistema, estando este **incompleto** sin ellas (sin una verdadera refactorización, documentación, viabilidad de cambios y seguridad de funcionamiento).

Tipos de test

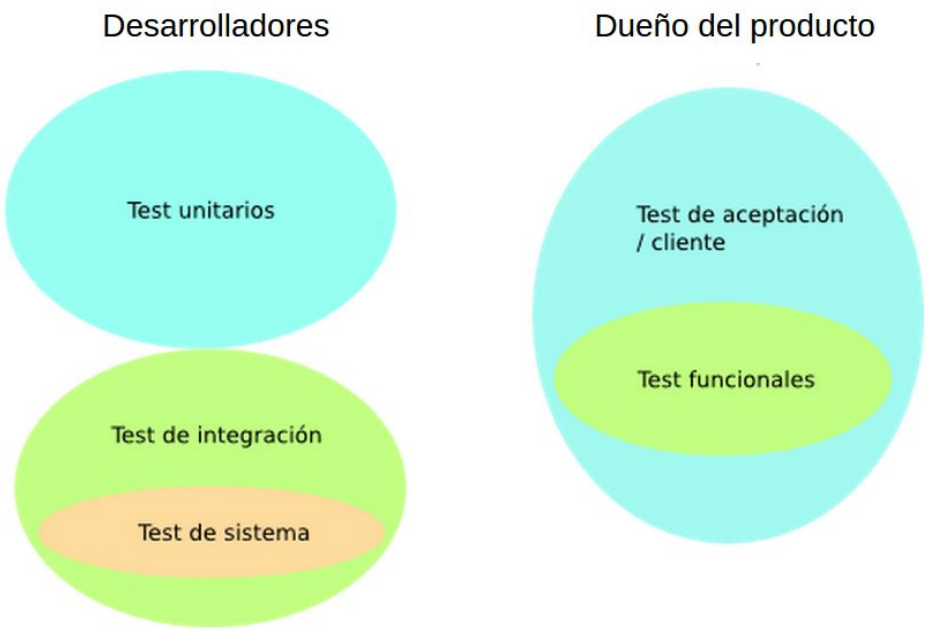
La tipología de los test es totalmente caótica. Ha sido fuente de discusión en los últimos años y sigue sin existir una categoría estandarizada. Cada equipo tiende a adoptar sus propias convenciones, ya que existen distintos aspectos a considerar para denominar los tipos de test. Por ejemplo la extensa y minuciosa clasificación propuesta por Dale H. Emery (

http://cwd.dhemery.com/2004/04/dimensions/) no tiene nada que ver con la clasificación descrita en la Wikipedia (http://en.wikipedia.org/wiki/Software_testing).

Tipología en la comunidad TDD

Desde el aspecto de la pertenencia distinguimos entre test escritos por desarrolladores y test escritos por el dueño del producto (analista o el propio cliente).

En el siguiente diagrama se muestra esta clasificación:



1. Test de aceptación

Es un test que permite comprobar que el software cumple con un requisito de negocio, es un ejemplo escrito con el lenguaje de cliente pero que puede ser ejecutado por un sistema, utilizando frameworks como concordion. (Podrían usar, no necesariamente, la interfaz de usuario; Los test de carga y rendimiento son de aceptación cuando el cliente los considera requisitos de negocio):

"Para acceder a la solicitud que se quiera modificar se deberá introducir la identificación de la solicitud. Únicamente el solicitante, o el usuario con permisos de administrador podrán realizar esta modificación"

2. Test funcionales

Todos los test son *funcionales,* ya que todos ejercitarán alguna función del SUT (Subject Under Test; El código que estamos probando). Estos son un subconjunto de los test de aceptación que comprueban alguna funcionalidad o requisito del sistema. El ámbito de los test de aceptación es mayor pudiendo referirse a aspectos no funcionales como tiempos de respuesta o capacidad de carga entre otros.

3. Test de sistema

Es el más amplio de los test de integración, ya que integra varias partes del sistema, pudiendo ir, en ocasiones, de extremo a extremo de la aplicación. Un test de sistema ejercita la aplicación tal y como lo haría un ser humano, usando los mismos puntos de entrada (la interfaz gráfica) y llegando a modificar la base de datos o lo que haya que hacer en el otro extremo.

Herramientas como Sellenium (http://www.seleniumhq.org/) o Spring MVC Test (http://docs.spring.io/spring/docs/current/spring-framework-reference/html/testing.html) nos permiten realizar este tipo de test.

Los test de sistema son muy frágiles en el sentido de que cualquier cambio en cualquiera de las partes puede romperlos, y no tienen por qué revelar estrictamente el código concreto por el que se han roto. Por lo que no es recomendable escribir un gran número de ellos, siempre que dependamos de test de integración y unitarios que tienen mayor granularidad. Si por el contrario no tenemos ningún tipo de test (caso muy típico cuando se trabaja con código *legacy*) se debe blindar el sistema antes de hacer modificaciones o integrar nuevos módulos.

4. Test unitarios

Son los test de menor granularidad, y los más importantes para TDD, un test unitario debe ser **atómico, independiente, inocuo y rápido.**

Para conseguir cumplir estos requisitos, un test unitario aísla la parte del SUT que necesita ejercitar de tal manera que el resto está inactivo durante la ejecución. Hay principalmente dos formas de validar el resultado de la ejecución del test: validación del estado y validación de la interacción, o del comportamiento. En los siguientes capítulos los veremos en detalle con ejemplos de código.

Los desarrolladores utilizamos los tests unitarios para asegurarnos de que el código funciona como esperamos que funcione, al igual que el cliente usa los tests de cliente para asegurarse que los requisitos de negocio se alcancen como se espera que lo hagan.

Los test unitarios deberían atacar y cubrir todos los aspectos del **comportamiento indivisible de negocio.**

5. Test de integración

Los test de integración son la pieza que falta para unir los test de sistema con los unitarios. Son una especie de test de sistema pequeños, y se escriben típicamente usando herramientas xUnit, y teniendo aspectos parecidos a los unitarios (deseablemente el mayor número) sólo que éstos rompen las cuatro reglas, ayudando a integrar distintas partes del sistema.

Un test de integración puedes escribir y leer de la base de datos siendo el complemento de los test unitarios donde antes habíamos "falseado" el acceso para limitarnos a atacar una lógica aislada.

Son por tanto de granularidad más gruesa que los test unitarios y serán más frágiles con lo que el número debería tender a ser menor. Deben permanecer a "suites" diferentes a los unitarios, pues serán más pesados, y al igual que los test de sistema debieran ser ejecutados preferiblemente en una máquina de integración continua.

Escritura de test

Realizar test limpios

Como comentábamos al inicio del capítulo en los 90 la filosofía predominante en la escritura de los test era: "Rápido y directo". No era necesario que las variables tuvieran nombres adecuados, ni que las funciones de prueba fueran breves y descriptivas. No era necesario que el código de los test estuviera bien diseñado. Bastaba con que funcionara. En definitiva la idea era "tener pruebas" o la idea de que "mejor tener algunas pruebas, aunque sean enrevesadas e incorrectas que no tenerlas".

Esto es un error crucial en el desarrollo del software. La única constante del desarrollo software en todos los lenguajes, sistemas y diseños, son los **cambios.** Cambios en diseño, arquitectura, requerimientos,.. Y será tan fundamental acotar el código propenso a cambios en producción, como mantener limpios lo test para que así puedan adaptarse a los cambios que vendrán.

Las pruebas cambian de acuerdo a la evolución del código, cuanto menos limpias sean, más difícil es cambiarlas, cuanto más enrevesado sea el código de prueba más probabilidades hay de que se dedique más tiempo en añadir nuevos test. Al modificar el código de producción pruebas antiguas empiezan a fallar y si el desastre impide que las pruebas se entiendan y superen, esto se vuelve un obstáculo insalvable, y acabamos probando a "la antigua usanza" habiendo sido todo el esfuerzo en vano.

El código de prueba es tan importante como el código de producción.

Los test abren posibilidades

Si las pruebas no están limpias y se mantienen, las perderás. Y al perderlas, perderás la pieza más importante para articular **cambios** en el código de producción. Sin pruebas, cada cambio es un posible error. Independientemente de la arquitectura, el diseño,.. sin pruebas no podremos estar seguros de no "romper" alguna funcionalidad del código.

Características más importantes de un test

Robert C Martín propone que hay tres elementos que hacen que un test sea limpio y correcto:

- Legibilidad.
- Legibilidad.
- Legibilidad.

La legibilidad es la pieza fundamental de los test. ¿Qué hace que una prueba sea legible?. Lo mismo que en el resto del código: Claridad, simplicidad, y expresión.

El código de la API de pruebas tiene un conjunto de estándares de ingeniería diferentes al código de producción. Después de todo, tienen diferente entorno de ejecución. El API de pruebas no se diseña con antelación, sino que evoluciona con la refactorización continua del código (en el anexo 2 hay un ejemplo de refactorización de código de pruebas).

Así el API de pruebas no tiene por qué ser *eficaz* pero ha de facilitar la legibilidad de los test, en el entorno de pruebas no tendremos la misma limitación de recursos que en el de producción. En el entorno de test los objetivos de limpieza del código deben ser **la limpieza y la legibilidad del código.**

Una regla comúnmente aplicada en la escritura de test es tener un único "assert" por prueba, las ventajas se ven claramente en el ejemplo del anexo 3: Se puede llegar de manera clara a la conclusión del test. Quizás sea más correcto intentar que aunque haya más de un assert ambos compatibilicen la misma conclusión.

F.I.R.S.T.

Las características de los test unitarios de manera que garanticen su correcto diseño cumplen el acrónimo F.I.R.S.T:

- **Fast.** Sobre todo las pruebas unitarias han de ejecutarse rápidamente, si no, no las ejecutaremos con frecuencia y se acumularán fallos (de otro lado habrá que apoyarse en un sistema de integración continua).
- **Independent.** Las pruebas no deben depender entre sí. Se deben poder ejecutar independientemente, y en el orden que se desee.

- **Repeatable.** Han de poder ejecutarse en cualquier entorno.
- **Shelf-validating.** Deben tener un resultado booleano, o aciertan o fallan, no deben redireccionar a un archivo de registro o depender de otro sistema para validar un concepto o no dar siempre el mismo resultado.
- **Timely.** Puntualidad. Deben crearse en el momento preciso. ¡**Cuanto antes!. "En desarrollo software: luego == nunca".**

A.A.A.

Las tres partes de un test se identifican con las siglas AAA (Arrange, Act, Assert).

La preparación puede estar parcial o totalmente contenida en el método SetUp, si es común a todos los métodos de la clase. Veamos un ejemplo de refactorización y adecuación de test:

```
public class NameNormalizerTests {

	@Test
	public void firstLetterUpperCase() {
		String name = "pablo lópez";
		NameNormalizer normalizer = new NameNormalizer();
		String result = normalizer.firstLetterUpperCase(name);
		Assert.assertEquals("Pablo López", result);
	}

	@Test
	public void surnameFirst() {
		String name = "enrique arnaz";
		NameNormalizer normalizer = new NameNormalizer();
		String result = normalizer.surnameFirst(name);
		Assert.assertEquals("arnaz, enrique", result)

	}
}
```

Test refactorizado:

```
public class NameNormalizerTests {

	NameNormalizer normalizer;

	@Before
	public void setUp() {
		normalizer = new NameNormalizer();
```

```
	}

	@Test
	public void firstLetterUpperCase() {
		String name = “pablo lópez”;

		String result = normalizer.firstLetterUpperCase(name);

		Assert.assertEquals(“Pablo López”, result);
	}

	@Test
	public void surnameFirst() {
		String name = “enrrique arnaz”;

		String result = normalizer.surnameFirst(name);

		Assert.assertEquals(“arnaz, enrrique”, result)
	}
}
```

Este ejemplo es muy sencillo pero nos sirve para hacer hincapié en la legibilidad del código, separando con una línea en blanco las partes de los bloques AAA, se añade legibilidad y queda claro cuál es la funcionalidad del código a primera vista. Lo que ayudará al equipo, a los cambios y a las refactorizaciones del código de pruebas.

Test de estado e interacción

La validaciones de entrada-salida o estado, como en el último ejemplo no tienen (en principio) más complicación, salvo que la ejecución del SUT provoque cambios en el sistema y tengamos que evitarlos para respetar las cláusulas del test. La filosofía será comprobar que a una entrada dada por nosotros la salida es tal y como esperamos.

Pero, ¿qué ocurre si queremos probar un método del tipo: “*void ejecutaAccion()”* ? al no poder validar la salida de ninguna entrada habrá que realizar una validación de interacción.

Las validaciones de interacción son un tipo de validaciones de comportamiento, es recomendable recurrir a esta técnica sólo cuando no podemos hacer una validación de estado, ya que estas pruebas necesitan conocer la implementación del SUT y por lo tanto son más frágiles (están acoplados a la estructura del SUT). Muchas veces se puede validar el estado aunque no sea evidente a simple vista. Quizás tengamos que consultarlo a través de alguna propiedad del SUT en vez de limitarnos al clásico ejemplo de entrada-salida.

El caso de validación de interacción más común es el de una colaboración que implica alteraciones en el sistema. Elementos que modifican el estado del sistema son, por ejemplo, las clases que acceden a la base de datos o que envían mensajes a través de un servicio web (u otra comunicación que salga fuera del dominio de nuestra aplicación) o que crean datos en un sistema de ficheros. Cuando el SUT debe colaborar con una clase que guarda en base de datos, tenemos que validar que la interacción entre ambas partes se produce y al mismo tiempo evitar que realmente se acceda a la base de datos. No queremos probar toda la cadena desde el SUT hacia abajo hasta el sistema de almacenamiento.

El test unitario pretende probar exclusivamente el SUT. Tratamos de aislarlo todo lo posible. Luego ya habrá un test de integración que se encargue de verificar el acceso a base de datos. Para llevar a cabo este tipo de validaciones es fundamental la inyección de dependencias. Si los miembros de la colaboración no se han definido con la posibilidad de inyectar uno en el otro, difícilmente podremos conseguir respetar las reglas de los tests unitarios.

Excepciones

El tratamiento de las excepciones en los test se considera validación de comportamiento, no se valida estado ni interacción de colaboradores, el objetivo será validar que un SUT devuelve una excepción concreta. Si usamos una excepción genérica, estaremos enmascarando posibles fallos del SUT.

Síntomas o heurísticos

A continuación se enumeran una serie recurrente de desajustes que se suelen dar en la escritura de pruebas:

1. **Pruebas insuficientes**: Las pruebas han de probar todo lo que puede fallar.Y han de documentar los fallos.
2. **Usar una herramienta de cobertura.**
3. **No ignorar pruebas triviales.** Son fáciles de escribir y su valor documental aporta más que lo que cuesta crearlas.
4. **Una prueba ignorada es una pregunta sin respuesta.**
5. **Probar condiciones límite.** Solo acertar en la parte central de un algoritmo y errar en sus límites.
6. **Probar de forma exhaustiva los errores.** Al detectar un error, haga una prueba que falle, y compruebe que los cambios en producción que la hacen pasar no rompen otras pruebas.
7. **Las pruebas deben ser rápidas.**
8. **Las pruebas lentas deben correr en un CI periódicamente.**

5. TDD

Descripción

El Desarrollo Dirigido por Tests (Test Driven Development), es una técnica de diseño e implementación de software incluida dentro de la metodología XP. Quizás el nombre es un tanto desafortunado; algo como Diseño Dirigido por Ejemplos , por ejemplo, hubiese sido más acertado. TDD es una técnica de diseño centrada en tres principios:

- La implementación de las funciones justas que el cliente necesita y no más.
- La minimización del número de defectos que llegan al software en fase de producción.
- La producción de software modular, altamente reutilizable y preparado para el cambio.

Cuando empezamos a leer sobre TDD creemos que se trata de una buena técnica para que nuestro código tenga una cobertura de tests muy alta, algo que siempre es deseable, pero es una herramienta de diseño para **asegurar que aplicamos la solución software correcta a unos requisitos definidos**.

No se trata de escribir pruebas a granel como locos, sino de diseñar adecuadamente según los requisitos. De pensar la API más deseable de nuestro código desde fuera. Pasamos de pensar en implementar tareas, a pensar en ejemplos certeros que eliminan la ambigüedad. Hasta ahora estábamos acostumbrados a que las tareas, o los casos de uso, eran las unidades de trabajo más pequeñas sobre las que ponerse a desarrollar código. Con TDD intentamos traducir el caso de uso o tarea en X ejemplos, hasta que el número de ejemplos sea suficiente como para describir la tarea sin lugar a malinterpretaciones de ningún tipo.

En otras metodologías de software, primero nos preocupamos de definir cómo va a ser nuestra arquitectura. Pensamos en las clases de infraestructura. ¿Y si luego resulta que no necesitamos todo eso? ¿Cuánto vamos a tardar en darnos cuenta de ello? En TDD dejamos que la propia implementación de pequeños ejemplos, en constantes iteraciones, haga emerger la arquitectura que necesitamos usar. Ni más ni menos.

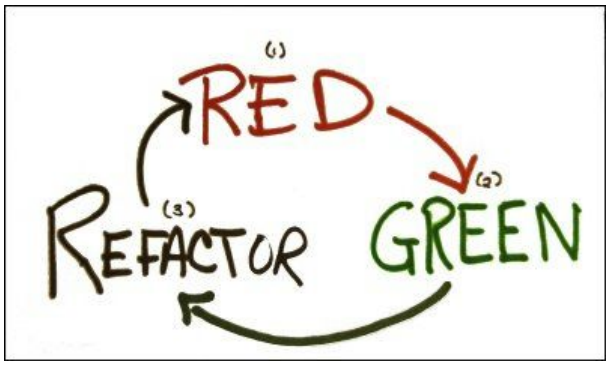
TDD produce una arquitectura que emerge de la no-ambigüedad de los tests automatizados, lo cual no exime de las revisiones de código entre compañeros ni de hacer preguntas a los desarrolladores más veteranos del equipo. Las primeras páginas del libro de Kent Beck propone unos argumentos muy claros y directos sobre por qué merece la pena intentar aplicar TDD:

- La calidad del software aumenta (y veremos por qué).
- Conseguimos código altamente reutilizable.
- El trabajo en equipo se hace más fácil, une a las personas.
- Nos permite confiar en nuestros compañeros aunque tengan menos experiencia.
- Multiplica la comunicación entre los miembros del equipo.
- Las personas encargadas de la garantía de calidad adquieren un rol más inteligente e interesante.
- Escribir el ejemplo (test) antes que el código nos obliga a escribir el mínimo de funcionalidad necesaria, evitando sobrediseñar.
- Cuando revisamos un proyecto desarrollado mediante TDD, nos damos cuenta de que los tests son la mejor documentación técnica que podemos consultar a la hora de entender qué misión cumple cada pieza del puzzle.

Algoritmo TDD

Ahora bien, todo esto no se va a cumplir mediante un conjuro TDD, como cualquier otra técnica nueva será necesario aprender las fundamentos, practicar, y refinar a través de la experiencia. La técnica en sí se puede describir de manera bastante breve, sólo tiene 4 sencillos pasos:

- Partir de una TODO LIST, o lista de ejemplos obtenidas de los requisitos directamente o tras un análisis (de funcionalidad completa).
- No se puede tocar el código de producción mientras no haya un test. Escribir la especificación del requisito (el ejemplo, el test) **[ROJO]**, esto nos hace pensar en la API más adecuada de nuestro código de producción.
- Implementar el código según el ejemplo. Al principio lo más rápido posible, imaginándonos que tuviésemos un par de minutos. **[VERDE]**
- REFACTORIZAR tranquilamente, limpiar el código. **[NEGRO]**



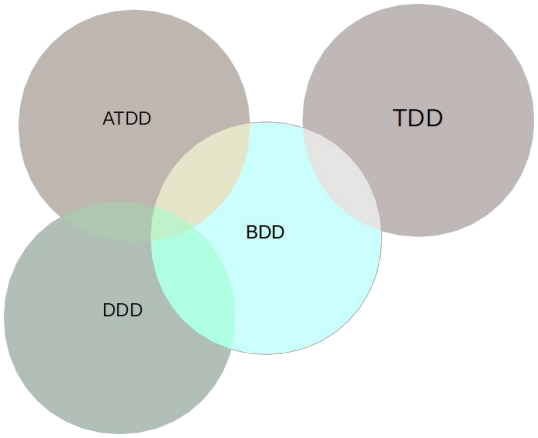
Axiomas

TDD funciona debido a constantes que se dan en todos los proyectos y en todos los programadores, y que nos deberían hacer reflexionar, se podrían destacar:

- LOS CAMBIOS son una constante en los desarrollos.
- Somos demasiado optimistas a la hora de resolver problemas.
- Nunca podemos tener en cuenta todas las relaciones e implicaciones => En todos los proyectos TODO EL MUNDO COMETE ERRORES.

Otra cuestión es el tema del tiempo, es decir ¿es realmente rentable en cuestión de tiempo? Se podría decir que actualmente está bastante establecido en las empresas software modernas la clara predominancia de las metodologías XP para el óptimo desarrollo, esto incluye ciertas técnicas como Pprograming para modulos importantes, Scrum, TDD,.. La base es que los cuellos de botella nunca serán las manos en el teclado, o el dedicar tiempo a hacer un test, los cuellos de botella son siempre las “chapuzas”, un análisis incorrecto, código que no se adecúe al cambio, en definitiva código sucio. TDD es una guía para producir código limpio.

BDD, DDD, ATDD, TDD



Para intentar contextualizar TDD dentro de su familia, presentaremos de manera esquemática en qué consiste todo este lío de siglas:

DDD

Es un enfoque para el desarrollo software basado en prácticas y terminologías para tomar decisores que adecúen el dominio software al dominio real que representa. (https://ddd-cqrs-base-project.googlecode.com/files/DomainDrivenDesignQuicklyOnline.pdf).

ATDD

El Desarrollo Dirigido por Test de Aceptación (ATDD) es TDD pero a un nivel diferente. Los tests de aceptación o de cliente son el criterio escrito de que el software cumple los requisitos de negocio que el cliente demanda. Son ejemplos escritos por los dueños de producto. Es el punto de partida del desarrollo en cada iteración, la conexión entre Scrum y XP (TDD).

No se trata de reemplazar toda la documentación, sino los **requisitos. Los requisitos son historias de usuario (ejemplos).** La definición original fue descrita por Dan North en su artículo Introducing a BDD. BDD es similar a Test Driven Development (TDD), aunque es diferente en algunos puntos sutiles, pero cruciales. Por las convenciones de TDD los métodos y las clases de pruebas reciben nombres basados en las clases de producción. BDD reorienta el enfoque al comportamiento del sistema. BDD usa una plantilla para poder pensar en el comportamiento de las pruebas del código:

- Dado (Given), un contexto inicial.
- Cuando (When), un evento se produce.
- Entonces (Then), aseguro algunos resultados

El ejemplo del artículo de Dan North es bastante ilustrativo. Supongamos la siguiente historia de usuario (user story):

Como un cliente, quiero sacar dinero del cajero automático, de modo que no necesite pasar tiempo en la cola.

¿Cómo podemos saber si esta historia está lista (el concepto de «hecho»)? Hay varios escenarios a considerar. Algunos ejemplos: que la cuenta tenga crédito, la cuenta no tiene más límite de crédito. Usando la plantilla de arriba el escenario 1 sería ese:

- Dado que la cuenta posee crédito. Y la tarjeta es válida. Y el cajero automático tiene dinero.
- Cuando el cliente pide dinero.

- Entonces asegure que la cuenta sea debitada. Y asegure que el dinero sea entregado. Y asegure que la tarjeta sea devuelta.

¿Qué significa este cambio en nuestras pruebas unitarias? Cambia, ya que podemos utilizar una herramienta como JBehave*** para desarrollar las pruebas en el formato Dado/Cuando/Entonces (Given / When / Then). La prueba unitaria está girando para el comportamiento en lugar de limitarse a probar los métodos.

Según Koskela, en su libro “Test Driven: Practical TDD and Acceptance TDD for Java Developers”, el TDD puro ayuda a los desarrolladores de software a producir código de mejor calidad y facilidad de mantenimiento. El problema es que los clientes rara vez se interesan en el código. Ellos quieren que los sistemas sean más productivos y generen retornos sobre la inversión. El ATDD ayuda a coordinar los proyectos de software de forma de entregar lo que el cliente desea.

Las pruebas de aceptación son especificaciones de comportamiento y funcionalidad deseados para un sistema. Ellos nos informan cómo, para una determinada historia de usuario o caso de uso, el sistema trata determinadas condiciones y entradas. Nuevamente según Koskela, una buena prueba de aceptación debe ser:

- Propiedad de los clientes.
- Escrito en conjunto con los clientes, desarrolladores y analistas de prueba.
- Sobre el Qué y no sobre el Cómo.
- Expresada en lenguaje de dominio del problema.
- Conciso, preciso y sin ambigüedades.

Hasta aquí hemos descrito una prueba de aceptación que podría simplemente estar dentro de un documento Word o una planilla Excel. Entonces, ¿cuál es la diferencia de la prueba de aceptación tradicional para la prueba de aceptación "clásica"? En primer lugar es que en ATDD la prueba debe ser automatizada. Este es un punto esencial, realizar las pruebas regresivas continuas. En segundo lugar, las pruebas de aceptación son escritas ANTES de la funcionalidad. Pero, ¿cómo? siguiendo los pasos a continuación:

- Tome una historia o flujo de caso de uso.
- Escriba las pruebas de aceptación en el lenguaje de dominio del cliente.
- Automatice las pruebas de aceptación.
- Implemente la funcionalidad.

Es claro que las herramientas direccionadas a ATDD ayudarían. Tenemos varias open source en el mercado, tales como Fitnesse, Exactor o Concordion.

BDD

Es un conjunto de prácticas que buscan mejorar la comunicación de los requerimientos y expresar mediante ejemplos el diseño de un software. Da soporte directo a los valores ágiles de colaboración con el cliente y respuesta al cambio, ya que ayuda a crear un lenguaje común entre el equipo que desarrolla y las personas que definen los requerimientos del negocio, permitiendo al equipo enfocarse en el desarrollo de software que verdaderamente cumpla las necesidades del cliente, y quedando las especificaciones en formato ejecutable, lo que resulta en una matriz de pruebas automatizadas que ayudan a detectar regresiones y fallas, permitiendo al equipo escribir código limpio y adaptable a cambios en el tiempo. Es decir sirve de conexión de TDD con BDD con ATDD.

6. Dobles de prueba

Antes de decidirnos a usar objetos dobles de prueba (en adelante mocks) hay que pensarlo dos veces. Lo primero, es saber en todo momento qué es lo que vamos a probar y por qué. Los mocks presentan dos inconvenientes fundamentales:

- El código del test puede llegar a ser difícil de leer.
- El test corre el riesgo de volverse frágil si conoce demasiado bien el interior del SUT.

Frágil significa que un cambio en el SUT, por pequeño que sea, romperá el test forzando a reescribirlo. La gran ventaja de los mocks es que reducen drásticamente el número de líneas de código de los tests de validación de interacción. En los tests de validación de estado, también se usan mocks y stubs cuando hay que acceder a datos procedentes de un colaborador (por ejemplo un repositorio).

Tipos de dobles

Martin Fowler publicó un artículo que se ha hecho muy popular “Los mocks no son stubs” (http://martinfowler.com/articles/mocksArentStubs.html), donde habla de los distintos dobles. De ahí extraemos el siguiente listado de tipos de doble:

- **Dummy**: se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar listas de parámetros.
- **Fake**: tiene una implementación que realmente funciona pero, por lo general, toma algún atajo o cortocircuito que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- **Stub**: proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. No tienen memoria.
- **Mock**: objeto preprogramado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. A diferencia del Stub el mock si tiene memoria. Un mock válida comportamiento en la colaboración, mientras que el stub

simplemente simula respuestas a consultas. El stub hace el test menos frágil pero, al mismo tiempo,nos aporta menos información sobre la colaboración entre objetos.

- **Spy:** un mock creado como un proxy de un objeto real, en este caso algunos métodos pueden estar mockeados, y otros hacer la llamada al objeto real.

Mockito

Mockito es la herramienta para generar mocks más extendida para Java, tiene una API expresiva, y permite generar de forma rápida y cómoda **mocks** y **spies**.

Creación de mocks

Los mocks son creados con la ayuda del método estático mock():

```
Flower flowerMock = Mockito.mock(Flower.class);
```

o mediante el uso de anotaciones:

```
@Mock
private Flower flowerMock;
```

Para el uso de @Mock, y de otras opciones de Mockito habilitadas mediante anotaciones será necesario invocar al método MockitoAnnotations.initMocks(testClass) o utilizar el runner de Mockito (MockitoUnit4Runner).

Stub de un valor de retorno de un método

Una de las funciones básicas de un framework para crear mocks es la habilidad de devolver valores cuando un método específico es llamado. Esto lo realiza mockito con el uso de Mockito.when() en conjunción con un método de tipo then…(), un ejemplo de este uso sería:

```
Flower flowerMock = mock(Flower.class);
when(flowerMock.getNumberOfLeafs()).thenReturn(TEST_NUMBER_OF_LEAFS);
int numberOfLeafs = flowerMock.getNumberOfLeafs(); assertEquals(numberOfLeafs,
TEST_NUMBER_OF_LEAFS);
```

Mockito proporciona una familia de métodos para crear diferentes comportamientos:

Método	Descripción
thenReturn(T valueToBeReturned)	Retorna el valor dado

thenThrow(Throwable toBeThrown) thenThrow(Class toBeThrown)	Lanza una excepción dada
then(Answer answer) thenAnswer(Answer answer	Usa una respuesta creada
thenCallRealMethod()	Llama al método del objeto original (spy)

AAA

Para facilitar el estilo de test AAA (BDD) Mockito tiene alias de funciones. Así, **when** se puede invocar mediante **given** y realizar las comprobaciones mediante el uso de **then.** Otra forma de escribir el ejemplo dado anteriormente es:

```
//given
Flower flowerMock = mock(Flower.class);
given(flowerMock.getNumberOfLeafs()).willReturn(TEST_NUMBER_OF_LEAFS);
//when
int numberOfLeafs = flowerMock.getNumberOfLeafs();
//then
assertEquals(numberOfLeafs, TEST_NUMBER_OF_LEAFS);
```

Verificación de argumentos

Mockito por defecto compara los argumentos mediante el método equals() de Java, en ocasiones es conveniente conocer los parámetros con los que se invoca un método, por ejemplo:

```
WateringScheduler schedulerMock= mock(WateringScheduler.class);
given(schedulerMock.getNumberOfPlantsScheduledOnDate(WANTED_DATE))
       .willReturn(VALUE_FOR_WANTED_ARGUMENT);

int numberForWantedArgument =
schedulerMock.getNumberOfPlantsScheduledOnDate(WANTED_DATE);
int numberForAnyOtherArgument =
schedulerMock.getNumberOfPlantsScheduledOnDate(ANY_OTHER_DATE);

assertEquals(numberForWantedArgument, VALUE_FOR_WANTED_ARGUMENT)
```

A menudo en lugar de pasar un argumento concreto queremos tener más flexibilidad en los valores de los parámetros. Mockito provee de una serie de **matchers** definidos en las clases **Matchers y AdditionalMatchers**, su uso lo podemos ver en el siguiente ejemplo:

```
given(plantSearcherMock.smellyMethod(anyInt(),
contains(“asparag”),eq(“red”))).willReturn(true);
```

Si para un argumento utilizamos un matcher debemos utilizar matchers para todos los argumentos del mismo:

```
given(plantSearcherMock.smellyMethod(anyInt(),
contains("asparag"), "red")).willReturn(true);
//incorrect - would throw an exception
```

Por otro lado será posible la creación de matchers extendiendo la clase ArgumentMatcher. En la siguiente tabla se muestra los matchers disponibles en Mockito:

Nombre	Reglas de coincidencia
any(), any(Class clazz)	Cualquier objeto o null
anyBoolean(), anyByte(), anyChar(), anyDouble(), anyFloat(), anyInt(), anyLong(), anyShort(), anyString()	Cualquier objeto del tipo dado o null
anyCollection(), anyList(), anyMap(), anySet()	Respectivamente cualquier tipo de colección
anyCollectionOf(Class clazz), anyListOf(Class clazz), anyMapOf(Class clazz), anySetOf(Class clazz)	Respectivamente cualquier colección del tipo dado
eq(T value)	Cualquier objeto igual al dado (equals)
isNull, isNull(Class clazz)	null
isNotNull, isNotNull(Class clazz)	cualquier objeto no nulo
isA(Class clazz)	Cualquier objeto que implemente la clase dada
refEq(T value, String... excludeFields)	Cualquier objeto que sea igual al dado usando reflexión
matches(String regex)	String que coincida con la expresión regular pasada como parámetro
startsWith(string), endsWith(string), contains(string) for a String class	String que cumpla las reglas
aryEq(PrimitiveType value[]), aryEq(T[] value)	Cualquier array que sea igual al dado (tiene la misma longitud y cada elemento es igual)

cmpEq(Comparable value)	Cualquier objeto que sea igual al dado usando el método compareTo()
gt(value), geq(value), lt(value), leq(value)	Cualquier valor mayor, mayor o igual, igual, menor o menor o igual para tipos primitivos u objetos que implementan Comparable
argThat(org.hamcrest.Matcher matcher)	Los objetos que satisfagan el Matcher pasado por parámetro
booleanThat(Matcher matcher), byteThat(matcher), charThat(matcher), doubleThat(matcher), floatThat(matcher), intThat(matcher), longThat(matcher), shortThat(matcher)	Cualquier objeto del tipo dado que satisfaga el matcher
and(first, second), or(first, second), not(first)	Para combinar diferentes matchers

Stub de múltiples llamadas al mismo método

En ocasiones podemos buscar obtener diferentes valores para una secuencia de llamadas al mismo método. La devolución de valores puede mezclarse con el lanzamiento de excepciones. Un ejemplo de uso sería:

```
WaterSource waterSource = mock(WaterSource.class);
given(waterSource.getWaterPressure()).willReturn(3, 5);

assertEquals(waterSource.getWaterPressure(), 3);
assertEquals(waterSource.getWaterPressure(), 5);
assertEquals(waterSource.getWaterPressure(), 5);
```

Stub de métodos void

Como hemos visto antes la respuesta de un método mockeado es pasada como un parámetro de **when/given**. Esto obviamente significa que no se puede usar para métodos que no devuelven nada. En lugar de esto se debería usar los métodos **willXXX...given** ó **doXXX...when.**

```
WaterSource waterSourceMock = mock(WaterSource.class);
doThrow(WaterException.class).when(waterSourceMock).doSelfCheck();

 //the same with BDD semantics
//willThrow(WaterException.class).given(waterSourceMock).doSelfCheck();

waterSourceMock.doSelfCheck(); //exception expected
```

En la siguiente tabla se muestra la familia de métodos will../do… disponibles:

Método	Descripción
doThrow(Throwable toBeThrown) doThrow(Class toBeThrown)	Lanza la excepción dada
doAnswer(Answer answer)	Usa el código de respuesta creado
doCallRealMethod()	Llama al método real
doNothing()	No hace nada

Verificación de comportamiento

Una vez creado, un mock recuerda todas las operaciones realizadas contra él. Es importante, desde la perspectiva del SUT, que estas perspectivas puedan ser verificadas fácilmente, para ello disponemos del método estático **verify()** aplicado a un método de un objeto mockeado:

```
WaterSource waterSourceMock = mock(WaterSource.class);
waterSourceMock.doSelfCheck();
verify(waterSourceMock).doSelfCheck();
```

Por defecto mockito verifica que un método es llamado una sola vez, esto puede ser modificado utilizando diferentes modos de verificación:

Nombre	Verifica que el método fue...
times(int wantedNumberOfInvocations)	llamado un numero exacto de veces
never()	nunca llamado
atLeastOnce()	llamado al menos una vez
atLeast(int minNumberOfInvocations)	llamado al menos un número exacto de veces
atMost(int maxNumberOfInvocations)	llamado un número máximo de veces
only()	la única llamada de un método de un mock

timeout(int millis)	ejecutado en un intervalo de tiempo especificado

Verificación de argumentos

Durante la verificación de interacciones, Mockito usa el método equals para validar argumentos. Esto usualmente es suficiente. También es posible el uso de **matchers** explicados anteriormente, sin embargo, en otros casos puede ser de ayuda "capturar" el valor concreto de un argumento para validarlo de una manera concreta:

```
//when
 flowerSearcherMock.findMatching(searchCriteria);

//then
ArgumentCaptor<SearchCriteria> captor =
ArgumentCaptor.forClass(SearchCriteria.class);
verify(flowerSearcherMock).findMatching(captor.capture());
SearchCriteria usedSearchCriteria = captor.getValue();
assertEquals(usedSearchCriteria.getColor(), "yellow");
assertEquals(usedSearchCriteria.getNumberOfBuds(), 3);
```

Spies

Con Mockito se puede hacer uso de los colaboradores reales en lugar de mocks, solo con algunos métodos mockeados, esto puede ser útil en algunas situaciones (con código legacy o contenedores de IoC), por ejemplo:

```
Flower realFlower = new Flower();
realFlower.setNumberOfLeafs(ORIGINAL_NUMBER_OF_LEAFS);
Flower flowerSpy = spy(realFlower);
willDoNothing().given(flowerSpy).setNumberOfLeafs(anyInt());
//stubbed - should do nothing
flowerSpy.setNumberOfLeafs(NEW_NUMBER_OF_LEAFS);

verify(flowerSpy).setNumberOfLeafs(NEW_NUMBER_OF_LEAFS);

//value was not changed
assertEquals(flowerSpy.getNumberOfLeafs(), ORIGINAL_NUMBER_OF_LEAFS);
```

Anotaciones

Mockito ofrece 3 anotaciones -@Mock, @Spy, @Captor- para simplificar la creación de objetos relevantes usando métodos estáticos. La @InjectMocks permite simplificar la inyección

de mocks y spies que de otra manera se debería hacer mediante inyección por constructor, métodos set o reflexión.

Limitaciones

Mockito tiene unas cuantas limitaciones a la hora reemplazar a objetos o métodos. Dichas restricciones tienen un carácter técnico y los autores de Mockito indican que acotan el buen diseño de código testeable y limpio.

Con mockito no podremos:

- Mockear clases finales
- Mockear enums
- Mockear métodos final
- Mockear métodos estáticos
- Mockear métodos privados
- Mockear los métodos hasCode() o equals()

7. Test de cliente

Con probar el código de cliente (cliente del patrón cliente-servidor), en un contexto de aplicaciones web, entendemos por la validación de los requisitos que tienen que ver con la ejecución de código (normalmente javascript) en un navegador. Es decir, la ejecución acotada del código de cliente no interactuando con el servidor (lo que sería validaciones de sistema y se realizarán con herramientas como Sellenium).

¿Por que probar el código de cliente?

Obviamente las mismas reglas ya comentadas sobre la escritura de “código limpio”, y TDD se aplican al código de cliente. El mismo código javascript se ejecuta en diferentes navegadores con diferentes intérpretes y ha de responder a todos ellos, así mismo javascript, como lenguaje interpretado y débilmente tipado proporciona más flexibilidad lo que se traducirá en mayor facilidad para extender, refactorizar y por lo tanto “romper” alguna funcionalidad sin darnos cuenta.

Cabe destacar el enorme uso de “plugins” de javascript para múltiples funcionalidades (bootstrap, datatable, ..) y su usual extensión para nuestros requisitos. Dichos plugins siempre suelen estar acompañados de los test que los validan y es necesaria su actualización al tiempo que añadimos funcionalidades, con lo que nos aseguramos del correcto funcionamiento del plugin en todos los ámbitos.

Jasmine

Introducción

Jasmine es un framework de desarrollo dirigido por comportamiento (BDD) para código JavaScript. No depende de ninguna otra librería JavaScript. No requiere un DOM. Y tiene una sintaxis obvia y limpia que permite escribir tests fácilmente.

En resumen, podríamos decir que desde que los señores de PivotalTracker sacaron a la luz este framework de test, prácticamente se ha convertido en el estándar de facto para el desarrollo con Javascript.

Especificaciones de Jasmine

Los test, como en otros frameworks ya comentados, están agrupados en suites que conformarán los requisitos o especificaciones (Spec) de un SUT.

Una suite empieza con una llamada a la función global de Jasmine **describe()** con dos parámetros: una String que será el título de la suite (usualmente el nombre de lo que está siendo probado) y una función que es el bloque de código que implementa la suite.

Las pruebas, denominadas especificaciones en la jerga de Jasmine (BDD) se definirán llamando a la función global **it()**, pasando como argumentos una String que será el título y la función que es la especificación propiamente dicha.

```
describe("Una suite", function() {
  it("contiene especificaciones", function() {
    expect(true).toBe(true);
  });

  it("que deben contener expectativas", function() {
    expect(false).toBe(false);
  });
});
```

A su vez las especificaciones están compuestas de una o más expectativas (asserts) que ponen a prueba el estado del código y se construyen mediante la llamada a la función **expect,** a la que se le pasa un argumento a probar o valor real, encadenada con un **matcher** que tendrá el valor esperado. Cada **matcher** realiza una comparación booleana entre el valor real y el esperado

Jasmine contiene una serie de **matchers** bastante completa y permite implementaciones propias.

Para ayudar a la limpieza de las pruebas (DRY) Jasmine proporciona las funciones globales **beforeEach, afterEach, beforeAll y afterAll** para evitar duplicados entre las especificaciones, y donde colocaremos el código de inicialización y de reseteado del SUT.

```
describe("A spec using beforeAll and afterAll", function() {
  var foo;

  beforeAll(function() {
    foo = 1;
  });

  afterAll(function() {
    foo = 0;
  });

  it("sets the initial value of foo before specs run", function() {
    expect(foo).toEqual(1);
    foo += 1;
  });

  it("does not reset foo between specs", function() {
    expect(foo).toEqual(2);
  });
});
```

Podemos definir las variables que vayamos a compartir en los bloques **describe** o en los bloques **it, beforeXXX** utilizando el contexto mediante **this:**

```
 describe("A spec", function() {
  beforeEach(function() {
    this.foo = 0;
  });

  it("can use the `this` to share state", function() {
    expect(this.foo).toEqual(0);
    this.bar = "test pollution?";
  });

  it("prevents test pollution by having an empty `this` created for the next
spec", function() {
    expect(this.foo).toEqual(0);
    expect(this.bar).toBe(undefined);
  });
});
```

Si quisiéramos ignorar suites o especificaciones tan sólo hay que añadir una x en la invocación de las funciones **it o describe**.

Jasmine permite la anidación de bloques **describe** para lograr una correcta definición de las expectativas del código probado.

Matchers de Jasmine

Las funciones **matcher()** que irán encadenadas después de las invocaciones **expect()** serán las encargadas de la validación de los valores reales y las expectativas.

A continuación se recogen los matchers que proporciona Jasmine:

Función	Descripción
toBe(null | false | true)	Comparación mediante '==='
toEqual(value)	Comparación para literales, variables y objetos
toMatch(regex | string)	Comparación para expresiones regulares
toBeUndefined() | toBeDefined()	Comparación contra 'undefined'
toBeNull()	Comparación contra null
toBeTruthy() | toBeFalsy	Comparación booleana con casting implícito
toContain(string)	Para encontrar elementos de un array
toBeLessThan(number) | toBeGreaterThan(number)	Para comparaciones matemáticas
toBeNaN()	Valor NaN
toBeCloseTo(number | precision)	Para comparaciones numéricas especificando la precisión en número de dígitos
toThrow()	Para verificar el lanzamiento de una excepción

Dobles de prueba de Jasmine

Jasmine tiene funciones para crear dobles de prueba (**spies**). Se puede crear un **spy** de cualquier función y rastrear sus llamadas y argumentos. Un spy sólo existe en el bloque **describe** que se define y se eliminará después de cada bloque **spec**. Para rastrear la interacción de estos dobles de prueba existen **matchers** especiales, como **toHaveBeenCalled ó toHaveBeenCalledWith** para rastrear las llamadas a determinados métodos

```
describe("A spy", function() {
  var foo, bar = null;

  beforeEach(function() {
           foo = {
             setBar: function(value) {
               bar = value;
             }
           };

           spyOn(foo, 'setBar');

           foo.setBar(123);
           foo.setBar(456, 'another param');
  });

  it("tracks that the spy was called", function() {
       expect(foo.setBar).toHaveBeenCalled();
  });

  it("tracks all the arguments of its calls", function() {
       expect(foo.setBar).toHaveBeenCalledWith(123);
       expect(foo.setBar).toHaveBeenCalledWith(456, 'another param');
  });

  it("stops all execution on a function", function() {
        expect(bar).toBeNull();
  });
});
```

La siguiente tabla muestra los diferentes comportamientos que puede tener el mock creado:

Método	Descripción
spyOn(obj, method_string)	Crear un spy del método dado para

	obj
spyOn(obj, method_string).and.callThrough();	El espía no sólo hace un seguimiento de todas las llamadas sino que además llama a las funciones reales
spyOn(obj, method_string).and.returnValue(value);	Al llamar al método espiado se obtiene el valor dado
spyOn(obj, method_string).and.callFake(function() { ... });	Al llamar al método espiado se delega en la función dada
spyOn(obj,method_string).and.throwError(er_string);	Idem lanzando una excepción
obj[method_string].and.stub();	Se retorna el comportamiento de stub original (vuelve a no hacer nada)

Por otro lado para rastrear las interacciones del spy además del uso de **toHaveBeenCalled ó toHaveBeenCalledWith** del ejemplo anterior podemos utilizar la propiedad **calls** del spy:

.calls.any()	false si no ha existido ninguna llamada al spy
.calls.count()	número de llamadas del spy
.calls.argsFor(index)	retorna el argumento pasado parametrizado por el índice (0 para el primer argumento)
.calls.allArgs ()	retorna todos los argumentos pasados al spy, incluso en llamadas sucesivas
.calls.allArgs ()	retorna el contexto (this) y los argumentos pasados al spy
.calls.mostRecent():	retorna el contexto (this) y los argumentos de la llamada más reciente al spy
.calls.first():	retorna el contexto (this) y los argumentos de la primera llamada
.calls.reset()	limpia todo el registro de rastreo del spy

Si se desea crear un mock pero no de una función, Jasmine proporciona dos métodos para rastrear objetos y variables que crearán mocks con las mismas características y funciones que las definidas: **createSpy y jasmine.createSpyObj.**

Otros frameworks de prueba

Para terminar, habría que mencionar otras utilidades disponibles que facilitan la escritura de test de cliente:

- QUnit: Framework de prueba más ligero que Jasmine, creado por los desarrolladores de JQuery con un estilo muy parecido a JUnit y una fácil integración con el DOM.
- Mocha: Es un potente framework de test de cliente que resalta por su flexibilidad y su facilidad para testear código asíncrono.
- karma: Es un runner que puede utilizarse con Jasmine, QUnit y Mocha, y que sirve para crear un entorno de pruebas pre-configurado para potenciar la productividad, facilitando los reports, la integración de los frameworks de prueba con otras piezas del sistema, y facilitando la retroalimentación.

8. Integración continua y calidad de software

¿Qué es la integración continua?

En palabras de Martin Fowler:

> *"Una práctica del desarrollo de software donde los miembros del equipo integran su trabajo con frecuencia: normalmente, cada persona integra de forma diaria, conduciendo a múltiples integraciones por día. Cada integración es comprobada por una construcción automática (incluyendo las pruebas) para detectar errores de integración tan rápido como sea posible. Muchos equipos encuentran que este enfoque conduce a la reducción significativa de problemas de integración y permite a un equipo desarrollar software cohesivo más rápidamente"*

Se puede asociar la integración continua (IC en adelante) con el uso de herramientas como jenkins ó team city, IC es mucho más que la utilización de una herramienta. En algunos proyectos, la integración se lleva a cabo como un evento (cada lunes integramos nuestro código ó un "clásico": un par de días antes de un pase integramos nuestros códigos y "probamos"), la práctica de la IC elimina esta forma de ver la integración, ya que forma parte de nuestro trabajo diario y queda automatizado.

La IC encaja muy bien con prácticas como TDD dado que se centra en disponer de una buena batería de pruebas y en evolucionar el código realizando pequeños cambios a cada vez.

Empezando con IC

Imaginemos que decidimos agregar una pequeña funcionalidad a nuestro software. Comenzamos descargando el código actual del repositorio a nuestra máquina local (working copy). En nuestra copia local, añadimos o modificamos el código necesario para realizar la funcionalidad elegida y para que esté completo sus pruebas asociadas.

Ya estamos listos para subir nuestros cambios al repositorio, sin embargo, otros desarrolladores han podido subir sus cambios mientras nosotros realizamos nuestra tarea. Por lo tanto debemos bajarnos los cambios del repositorio, resolver los conflictos si los hubiera y lanzar de nuevo nuestras pruebas.

Finalmente, podemos subir nuestros cambios al repositorio. La “máquina de integración” basándose en el código actual del repositorio lanzará todas las pruebas (unitarias, integración, sistema) tanto las antiguas que protegen las funcionalidades antiguas como las nuevas lo que, con una adecuada cobertura, protege todos los requisitos funcionales del software.

En caso de haber “roto” algo, o haber olvidado subir un fichero, o cualquier anomalía, el IC detecta el fallo y nos lo reporta al subir el código al repositorio con lo que debemos arreglarlo antes de dar por terminado nuestro trabajo.

De esta forma, disponemos de una base estable en el repositorio del cual cada desarrollador partirá para realizar su trabajo diario. Llegados a este punto, es posible que piense que es un proceso latoso. Pero al igual que con la metodología TDD se trata de automatizar los procedimientos de construcción y los beneficios son enormes: para cualquier funcionalidad nueva, o cambio, la tarea de “probar” queda delegada al IC.

Así, el proceso de construcción implica mucho más que compilar, puede consistir en compilar, ejecutar pruebas, usar herramientas de análisis de código , desplegar... entre otras cosas. Un build puede ser entendido como el proceso de convertir el código fuente en software que funcione.

Los test son una pieza vital de la construcción

Ya se ha hecho hincapié, aunque sea de manera indirecta, en que los tests forman parte de la construcción. Sin embargo, no está de más reafirmar la importancia de éstos en el proceso. Es muy difícil tener una larga batería de test que prueben todas las partes del proyecto (100 % de cobertura) o que todas estas sean perfectas. Pero, como bien dice Martin Fowler:

"Pruebas imperfectas, que corren frecuentemente, son mucho mejores que pruebas perfectas que nunca se han escrito".

Aunque esto no supone que debamos dejar de mejorar nuestras habilidades para desarrollar pruebas de mejor calidad. Es necesario automatizar la ejecución de las pruebas para que formen parte de la construcción.

Buenas prácticas:

- Estructura el proyecto por cada tipo de test para lanzarlas por separado.
- Escribe también pruebas para los defectos/bugs que irán saliendo antes de corregirlos y como garantía del arreglo del fallo.

Construcción en una CI

Al delegar la construcción a la CI se resuelven los fallos del tipo: ¡En mi máquina funciona!: La construcción válida, visible para todo el mundo, con un despliegue para poder acceder a la aplicación y ver informes de todas las pruebas está en la CI.

Usando un servidor de IC, cada vez que alguien sube sus cambios al repositorio, se realiza la construcción de manera automática, notificando del resultado del proceso (por e-mail, jabber, etc).

IC para reducir riesgos

- Despliegues demasiado "complicados": El despliegue se realiza periódicamente lo que (si se trata de manera adecuada los entornos de producción y desarrollo) elimina la incertidumbre del despliegue el temido día del pase.
- Descubrir defectos demasiado tarde: Todo el código está integrado todo el tiempo, todas las pruebas se están ejecutando para cada cambio, y el código está desplegado y es accesible durante todo el desarrollo ¿Hay una manera más efectiva de tratar de reducir los riesgos?
- Baja calidad del software: Desarrollar código de baja calidad suele producir un elevado coste en el tiempo (el mayor coste de tiempo), esta afirmación puede parecer gratuita para muchas personas pero el programador que haya regresado a modificar o leer esa parte del código que no huele bien, no la verá como tal. Disponer de herramientas automatizadas que analicen el código, pudiendo generar informes sobre asuntos tales como código duplicado, etc. puede ser crucial a la hora de gestionar estos problemas y solventarlos antes de que nuestro proyecto se convierta en un enorme pantano y la productividad sea una catástrofe.

9. Anexos

Anexo 1. Git

Las prácticas de esta guía se han construido y están pensadas para ser realizadas bajo versionado en git.

El código de los enunciados y las soluciones se encuentra en el repositorio:
http://git.crcit.es:9000/rvalero/cursoTest.git
Todos los enunciados de las prácticas se encuentran el en repositorio:
http://git.crcit.es:9000/rvalero/enunciadosCursoTest.git

El uso de git no es necesario aunque por supuesto si (sí) recomendable, con lo que se adjunta el siguiente recetario:

INSTALACIÓN
https://git-scm.com/downloads

CREACIÓN
Clonado de un repositorio existente

```
git clone <repositorio>
```

Creación de un nuevo repositorio local

CAMBIOS LOCALES
Cambios en tu directorio de trabajo

```
git status
```

Añadir todos los cambios al próximo commit

```
git add .
```

Commit de los cambios cargados

```
git commit -m “<mensaje>”
```

Anexo 2. Refactorización de pruebas

Método de prueba de partida:

```
@Test
public void turnOnLoTempAlarmAtThreashold() {
       hw.setTemp(WAY_TOO_COLD);
       controller.tic();
       Assert.assertTrue(hw.heaterState());
       Assert.assertTrue(hw.blowerState());
       Assert.assertFalse(hw.coolerState());
       Assert.assertFalse(hw.hiTempAlarm());
       Assert.assertTrue(hw.loTempAlarm());
}
```

Método refactorizado:

```
@Test
public void turnOnLoTempAlarmAtThreashold() {
       wayTooCold();
       assertEquals("HBchL", hw.getState());
}

[...]
public String getState() {
       String state = "";
       state += heater ? "H":"h";
       [...]
       return state;
}
[...]
```

Anexo 3. Cheat sheet Mockito

mock/@Mock: Crea un mock

- Opcionalmente se especificará su comportamiento via **Answer/ReturnValues/MockSettings**
- **When/Given** para verificar las interacciones del mock

spy()/@Spy: Para realizar un mock que no interviene en la interacción de los colaboradores y puede validar sus comportamientos.

@InjectMocks: Automáticamente injecta los mocks/spies anotados con **@Mock/@Spy.**

`verify:` comprueba las llamadas hechas a métodos y sus argumentos.

- **`any()`** se puede utilizar como comodín para cualquier valor/tipo de argumento a la hora de representar una llamada a un método.
- o se puede capturar el argumento mediante **`@Captor.`**