¡Te doy la bienvenida!

Muchas gracias por tu interés en esta guía. Para mí es un honor poder ayudarte a avanzar en tus conocimientos para que puedas crear código de calidad.

La guía está basada en tres artículos que Sergio Martínez escribió en el blog de DevExperto, por lo que todo el crédito del contenido es suyo.

Esta guía te ayudará a dar los primeros pasos para iniciarte en el mundo del testing.

¿Quién soy yo?

Mi nombre es Antonio Leiva, y llevo desde el 2012 ayudando a otros desarrolladores a evolucionar en su carrera profesional. Soy formador y Kotlin Certified Trainer por JetBrains.

Escribí hace un tiempo un libro llamado **Kotlin for Android Developers**, en el que explico cómo pasar de Java a Kotlin. También existe el curso online y la formación para empresas.

Además **soy el creador del programa Architect Coders**, donde ayudo a desarrolladores Android a convertirse en verdaderos ingenieros, dominando todos los conocimientos que las grandes empresas y startups del sector demandan. Puedes apuntarte aquí para que te informe de la próxima edición.

Si quieres conectar conmigo en las redes sociales, puedes encontrarme en:

- Instagram
- YouTube
- Canal de Telegram
- Twitter
- Facebook
- Podcast

Y si has llegado a esta página por otra vía, puedes suscribirte aquí y llevarte de regalo la guía para iniciarte en Kotlin.

¡Empezamos!

Testing nivel 1: un mundo de sensaciones por descubrir

por Sergio Martinez Rodríguez | Software Craftmanship

El testing es una técnica de validación de nuestro código que, aunque lleva existiendo desde hace mucho tiempo, no ha sido hasta hace pocos años que ha empezado a cobrar la importancia que se merece. Todo desarrollador de calidad debe conocer los conceptos principales sobre el testing y ser capaz de utilizarlos en su día a día. Así que si nunca habías oído hablar del testing, o bien te suena el concepto pero nunca has adentrado en él, te recomiendo que no te pierdas esta guía.

En este nivel vamos a empezar hablando sobre testing en general. Sobre por qué hacer tests al software que desarrollamos, sobre sus ventajas, vamos a destapar mitos y

falsas creencias acerca del testing. Después veremos los tipos de test que hay y en que consiste cada tipo. De todos esos tipos nos vamos a centrar en el más importante, aunque es cierto que todos son importantes… Vamos a centrarnos en hablar sobre Test unitarios, ¿por qué catalogamos a los unitarios como los más importantes? ¡¡sigue leyendo y lo descubrirás!!

¿Por qué la necesidad de hacer test?

- Porque nos ayudan a comprobar que lo que desarrollamos hace lo que tiene que hacer: evitamos comportamientos inesperados y excepciones.
- Nos permiten desarrollar las funcionalidades de una manera completa sin atarnos al tiempo que requiere ejecutar el software completo. Por lo tanto el tiempo de testing no es una pérdida de tiempo, al final es tiempo que ganamos, por este y por más motivos.
- Nos obliga a que las responsabilidades se distribuyan de una manera uniforme y más correcta a lo largo de nuestro código.
- Obtenemos como resultado código que sólo por haberlo desarrollado de un forma testable cumple muchos de los principios SOLID. Además este código desarrollado de esta forma nos va a ayudar en muchas ocasiones a detectar code smells.

- Más facilidad a la hora de acometer cambios que no requieren que la funcionalidad cambie, o lo que es lo mismo podremos hacer refactor de nuestro código sin miedo a que se vea alterado el funcionamiento del resto del sistema.

SOLID y el testing

Como ya se ha hablado bastante de SOLID en la serie de artículos publicados con anterioridad, no voy a ponerme pesado con este tema, porque ya es de todos los seguidores conocido que aplicar estos principios nos aporta infinidad de ventajas. Si alguien no ha visto los 5 artículos de SOLID, está aún a tiempo de verlos antes de seguir leyendo por aquí.

Nos vamos a centrar en remarcar desde el punto de vista del testing lo que nos va a aportar respetar cada principio.

Principio de responsabilidad única: Para los test es importante que las responsabilidades estén segregadas y las clases y métodos hagan el menor trabajo posible. Esto es así porque cuando algún punto de nuestro código hace muchas labores esto quiere decir que los resultados pueden ser varios. Esto se traduce a testing en muchos casos posibles de salida que hay que testar.

Recuerda: Cuando hay diferentes entradas y puede haber diferentes caminos o acciones. El número de combinaciones de

estados y comportamientos al final de una ejecución crece de manera exponencial.

Por eso es importante que nuestras clases y métodos respeten SRP.

El principio de abierto cerrado: si nuestro código crece a medida que evolucionamos nuestro software, es posible que no estemos distribuyendo bien las responsabilidades y que no estemos encapsulando lo que varía, por lo tanto vamos a tener que modificar nuestros tests, porque nuestro código testado está cambiando.

Recuerda: Cambiar en numerosas ocasiones un test cuando nuestro software crece por extensión y no por modificación directa de requerimientos es un claro indicador de la violación del principio abierto/cerrado.

Principio de sustitución de Liskov: como se ha comentado en el post de este principio, es muy fácil comprobar que se está violando cuando hacemos tests puesto que cuando hacemos test sobre la clase padre que no funcionan sobre la clase hija vemos una clara violación del mismo. Esto nos va a obligar a realizar nuevos test sobre algo que en principio estaba testado, lo cual no suena muy bien y empieza a no ser mantenible.

Recuerda: a la hora de usar la herencia, y sobre todo la sobreescritura de métodos de los supertipos ten en cuenta que

tienes que pasar test sobre eso que vas a heredar y sobreescribir y piensa dos veces si es la mejor manera de hacerlo, o simplemente es mejor que hagas uso de una composición.

Principio de segregación de interfaces: si tenemos métodos que no sirven para nada en clases que sólo están ahí como consecuencia de implementar una interfaz de la cual nos interesa un sólo método, está claro el primer problema, como ya se ha visto eso genera un boilerplate indeseable… Pero a efectos de test, es incluso peor ¿Vamos a testar esos métodos boilerplate que no hacen nada?, ¡¡que cosa más fea!!, o testamos algo que no vale para nada o hacemos que baje nuestra cobertura de código testado por culpa de un mal diseño…

Recuerda: cada método de una interfaz es siempre un claro candidato a ser testado. Por lo tanto es bueno que nuestras interfaces sean concretas y específicas. De lo contrario tendremos varias implementaciones innecesarias que tendrán test inservibles.

Principio de Inversión de dependencias: sin duda este principio va a ser nuestro amigo fiel en la implementación de nuestros test. Es el que nos va a permitir adaptar el test a nuestras necesidades y recrear exactamente los escenarios que necesitemos para testar exclusivamente lo que queremos testar. Gracias a este principio nosotros establecemos el alcance o “Scope” de nuestro test. Se testará tanto código real como nosotros decidamos.

Recuerda: que nuestras implementaciones concretas no dependan del sujeto bajo pruebas sino de una abstracción de mayor nivel nos abre la puerta en tiempo de test a servir implementaciones exclusivas para el contexto de testing que nos permitan ejecutar sólo el código real que queremos testar.

Tipos de Test

Test de integración

Los test de integración nos ayudan a automatizar procesos en los cuales intervienen otros sistemas para detectar fallos o casos no esperados en la interfaz de comunicación. Testan como se integra o interacciones de nuestro sistema con un agente externo. Entre sus principales características encontramos:

- No son especialmente rápidos a la hora de ejecutarse.
- Requieren una preparación específica para crear un contexto de prueba que replique las condiciones que deseamos. Ej: podemos ejecutar un test de integración contra la BD que soporta Mongo y usa una caché externa y querer hacer un "Test-double" dé la pieza de Mongo para forzar a trabajar a la caché con unos resultados concretos.

Test de aceptación

Estos test también se suelen llamar de caja negra, puesto que validan resultados a acciones directas del usuario acorde a la descripción del negocio tratando el sistema como una caja negra.

Validan los requisitos del negocio, desde el punto de vista del usuario final. Tanto el resultado de las acciones como la apariencia final de la app. Pueden englobarse aquí los test de instrumentación, UI, pixel perfect, y alguno más.

Aunque el caso concreto de lo que se quiera testar sea muy reducido, de manera indirecta el test está pasando por gran parte de nuestro código, y de hecho está desarrollando una prueba end to end pero lo que pasa entre la acción y el resultado es una caja negra.

Test unitarios

Aquí viene la fiesta…

Son test también llamados de caja blanca. Se centran en probar que nuestro código clase por clase y método por método hace lo que tiene que hacer. Comprobando que el comportamiento y el estado del sistema sean los esperados. Es decir, a unas entradas,

tras una ejecución esperamos unas salidas o que se hayan ejecutado otras “N” cosas.

Al principio del post catalogaba a los test unitarios como los más importantes y resulta que todavía no he dado un por qué, quizá muchos ya lo intuyen… La explicación es sencilla de contar pero llevarla a cabo no es tan sencillo y es algo que debemos ir consiguiendo de manera progresiva.

Si nuestro código está desacoplado del framework y de los agentes externos al máximo y la dependencia entre estos y nuestro código es la mínima, en otras palabras, si somos capaces de aislar nuestro software; simplemente con la cobertura que nos brindan los test unitarios cubriremos la mayor parte de nuestro código.

Además los test unitarios son rápidos y por lo tanto los podemos correr muy a menudo (con cada commit por ejemplo) para comprobar si todo sigue estando correcto. Llegar a desarrollar el 100% para que esté desacoplado y sea testable es algo que se va adquiriendo con la práctica, no te desesperes, la práctica hace la perfección.

Como podemos ver los test unitarios nos aportan infinidad de ventajas y son un punto ideal para introducirse en el mundo del testing. Con ellos podemos ir desde los conceptos más básicos

hasta desarrollar pipelines de testing complejas, versátiles y muy potentes.

Es por eso que en los siguientes niveles vamos a ir desgranando los test unitarios desde lo más básico hasta conceptos algo más complejos, pasando por algún consejo para no caer en fallos comunes.

Así que si quieres aprender de que van las “palabrejas” que ya hemos ido usando por aquí como “Test-double”, “Scope” y muchas cosas más, solo tienes que seguir leyendo.

Testing nivel 2: ¿Qué hace que un test sea un Test?

En el nivel sobre testing estuvimos hablando sobre el mundo del testing en general, de sus beneficios, de los tipos de test y de las ventajas que nos ofrece SOLID de cara al testing. Ahora que ya tenemos una visión global es hora de **profundizar en los test unitarios**.

En esta ocasión vamos a usar definiciones y conceptos teóricos, que son aplicables al código que hacemos. En un primer momento pueden ser duros de entender, sobre todo si es tu primera toma de contacto con el mundo de los test; por eso mismo, si al acabar te sientes algo confuso, te invito a releer el artículo una segunda vez para que veas que todo empieza a cobrar más sentido. Vamos «al turrón».

Un poco de vocabulario, para empezar…

Sujeto bajo pruebas: el sujeto bajo pruebas es la clase testada. Es decir, el actor principal que va a ser el objeto de todos nuestros test. En inglés se dice «Subject Under Test» y sus iniciales son

SUT, por lo que no te extrañe ver esas iniciales más de una vez a lo largo del post.

Cobertura de código: es la parte de código o **caminos** de ejecución que han sido **testados**.

Alcance del test: o también llamado en inglés «**Scope**» del test. Es lo que abarca en cobertura de código la ejecución de nuestro test. Es importante tener consciencia de la importancia de este concepto y su estrecha relación con la palabra test unitario. En otras palabras, un test **unitario puro** tiene unas **fronteras** o límites que no salen más allá del propio SUT, por lo tanto podemos deducir que su Scope es reducido.

Teniendo claros estos tres conceptos es hora de abordar los tipos de test. Vamos con ello…

Tipos de Test unitarios

Vamos a clasificar los tipos de test unitarios dentro de 2 grupos perfectamente compatibles y complementarios: **según su interacción** con el resto de test y **según** aquello **que van a testar o su intención.**

Interacción

- **Aislados, solitarios, no sociables:** son la base de los test unitarios, también los podemos llamar test **unitarios puros**. Hemos de empezar por hacer test de este tipo ya que van a ser el punto de partida. **El Scope de estos test es la unidad mínima testable**, es decir sólo testarán el SUT sin cruzar sus fronteras. Por lo tanto, estos test se encargan de validar en un **entorno aséptico** y no contaminado comportamientos o estados de una pieza de código concreta. Si fallan es porque lo que estamos testando tiene algún error. **No son susceptibles de verse afectados por efectos de lado** producidos por algún colaborador del test, porque todos los colaboradores están bajo nuestro control en el test y no usan código real, sólo se encargan de **generar un entorno idóneo para nuestro caso de test**. Veremos cómo conseguir eso más tarde en la sección de **«colaboradores de nuestros tests».**

- **Sociables o en colaboración:** En estos test pueden interactuar varios de los SUT ya testados de forma que el **scope del test es bastante más amplio**. Estos test tardan más tiempo en pasar y son susceptibles de producir **fallos en cascada**, como podremos imaginar, además su mantenimiento suele ser más costoso. A cambio nos dan una **cobertura de código más amplia**.

No es suficiente con hacer sólo test no sociables. La experiencia nos demuestra que **dos piezas probadas con test unitarios no sociables, no nos asegura que combinadas vayan a funcionar en nuestro sistema como es de esperar**. Es por eso que partiendo

de la base de los test unitarios debemos ir avanzando y conectando piezas en test sociables para ver si encajan como si de engranajes en un reloj suizo se tratasen: a la perfección, otorgando el resultado esperado por el sistema.

Intención

- **De verificación de estado: comprueban que a una entrada el resultado tras la ejecución es la salida esperada** o lo que es lo mismo, un estado (variable) del sistema tras una ejecución queda en otro estado. Las comprobaciones de este tipo son las que llamamos «Asserts» o «asertar».
- **De verificación de comportamiento:** en estos test en lugar de testar en que estado ha quedado el sistema simplemente **verificamos que se han hecho una o varias llamada, es decir, estamos verificando que el sistema se comporte como esperamos**. Estos test suelen identificarse con la palabra clave «Verify» o verificación.

Estructura de un Test : «AAA»

La estructura de un test, o lo que es lo mismo, **el cuerpo del test**, las partes claramente diferenciadas se dividen principalmente en tres, las vamos a definir ahora y siempre han de **respetar el orden** que aquí exponemos:

- **«Arrange» o preparación:** hacemos los **preparativos** para que lo que vayamos a testar esté todo lo aislado que queramos y **no cruce fronteras que no nos interesen**. Marcamos el límite de nuestro sujeto bajo pruebas y **preparamos la entrada** que le vamos a dar. En concreto **preparamos el «Escenario de test»**

- **«Act» o Llamada al método: ejecutamos** la acción que queremos testar en el sujeto bajo pruebas.

- **«Assert» o comprobación:** tras la ejecución **vemos que el sistema ha quedado en el estado esperado** a las entradas o que se ha comportado de la manera que esperamos porque ha disparado una o varias acciones. Véase varias como las mínimas necesarias. Ej: un test que hace 10 verificaciones de comportamiento o comprobaciones de estado es un smell de código, porque está haciendo demasiado trabajo. Esto quiere decir que probablemente el método que verifica haga demasiado también.

Como podrás imaginar, se suelen preparar muchos escenarios de prueba diferentes, pero es muy posible que parte del preparativo sea común para estos diferentes escenarios. Es importante reutilizar nuestro código también en nuestros test.

Buenas Prácticas

La importancia de los nombres

El nombre que damos a nuestros test importa, y mucho. Porque de lo anterior podemos deducir que vamos a tener que preparar varios test contra un mismo sujeto de prueba. Esto nos lleva a pensar que los nombres son más que importantes: han de describir el escenario en relación con lo que esperamos obtener y la acción ejecutada. Por ejemplo: `shouldObtainEmployeeListWhenManagerIsNotEmpty()` es un buen nombre para un método de test.

Como hemos dicho el sujeto bajo pruebas tendrá varias pruebas que configuran diferentes escenarios y salidas esperadas. De ahí que las clases de test tengan que tener como nombre el sujeto bajo pruebas y como nombres de métodos lo anteriormente descrito. Además es aconsejable que la ruta en el directorio de test sea lo más similar posible a la ruta del código real, para poder ubicar los test de una manera rápida y sencilla.

Siguiendo esto **conseguiremos que los test sean documentación para nuestro código** también. Algo que puede parecernos poco interesante a priori, pero que es muy de agradecer cuando volvemos sobre nuestro código al tiempo o cuando entra un nuevo compañero al proyecto.

Reutilizar nuestro código de test

Es la clave para hacer test fácilmente mantenibles e ir desarrollando cada vez con menos esfuerzo. Pero ten **paciencia**, el escribir test es algo diferente a la programación a la cuál estamos acostumbrados. Al principio costará y habrá que hacer mucho refactor sobre la forma de nuestro test, pero es una técnica que se va puliendo y como resultado puede dar lugar a sorprendentes resultados.

Crear una suite de herramientas para test es una buena práctica que podemos adoptar al coger expertise en el desarrollo de testing. En esta librería o librerías estaría bien colocar operaciones que solemos hacer con frecuencia: tratar fechas, validar ciertos estados por ejemplo cadena vacía o null… Esto puede evolucionar tanto como imaginemos. Hasta una pipeline de desarrollo para testing en la que podamos intercambiar piezas en tiempo de ejecución. Cuando usamos herramientas del estilo de «Harmcrest» es indispensable diseñar nuestros propios «matchers» y pueden ser carne de esa suite de herramientas. Puedes encontrar un par de enlaces al final del artículo interesantes relacionados con esto.

Dentro del mismo sujeto bajo test hay varios escenarios que comparten configuraciones similares aquí seguro que tiene sentido encapsular lo que varía de unos test a otros y usar las piezas encapsuladas según convenga para preparar los escenarios. También se pueden usar métodos de tipo `setup` y

`tearDown`, que nos ayudan a crear condiciones iniciales y a «limpiar» el escenario una vez ejecutado el test. Sin embargo no debemos olvidar que el ámbito de esto es global al sujeto bajo pruebas y que se ejecutan siempre antes de cada test, por lo que podemos estar ejecutando siempre código que sólo los vale para un test.

Practica el «TOD»

«Tod» Es un concepto que me acabo de inventar y no viene en los libros, no intentes buscarlo, son las iniciales de «Test Oriented Development» y viene a decir algo tan simple como:

«Cuando programamos tests también programamos, y debemos seguir haciéndolo de la manera ordenada y limpia que requiere aquello que estamos haciendo: tests»

Por lo tanto **con frecuencia habrá que construir entidades que sean necesarias para aislar nuestro SUT**, o para emular ciertas entradas que le queremos dar. Esto desemboca en nuevos objetos,»builders», «Test doubles», estructuras de datos y utilidades que al fin y al cabo **colaboran en la construcción y ejecución de nuestro test**.

Colaboradores de nuestros tests

«Podemos llamar colaboradores de un test a aquellos objetos que participan de manera pasiva en el desarrollo del mismo. Es decir, nuestro SUT hace uso de ellos, pero no entran en el SCOPE de nuestro test». También los podemos llamar **«Test doubles».** Como podrás imaginar los test sociables usan menos «Test doubles» que los unitarios puros.

En la ejecución de nuestro test, intercambiamos todo lo que no sea nuestro SUT por piezas como las que vamos a ver ahora, sobre las cuales tenemos control y además nos aportan información relevante sobre la ejecución del test.

- **Dummy:** No hace nada, esta vacío y devuelve vacío.

```
1. public class DummyAuthorizer implements Authorizer {
2.
3. @Override
4. public Boolean authorize(String username, String
   password) {
5.  return null;
6. }
7. }
```

- **Stub:** No hace nada más que devolver un valor por defecto.

```
1. public class AcceptingAuthorizerStub implements
   Authorizer {
2.
3. @Override
```

```
4. public Boolean authorize(String username, String
   password) {
5.  return true;
6. }
7. }
```

- **Spy:** Introducimos un parámetro al que tenemos acceso para que nos informe de algo que ocurre dentro del test.

```
1. public class AcceptingAuthorizerSpy implements Authorizer
   {
2. public boolean authorizeWasCalled = false;
3.
4. @Override
5. public Boolean authorize(String username, String
   password) {
6.  authorizeWasCalled = true;
7.  return true;
8. }
9. }
```

- **Mock:** Añadimos funcionalidad a un objeto que cumple la interfaz de SUT sólo para el propósito del test.

```
1. public class AcceptingAuthorizerVerificationMock
   implements Authorizer {
2. public boolean authorizeWasCalled = false;
3.
4. @Override
```

```
5. public Boolean authorize(String username, String
   password) {
6.  authorizeWasCalled = true;
7.  return true;
8. }
9.
10.   public boolean verify(){
11.    return authorizeWasCalled;
12.   }
13.   }
```

- **Fake:** Implementa una funcionalidad que puede emular a la realidad porque funciona según los datos de entrada se reciban, pero no es real.

```
1. public class AcceptingAuthorizerFake implements
   Authorizer {
2.
3. @Override
4. public Boolean authorize(String username, String
   password) {
5.  return username.equals("Bob");
6. }
7. }
```

Construcción de los colaboradores

En la construcción de los «Test Doubles» podemos ver diferentes **estrategias para la creación** de los mismos, entre ellas podemos observar con bastante frecuencia.

- **Mothers:** Es un conjunto de «Factory Methods» que nos permiten crear diferentes objetos para nuestros tests.
- **Builders:** Es el uso del patrón de diseño «builder*» para configurar el diferentes objetos a nuestro antojo que sean lo que necesitamos para nuestro test.

Definición original del patrón: El patrón de diseño «Builder» es un patrón de diseño de software de tipo creacional. La intención del patrón «Builder» es encontrar una solución al problema del anti-patrón de diseño del constructor telescópico.

No quiere decir que estos sean los únicos patrones de diseño o estrategias para la construcción de colaboradores que podemos usar. Al fin y al cabo estás programando como hemos dicho y tienes total libertad para diseñar los elementos que quieras y como quieras.

Pues muy bien, hasta aquí llegamos en este nivel. Hemos visto **la forma que tiene un test** y los diferentes **elementos que interactúan** en un test para ayudarnos a aislar nuestro código testado o SUT y ofrecernos información acerca de la ejecución del test. **Es importante profundizar en estos conceptos y leer bastantes test** en proyectos que podamos encontrar en la

comunidad. De esta forma puedes asentar esta base teórica y darle forma cuando empieces a escribir las primeras líneas de test. ¿Te atreves a implementar tus primeros test ya?.

Si no te atreves, en la siguiente entrega, vamos a poner **algún ejemplo interesante**. Además vamos a ver cuáles son las **partes más interesantes de nuestro código para testar**, veremos también los **errores comunes** y algún concepto interesante como es el **TDD**, que no es una herramienta de diseño de test, como algunos piensan.

- Web oficial de «Harmcrest»: http://hamcrest.org/
- Un ejemplo de pipeline de desarrollo usado en Karumi explicada en un serie de cuatro posts: parte uno, parte dos, parte tres y parte cuatro.

Testing nivel 3: ¡No se cómo hacer mi primer test!

¡Bueno!, ya tienes un bagaje bastante amplio del mundo del testing. Pero el mundo real del desarrollador es duro, por lo tanto en este último nivel **vamos a exponer conceptos clave para iniciarse en el mundo de testing con el menor margen de fracaso deseable**. ¡No pierdas detalle!

¿Qué tengo que testar?

¿Qué es lo más importante?

En orden de importancia, vamos a enumerar lo aquello en lo que debemos centrar nuestra atención para empezar con nuestros primeros test:

Core de la lógica de negocio en general, en esto se incluyen las decisiones lógicas, como los if, switch/case, que van a derivar en un camino u otro dependiendo del estado de nuestro sistema. También englobamos aquí las **operaciones matemáticas, o la algoritmia** en general que tiene lugar en nuestro sistema. Las operaciones sobre conjuntos y colecciones también son candidato fiel a ser testadas en primer lugar.

Este gran grupo **a priori puede ser difícil de abordar**, pero si respetamos lo que hemos ido aprendiendo y los errores comunes que más tarde veremos, nos daremos cuenta de que los métodos

que componen nuestro core de negocio son bastante escuetos y pueden ser abordados de una manera bastante asequible.

Haciendo este tipo de tests debemos **prestar atención a los valores límite o no comunes**, que con frecuencia pueden hacer que nuestro código se comporte de manera inestable. Ejemplos de esto son: números negativos, números excesivamente altos, null, cadenas vacías… **Es una buena práctica intentar estresar nuestros test**, hacer que los parámetros de entrada puedan ser una combinación de variables que la computadora ejecute por nosotros. Un ejemplo de esto son los parametrized de Junit. ¡Seguro que en tu lenguaje favorito también existe algo parecido!

En segundo lugar tenemos la **construcción de objetos.** Esto puede ser un punto de fallo bastante habitual, ya que es muy probable construir objetos y estructuras de datos que no son del todo consistentes o no interactúan bien, o de la forma esperada, entre los elementos que las componen. Es habitual testar patrones de diseño creacionales como los `Builder`, `Abstract Factory`, etc. Verificar el comportamiento cuando varios objetos interactúan entre sí como vimos en el post anterior con test sociables es parte importante para comprobar el correcto funcionamiento de nuestro sistema.

Y por último, el **trasiego de información entre capas**, o entre diferentes ámbitos de nuestro sistema. Es decir aquello que requiere de *mappers*, *adapters* o elementos de este tipo que transforman un tipo de objeto en otro, son también un punto importante a tener en cuenta a la hora de hacer nuestros test. Un ejemplo de SUT para estos test también puede ser el *parser* de

turno que implementemos para recoger los datos de red que esperamos.

¡Recuerda que hemos respetado un estricto orden de importancia!, no deberías hacer test de tus *mappers* sin haber testado nada de la lógica de negocio.

¿Y qué es lo que no es importante?

No tiene sentido testar un SDK de terceros, porque no es parte de nuestro código, y porque tendrá sus propios test, y aún así, si no los tuviera, tampoco podemos hacer mucho porque **es probable que sea una caja negra y ni siquiera tengamos acceso al código**. Además las librerías que usamos son susceptibles de ser cambiadas o actualizadas con el tiempo, por lo que nuestros test probablemente tendrían fecha de caducidad.

Algo parecido pasa con tests que se hacen al framework o al lenguaje usados. Por ejemplo, **no tiene sentido comprobar que java hace bien un remove en una colección**. Puede parecer muy claro que es una estupidez hacer este tipo de test, pero más de una vez **podemos acabar testando el framework o el lenguaje sin querer**, por lo que no viene mal mirarse de vez en cuando si caemos en este típico fallo.

Los *getters* o *setters* en general, o código que no hace nada más que devolver o asignar valores. Los POJO's en Java por ejemplo y clases de este tipo… **No tiene sentido probar algo que no controla nada ni tiene comportamiento**.

¿Cómo tengo que testar?

Concepto de `Assert` y su aplicación

Diferencia entre assert lógico y assert físico: un assert físico es una llamada al método `assert` de tu jUnit o el framework de testing usado. Sin embargo, más importante es el concepto de **assert lógico: este es el mero hecho de comprobar que el estado de nuestro sistema es el que queremos que sea y para ello se pueden estar utilizando varios assert físicos**. Este es el concepto clave para comprobar que nuestro test se adapta a lo que buscábamos.

Personalmente soy partidario de usar un assert lógico por Test, aunque se puede leer mucha literatura sobre este tema, yo pienso que lo ideal es que nuestros test contengan sólo un assert lógico y que el nombre de nuestro test vaya acorde al assert y el escenario como mencionamos en el post anterior. **Con esto ganaremos mucho en legibilidad en nuestros test y documentaremos así nuestro código de una manera mejor y más limpia**.

Hemos estado hablando de assert como concepto lógico, pero como podrás intuir, el asunto del «Verify» o **la verificación de comportamiento de nuestro sistema se puede tratar de una manera idéntica**. Sin embargo ten en cuenta que no es conveniente mezclar comprobaciones de estado y de comportamiento dentro de un mismo test.

¿Cómo puedo empezar con mis test?

Bien, esta es una de las preguntas claves que te habrás hecho a lo largo de tu vida como desarrollador y quizás incluso mientras leías este post. Más si cabe si has intentado hacer tests al software que has desarrollado desde tus inicios como programador. ¡Pues vamos a ver cómo!

Al principio cuando **entramos en el mundo del testing** e intentamos escribir los test de algo que ya está implementado por nosotros mismos, por nuestro compañeros o por quién sabe qué desarrollador, es muy posible que nos venga a la cabeza esta típica frase:

«No tengo in idea de cómo empezar a meter mano para hacer un test de esta clase»

Pero principalmente el fallo no está en que no tengamos ni idea de como hacer un test. De hecho es bastante frustrante leer un montón de documentación, ejemplos de testing y ponerse manos a la obra para no tener idea de cómo hacer la primera línea. Aunque para aliviar esa sensación de frustración aquí tengo la clave: muy probablemente **la clase está mal diseñada de cara al testing**, es por eso que no vemos claro cómo empezar.

Entonces estarás de acuerdo conmigo en que es más que interesante ver los **principales errores de programación que crean impedimentos a la hora de hacer test unitarios**, ¿verdad?, vamos allá:

- Estáticos: **no podemos aislar una porción de código que hace uso de un estático**. El test de una funcionalidad que usa un estático está condenado a ser sociable. Piensa y razona esta afirmación en base a lo que sabes y el significado del modificador static. Un problema frecuente es que la inicialización y declaración de variables estáticas o de clase sólo plantean dificultades a la hora de aislar nuestro código testado.
- Los métodos que **hacen más de una cosa**: la creación de colaboradores, preparación de escenarios o la cantidad de verificaciones que tenemos que hacer cuando los métodos no respetan el hacer una y solo una cosa, crecen de manera exponencial.
- Los métodos que tienen **complejidades ciclomáticas elevadas:** también hacen que crezcan de manera exponencial los escenarios, colaboradores a preparar y la cantidad de resultados de estado y comportamiento en los cuales desembocan normalmente. **Un nivel de bucle anidado mayor a dos empieza a ser demasiado para que el SUT sea testable**. Intenta evitar para no generar ruido indeseable e incomodidades de cara al testing:
- Los **métodos privados.**
- Los **métodos y clases finales**.
- El uso de «new» o **la creación de dependencias** en forma de nuevos objetos.

Que incluyamos los métodos privados en la lista anterior, no quiere decir que tengamos que exponer todos nuestros métodos para poder testarlos, de hecho mientras menos métodos se expongan mejor.

Sin embargo puede ser un fallo de diseño si tenemos una clase que hace demasiadas cosas y tiene métodos públicos y privados en abundancia.

En lugar de eso podrás usar varias clases que tienen una labor concreta e interactúan entre sí usando métodos públicos que son su interfaz, estas clases tras la división también tendrán esos métodos privados para lo que realmente no deben dar a conocer, pero en este caso el número de métodos privados no será tan elevado y será mucho más fácil hacer uso de los métodos públicos en nuestro test para dar cobertura al 100% de la clase.

Al fin y al cabo esto no es más que hacer caso al principio de responsabilidad única.

También podemos ver en este listado el uso de la palabra «new», o al fin y al cabo la creación de una nueva instancia dentro de nuestro código que será testado. En otras palabras y haciendo referencia al primer post, estamos creando una dependencia bastante fuerte. Para evitar esto, como bien sabemos hemos de usar la inversión de dependencias.

«Evitando todos estos fallos comunes conseguiremos una cantidad considerable de código testable»

De lo contrario, nuestro código no va a poder darnos facilidad para hacer nuestros test y puede incluso que la cantidad de código testable, sea inferior de lo deseable. Por ejemplo, no

podemos llegar a una cobertura de código de 100% porque el 100% de nuestro código no es testable desde un punto de vista unitario.

Tener consciencia de las limitaciones

Ten en cuenta que no podemos volvernos expertos de testing y querer abarcar todo nuestro software en un día ni en un mes. Es bueno tener consciencia de ciertas limitaciones…

Sobre el mito de la cobertura 100%: es un mito a la vez que una locura pretender tener testado el 100% de nuestro código, sería una pérdida de tiempo porque como hemos visto hay muchas partes que no merece la pena testar. Además **hay partes de código que ya tendrán su cobertura con otro tipo de test** que no sean unitarios, ex: UI, integración…

Sobre el legacy code: Es posible que incluir test en un proyecto legacy requieran un refactor anterior del código, pero las nuevas piezas no lo requieren y son candidatas a programarse con consciencia de ser SUT. Con esto, de nuevo desde mi opinión, no merece la pena intentar hacer test sobre un código no testable, nos puede traer muchos quebraderos de cabeza. Habría que entrar en una fase de refactor para acometer dichos cambios. Sin embargo los desarrollos que vengan como extensión del sistema sí son fieles candidatos a tener test.

Mantenimiento de los test

«Es importante que tengas en cuenta que el mantenimiento de los test es tu responsabilidad y no debes huir de ella»

El mantenimiento es una parte muy, muy importante de los test. Esto es debido a que **los test son código también y viven por y para el código de producción. Los test cambian con el código** y esto quiere decir que:

Si y sólo **si una funcionalidad o comportamiento del sistema cambia, los test deben cambiar y ser mantenidos con ella**.

Esto es importante porque **los test no han de cambiar bajo un refactor, hacer que los test cambien tras un refactor desencadenarían en esos test falsos positivos** que dejan pasar una funcionalidad que está mal refactorizada. Presta atención a este punto porque es un error bastante recurrente, es una tentación presente en nuestro día a día que debemos evitar.

Como puedes ver en cierto modo, «**nuestros tests también respetan el principio OPEN / CLOSE**», respetan el mismo ciclo de vida que nuestro software.

TDD: Test Driven Development

Se ha hablado mucho de este concepto y hay un libro muy bueno sobre «Test driven Development» de Kent Beck. Aquí vamos a hacer sólo una pequeña introducción.

Es una herramienta que se usa para el diseño y el desarrollo de software, y no es una herramienta para Testing como tal. Como su nombre indica, en esta herramienta son los test los que guían al desarrollo, por lo tanto es es requisito indispensable escribir los test antes de desarrollar la funcionalidad.

Por lo tanto si no vamos a escribir los test antes que el código real no estaremos usando TDD. Ojo: no hay problema ninguno en no usar TDD, **se pueden conseguir magníficos resultados sin su uso**. Sin embargo sí nos conviene pensar en cierto modo como se piensa en TDD. En otras palabras, **tenemos que pensar en desarrollar un código que será testado**, por lo tanto tiene que ser testable, esto no es más que evitar cometer los fallos que se han descrito anteriormente.

Al principio escribiremos código que creemos que es testable que vamos a tener que refactorizar. Esto es normal, de hecho es el proceso habitual que vamos a seguir en nuestro aprendizaje y es bueno. Así iremos dándonos cuenta de los fallos de la mejor manera: con la experiencia que nos van aportando los sucesivos casos de test que desarrollamos.

Lo que podemos llegar a conseguir

Como no podía ser menos, el incluir test nos aporta muchas ventajas, te dejo algunas de ellas:

Código funcionalmente correcto, o casi… esto es algo que se consigue con la práctica. Es posible que nuestros test no validen bien y de forma completa nuestros requisitos, sobre todo cuando

estemos empezando, como te comentaba. Pero aún así, es mejor que no tenerlos porque entrenaremos nuestra habilidad para implementar casos de test. Este entrenamiento nos traerá de la mano que poco a poco vayamos detectando fallos, al principio de programación, como referencias nulas o arrays fuera de rango quizás, pero poco a poco iremos descubriendo más y más errores de funcionamiento. Esta detección nos ayudará a mejorar el entregable definitivo de nuestro sistema y a reducir su número de «bugs». Al fin y al cabo estaremos validando de mejor manera lo que tiene que hacer nuestro código.

Como hemos podido ver, **el centrarnos en un desarrollo que admita tests de una manera sencilla guía nuestro código por el camino Clean y SOLID**, como ya bien sabes. No hace falta volver a nombrar las ventajas de esto ¿verdad?. El ciclo, como puedes ver se retroalimenta.

Los test también nos van a ayudar a perder el miedo al «refactoring». Esto es una gozada, sin duda cambiar código que está testado es mucho más fácil, porque podemos pasar nuestros test al finalizar el «refactor» y comprobar que todo ha quedado en el mismo estado en el que estaba antes de «meter mano» a ese código.

Dormir más tranquilos porque la **integración continua** pasa los test cada noche por nosotros. ¡Sí señor!, hay una herramientas como Jenkins o Travis, entre muchas, que pueden automatizar el paso de nuestros test, para una hora concreta, para cada vez que se sube al repositorio, e infinidad de cosas más. Con lo cuál no tenemos que vivir preocupados por haber pasado o no los test cada vez que hacemos un cambio por mínimo que sea, cosa que

es recomendable al 100%, porque sabemos que en algún momento el CI lo pasará y nos puede avisar de que algo ha fallado.

Hay muchas más ventajas que irás descubriendo por tu cuenta, así que ¡¡dale caña a tus tests!!.

Ejemplo

Como lo prometido es deuda aquí dejamos un ejemplo del test. Iré comentando por cada trozo de código. Primero podemos encontrar nuestro SUT:

Lo que ha de hacer el software es comprobar en base a todos los datos del usuario que no sean el email si se trata de un usuario duplicado o no. Esto es así porque el sistema no permite usuarios duplicados pero si permite usuarios que usen el mismo mail. Es un poco tonto el ejemplo pero nos servirá.

```
1. public class UserDuplicityChecker {
2.
3.  ApiService apiService;
4.
5.  UserDuplicityChecker(ApiService apiService){
6.    this.apiService = apiService;
7.  }
8.
9.  public boolean areSameUsers(User newUser){
10.
11.     User oldUser =
   apiService.getUserWithEmail(newUser.getEmail());
```

```
12.
13.     return
  oldUser.getDocumentId().equals(newUser.getDocumentId())
14.       && oldUser.getName().equals(newUser.getName());
15.
16.    }
17.   }
```

Puedes ver que es una clase muy sencilla con sólo método público, pero es suficiente para un test, hace una comprobación de dos campos con un if, nada complicado.

```
1. public interface ApiService {
2.  User getUserWithEmail(String s);
3. }
```

Hemos creado esta interfaz para no acoplarnos a una dependencia concreta, como puedes ver, nuestro SUT hace uso de ella. ¡Estamos usando inversión de dependencias!

```
1. public class MyFakeApiService implements ApiService {
2.  @Override public User getUserWithEmail(String email) {
3.    return new UserTestBuilder()
4.       .name("Jose")
5.       .documentId("myTestId")
6.       .email(email)
7.       .build();
8.  }
9. }
```

Si pensabas que esto de arriba tiene forma de «Fake» – «Test Double» ¡es exactamente eso!, además como puedes ver hace uso de un Builder que hemos creado para nuestros tests y puedes verlo aquí abajo.

```
1. public class UserTestBuilder {
2.
3.    private String name;
4.    private String documentId;
5.    private String email;
6.
7.    public UserTestBuilder name(String name) {
8.      this.name = name;
9.      return this;
10.      }
11.
12.      public UserTestBuilder documentId(String
   documentId) {
13.        this.documentId = documentId;
14.        return this;
15.      }
16.
17.      public UserTestBuilder email(String email) {
18.        this.email = email;
19.        return this;
20.      }
21.
22.      public User build() {
23.        return new User(name, documentId, email);
24.      }
25.
26.   }
```

La clase «User» es una clase cualquiera de nuestro dominio.

```
1. public class User {
2.
3.  private final String name;
4.  private final String email;
5.  private final String documentId;
6.
7.  public User(String name, String documentId, String
   email) {
8.    this.name = name;
9.    this.email = email;
10.      this.documentId = documentId;
11.    }
12.
13.    public String getName() {
14.      return name;
15.    }
16.
17.    public String getDocumentId() {
18.      return documentId;
19.    }
20.
21.    public String getEmail(){
22.      return email;
23.    }
24.
25.   }
```

Y por último en el test puedes ver cómo todos los elementos que hemos ido implementando van interactuando.

```
1. @Test
```

```
2. public void
   shouldBeSameUsersWhenCheckingUserTakesFakeSameValues(){
3.
4.  ApiService apiService = new MyFakeApiService();
5.  User user = new User("Jose", "myTestId",
   "testEmail@test.com");
6.
7.  UserDuplicityChecker userChecker = new
   UserDuplicityChecker(apiService);
8.  boolean sameUsers = userChecker.areSameUsers(user);
9.
10.    assertTrue(sameUsers);
11.   }
```

En la primera línea puedes ver cómo instanciamos el Fake que implementa la interfaz que nuestro SUT espera. A continuación, preparamos nuestros datos de entrada para el test. Como puedes suponer las dos siguientes líneas son el «Arrange». Luego tenemos dos líneas en las cuales instanciamos nuestro SUT y ejecutamos el método que entra en el «scope» de este test, esto es el «Act». Por último en la última línea hacemos el «Assert», es decir, comprobamos que el estado es lo que esperábamos. Como último detalle fíjate en el método y en lo que hace todo el test. Tiene sentido el nombre que hemos puesto ¿verdad?.

Aún así, la forma de implementar el test es mejorable porque podríamos haber usado un «Factory» para pasarle el valor al «Fake» de ApiService y así poder crear varios escenarios en nuestros siguientes métodos de «@Test».

En este caso hemos creado el Test double a mano, pero hay herramientas como «Mockito» que ayudan en estas labores, tienes la referencia al final del artículo.

Conclusión

Pues tengo poco más que decir, sobre este tema, creo que ya he dicho bastante, ahora es tu turno, te toca actuar y recuerda esta frase, que aplica no solo al mundo del testing sino a muchos más: **«La práctica hace la perfección», así que una vez empieces con el mundo del testing, no lo dejes de lado**, mantente practicando test.

No quería acabar sin antes dar las gracias a la comunidad Android en general que me ha ayudado bastante a introducirme en el mundo de testing y entre ellos en particular a Pablo Guardiola, ex-compi de curro y un grande en esto de hacer tests y muchas más cosas. A José Benito, que curra conmigo en Gigigo y está haciendo una gran labor para enseñarnos a todos mucho sobre test unitarios. Por supuesto también gracias a Antonio Leiva por dejarme poner mi granito de arena en este blog.

¡Espero que te haya gustado!

Siempre puedes enviarme tu feedback a contacto@devexperto.com y estaré encantado de leerlo.

De nuevo, dar las gracias a Sergio Martínez por este contenido.

Recuerda que si estás buscando amplicar tus conocimientos, tengo las siguientes opciones:

- El libro **Kotlin for Android Developers**,
- El curso online
- La formación para empresas.
- El programa Architect Coders (¡incluye un training gratuito de 2 semanas!)

Muchas gracias, y seguimos en contacto por:

- Instagram
- YouTube
- Canal de Telegram
- Twitter
- Facebook
- Podcast

Y si has llegado a esta página por otra vía, puedes suscribirte aquí y llevarte de regalo la guía para iniciarte en Kotlin.

¡Un abrazo!
Antonio Leiva