Testing









Contenido

[¿Qué es un test y que herramientas vamos a utilizar? 3](#_Toc64631101)

[Etiquetas 3](#_Toc64631102)

[Asserts 4](#_Toc64631103)

[Trucos y consejos. 4](#_Toc64631104)

[Hoja de Referencia Espresso: 5](#_Toc64631105)

[Ejemplos Espresso: 6](#_Toc64631106)

[Jacoco (Cobertura de código) 7](#_Toc64631107)

# ¿Qué es un test y que herramientas vamos a utilizar?

Es una comprobación de nuestro código para verificar que ante una entrada de datos devuelve un resultado esperado.

Lo que nos va a permitir hacer pruebas de forma rápida y asegurar que si algún test ha fallado es porque el entorno a cambiado, ya sea por nueva funcionalidad (nos obliga a revisar la programación de los test) o algún actor independiente a la app se comporta de forma diferente.

Los frameworks de JUnit, Espresso y Mockito nos ayudaran en esta aventura:

Junit, es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java. <https://junit.org/junit5/> <https://es.wikipedia.org/wiki/JUnit>

Espresso, para escribir pruebas de IU de Android concisas, hermosas y confiables. Lo desarrolla Google y nos asegura que la activity se inicia antes de empezar el test, simulamos el uso del dispositivo por un usuario. <https://developer.android.com/training/testing/espresso>

Patrón:

* onView(ViewMatcher) -> Buscamos la vista.
* Perform(ViewAction) -> Ejecutamos una acción: click, escribir texto, ….
* Check(ViewAssertion) -> Comprobamos que se cumple lo que queremos.

Mockito, genial marco de simulación para pruebas unitarias en Java <https://site.mockito.org/>

# Etiquetas

@Test, indica que es un método de prueba.

@Test(expected = ArithmeticException::class), nos permite validar excepciones como se explica más adelante.

@Test(timeout = 1000), da un tiempo de espera máximo para que se valide el test. En este ejemplo si ejecutar el test lleva más de un segundo, el test dará fallo.

@Before, es un método que se va a ejecutar antes de cada método de prueba, es el sitio ideal para inicializar datos o instanciar objetos. Se le suele llamar “setUp()”.

@BeforeClass, el método debe ser estático por lo demás es igual que @before, pero se ejecuta una única vez antes de lanzar todos los métodos de la clase.

@After, es un método que se va a ejecutar después de cada método de prueba, es el sitio ideal para liberar recursos de memoria. Se le suele llamar “tearDown()”.

@AfterClass, el método debe ser estático por lo demás es igual que @after, pero se ejecuta una única vez después de lanzar todos los métodos de la clase.

@Ignore, nos sirva para ignorar un método a la hora de pasarlos todos. En el detalle de ejecución nos dirá los exitosos, los fallidos y los ignorados para que los tengamos en cuenta.

@RunWith, invoca a la clase que se le indica para lanzar los test, si no se pone nada se lanza la clase por defecto. Como por ejemplo para pruebas parametrizadas se usa: @RunWith(value = Parameterized::class) o para suite: @RunWith(Suite::class)

@Rule, las reglas de JUnit brindan más flexibilidad y reducen el código estándar requerido en las pruebas. <https://developer.android.com/training/testing/junit-rules> por ejemplo:

@get:Rule

var activityScenarioRule: ActivityScenarioRule<ClassActivity> = ActivityScenarioRule(ClassActivity::class.java)

Se encarga de iniciar la activity y cerrarla creando el flujo correcto para el correcto funcionamiento del código de test.

# Asserts

Los métodos Assert nos valida si tenemos un resultado esperado, y por lo tanto validar el test, como recomendación es interesante tener solo un Assert por método. Recordad que es mejor hacer muchos test de cada cosa pequeñita que comprobar en solo test muchas cosas.

En la mayoría de métodos se puede añadir un mensaje que se verá en caso de que no se valide correctamente. (mesage / expected / actual)

Algunos métodos permiten un parámetro delta, para que el expected acepte un rango de valores numéricos, por encima y por debajo.

Assert.fail(“Fallo manual”) nos permite dejar un test para que falle siempre, por si queremos dejarnos una especie de TODO o recordatorio.

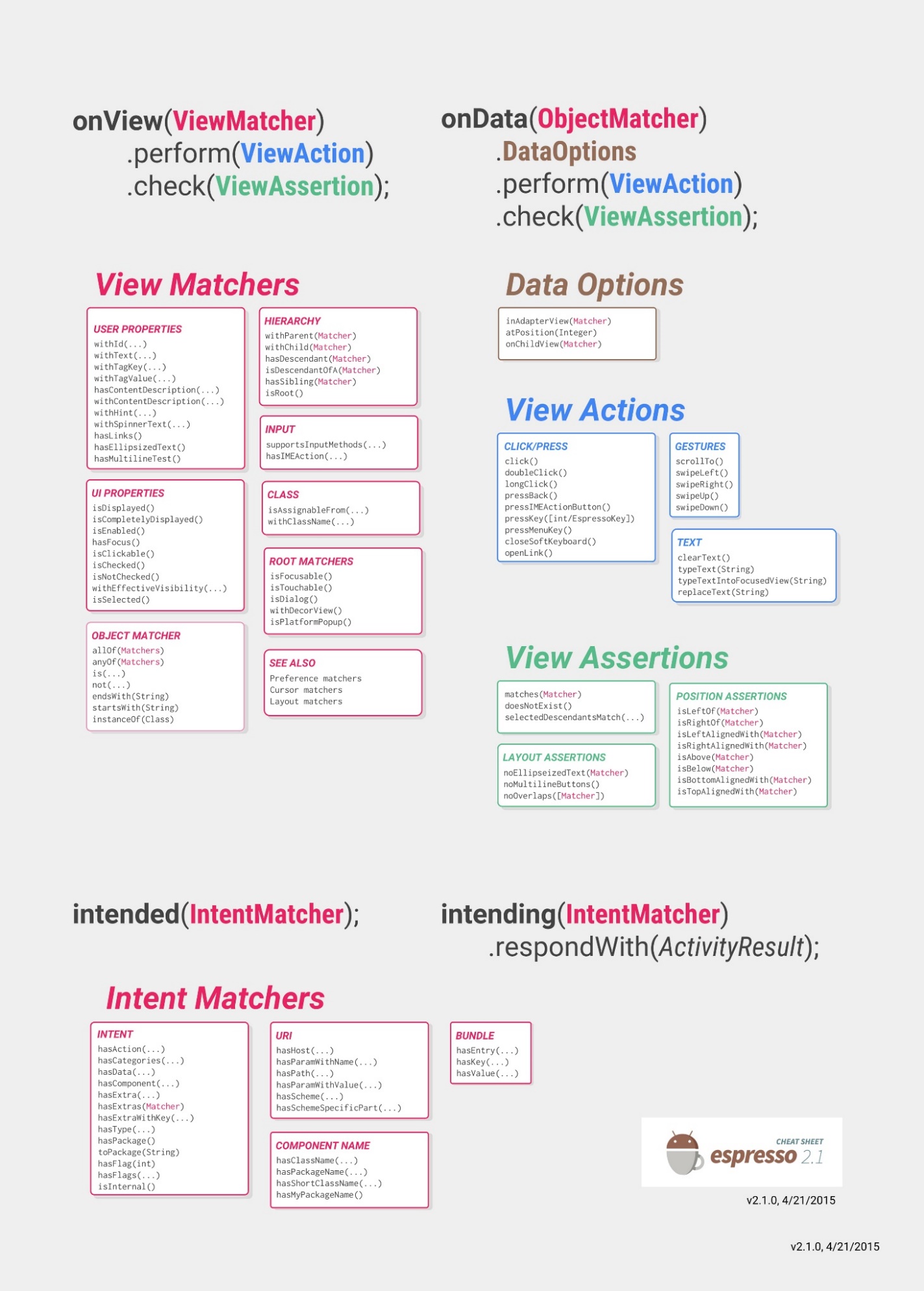
Podemos validar que un test con ciertos parámetros de entrada, ocasione una excepción en la etiqueta test añadimos el tipo de excepción esperado -> @Test(expected = Exception::class), se cambia por la clase de excepción que necesitemos y nos valida el test si el funcionamiento del mismo lanza dicha excepción.

# Trucos y consejos.

* Crear un test de una clase: botón derecho en la clase / go to / test / create new test / revisar nombre / ok / ver si lo queremos situar en los test de UI (probamos interfaz de usuario) o Unit Test (solo código)
* Deshabilitar las animaciones en el dispositivo de pruebas, porque pueden hacer que los test fallen. Opciones de desarrollador -> Dibujo -> Desactivar: Escala Animación de ventana, Escala Transición-animación y Escala duración de la animación
* Podemos definir los nombre de los test (UNITARIOS) con comillas invertidas para ponerlos en plan texto: fun `success if observe true when change result`() {

# Hoja de Referencia Espresso:

<https://developer.android.com/training/testing/espresso/cheat-sheet>



# Ejemplos Espresso:

**Búsqueda:**

onView(withId(R.id.rvListUsersBGG))

onView(withText("Lista usuarios"))

onView(withText(startsWith("Prueba")))

onView(withContentDescription("imagen de prueba"))

onView(withHint("Password"))

onView(allOf(withText("Prueba Espresso"), withParent(withId(R.id.clContainer))))

onView(allOf(withId(R.id.rvListUsersBGG), isDisplayed()))

onView(allOf(instanceOf(TextView::class.java), withId(R.id.tvDetail)))

**Acciones:**

onView(withId(R.id.etPassword)).perform(typeText("Todo correcto"))

onView(withId(R.id.etPassword)).perform(typeText("123456"), ViewActions.closeSoftKeyboard())

onView(withId(R.id.etPassword)).perform(replaceText("Vamos"))

onView(withId(R.id.tvResult)).perform(clearText())

onView(withId(R.id.btLogin)).perform(click())

onView(withId(R.id.clConfigListThings)).perform(swipeDown())

onView(withId(R.id.clConfigListThings)).perform(openLinkWithText("www.google.com"))

**Comprobaciones:**

onView(withId(R.id.rvListUsersBGG)).check(matches(isDisplayed()))

onView(withId(R.id.tvResult)).check(matches(withText("Prueba Espresso")))

onView(withId(R.id.btLogin)).check(matches(withText(R.string.accept)))

onView(withId(R.id.rvListUsersBGG)).check(isCompletelyLeftOf(withId(R.id.clContainer)))

onView(withId(R.id.etName)).check(doesNotExist())

# Jacoco (Cobertura de código)

Páginas de consulta para montarlo:

<https://ichi.pro/es/cobertura-de-android-kotlin-code-con-jacoco-sonar-y-el-complemento-gradle-6-x-62895187573358>

<https://medium.com/@ranjeetsinha/jacoco-with-kotlin-dsl-f1f067e42cd0>

Te hace falta la parte de servidor que interpreta los ficheros de Jacoco, para mostrar la cobertura de código, en el repositorio solo hemos llegado a lanzar todos los test unitarios y de interfaz para que nos muestre el informe de los que han fallado.