Documentación TESTNG

Tabla de contenido

1. Introducción

TestNG es un marco de prueba diseñado para simplificar una amplia gama de necesidades de prueba, desde pruebas unitarias (prueba de una clase aislada de las demás) hasta pruebas de integración (prueba de sistemas completos hechos de varias clases, varios paquetes e incluso varios marcos externos, como servidores de aplicaciones).

Escribir una prueba es típicamente un proceso de tres pasos:

- Escriba la lógica comercial de su prueba e inserte anotaciones TestNG en su código.
- Agregue la información sobre su prueba (por ejemplo, el nombre de la clase, los grupos que desea ejecutar, etc.) en un archivo `testng.xml` o en build.xml.
- Ejecute TestNG .

Puede encontrar un ejemplo rápido en la página de bienvenida .

Los conceptos utilizados en esta documentación son los siguientes:

- Una suite está representada por un archivo XML. Puede contener una o más pruebas y está definido por la etiqueta `<suite>` .
- Una prueba está representada por `<test>` y puede contener una o más clases TestNG.
- Una clase TestNG es una clase Java que contiene al menos una anotación TestNG. Está representado por la etiqueta `<class> y puede contener uno o más métodos de prueba.`
- Un método de prueba es un método Java anotado por `@Test` en su código fuente.

Se puede configurar una prueba TestNG mediante anotaciones `@BeforeXXX y @AfterXXX`, lo que permite realizar alguna lógica de Java antes y después de cierto punto, siendo estos puntos cualquiera de los elementos enumerados anteriormente.

El resto de este manual explicará lo siguiente:

- Una lista de todas las anotaciones con una breve explicación. Esto le dará una idea de las diversas funcionalidades que ofrece TestNG, pero probablemente querrá consultar la sección dedicada a cada una de estas anotaciones para conocer los detalles.
- Una descripción del archivo testng.xml, su sintaxis y lo que puede especificar en él.
- Una lista detallada de las diversas funciones y cómo usarlas con una combinación de anotaciones y testng.xml.

2 - Anotaciones

Aquí hay una descripción general rápida de las anotaciones disponibles en TestNG junto con sus atributos.

	Información de configuración para una clase TestNG:
`@BeforeSuite` **`@AfterSuite`** **`@BeforeTest`** **`@AfterTest`** **`@BeforeGroups`** **`@AfterGroups`** **`@BeforeClass`** **`@AfterClass`** **`@BeforeMethod`** **`@AfterMethod`**	**@BeforeSuite:** el método anotado se ejecutará antes de que se ejecuten todas las pruebas en este conjunto. **@AfterSuite:** el método anotado se ejecutará después de que se hayan ejecutado todas las pruebas en este conjunto. **@BeforeTest** : el método anotado se ejecutará antes de que se ejecute cualquier método de prueba que pertenezca a las clases dentro de la etiqueta <test>. **@AfterTest** : el método anotado se ejecutará después de que se hayan ejecutado todos los métodos de prueba que pertenecen a las clases dentro de la etiqueta <test>. **@BeforeGroups** : la lista de grupos que este método de configuración ejecutará antes. Se garantiza que este método se ejecutará poco antes de que

		se invoque el primer método de prueba que pertenece a cualquiera de estos grupos. **@AfterGroups**: La lista de grupos después de los cuales se ejecutará este método de configuración. Se garantiza que este método se ejecutará poco después de que se invoque el último método de prueba que pertenece a cualquiera de estos grupos. **@BeforeClass** : el método anotado se ejecutará antes de que se invoque el primer método de prueba en la clase actual. **@AfterClass** : el método anotado se ejecutará después de que se hayan ejecutado todos los métodos de prueba en la clase actual. **@BeforeMethod** : el método anotado se ejecutará antes de cada método de prueba. **@AfterMethod** : el método anotado se ejecutará después de cada método de prueba. **Comportamiento de anotaciones en superclase de una clase TestNG** Las anotaciones anteriores también se respetarán (heredadas) cuando se coloquen en una superclase de una clase TestNG. Esto es útil, por ejemplo, para centralizar la configuración de la prueba para varias clases de prueba en una superclase común. En ese caso, TestNG garantiza que los métodos "@Before" se ejecuten en orden de herencia (primero la superclase más alta, luego bajando en la cadena de herencia) y los métodos "@After" en orden inverso (ascendiendo en la cadena de herencia).
	`alwaysRun`	Para los métodos anteriores (beforeSuite, beforeTest, beforeTestClass y beforeTestMethod, pero no beforeGroups): si se establece en verdadero, este método de configuración se ejecutará independientemente de a qué grupos pertenezca. Para métodos posteriores (afterSuite, afterClass, ...): si se establece en verdadero, este método de configuración se ejecutará incluso si uno o más métodos invocados anteriormente fallaron o se omitieron.
	`dependsOnGroups`	La lista de grupos de los que depende este método.
	`dependsOnMethods`	La lista de métodos de los que depende este método.
	`groups`	Si los métodos en esta clase/método están habilitados.
	`inheritGroups`	La lista de grupos a los que pertenece esta clase/método.
	`heredargrupos`	Si es verdadero, este método pertenecerá a los grupos especificados en la anotación @Test en el nivel de clase.
	`onlyForGroups`	Solo para @BeforeMethod y @AfterMethod. Si se especifica, este método de configuración/desmontaje solo se invocará si el método de prueba correspondiente pertenece a uno de los grupos enumerados.
`@DataProvider`		**Marca un método como fuente de datos para un método de prueba. El método anotado debe devolver un Object[][] donde a cada Object[] se le puede asignar la lista de parámetros del método de prueba. El método @Test que quiere recibir datos de este proveedor de datos necesita usar un nombre de proveedor de datos igual al nombre de esta anotación.**
	`name`	El nombre de este proveedor de datos. Si no se proporciona, el nombre de este proveedor de datos se establecerá automáticamente en el nombre del método.

parallel	Si se establece en verdadero, las pruebas generadas con este proveedor de datos se ejecutan en paralelo. El valor predeterminado es falso.
@Factory	**Marca un método como una fábrica que devuelve objetos que serán utilizados por TestNG como clases de prueba. El método debe devolver Object[].**
@Listeners	**Define oyentes en una clase de prueba.**
value	Una matriz de clases que amplían `org.testng.ITestNGListener`.
@Parameters	**Describe cómo pasar parámetros a un método @Test.**
valor	La lista de variables utilizadas para llenar los parámetros de este método.
@Test	**Marca una clase o un método como parte de la prueba.**
`alwaysRun`	Si se establece en verdadero, este método de prueba siempre se ejecutará incluso si depende de un método que falló.
`dataProvider`	El nombre del proveedor de datos para este método de prueba.
`dataProviderClass`	La clase donde buscar el proveedor de datos. Si no se especifica, el proveedor de datos se buscará en la clase del método de prueba actual o en una de sus clases base. Si se especifica este atributo, el método del proveedor de datos debe ser estático en la clase especificada.
`dependsOnGroups`	La lista de grupos de los que depende este método.
`dependsOnMethods`	La lista de métodos de los que depende este método.
`description`	La descripción de este método.
`enabled`	Si los métodos en esta clase/método están habilitados.
`expectedExceptions`	La lista de excepciones que se espera que genere un método de prueba. Si no se lanza ninguna excepción o una diferente a la de esta lista, esta prueba se marcará como fallada.
`groups`	La lista de grupos a los que pertenece esta clase/método.
`invocaciónCount`	El número de veces que se debe invocar este método.
`invocationTimeOut`	El número máximo de milisegundos que debe tomar esta prueba para el tiempo acumulado de todos los recuentos de invocaciones. Este atributo se ignorará si no se especifica el número de invocaciones.
`priority`	La prioridad para este método de prueba. Las prioridades más bajas se programarán primero.
`successPercentage`	El porcentaje de éxito esperado de este método
`singleThreaded`	Si se establece en verdadero, se garantiza que todos los métodos en esta clase de prueba se ejecutarán en el mismo subproceso, incluso si las pruebas se ejecutan actualmente con paralelo="métodos". Este atributo solo se puede usar a nivel de clase y se ignorará si se usa a nivel de método. Nota: este atributo solía llamarse `secuencial` (ahora en desuso).
`timeOut`	El número máximo de milisegundos que debe tomar esta prueba.

	`threadPoolSize`	El tamaño del grupo de subprocesos para este método. El método se invocará desde múltiples subprocesos según lo especificado por invocationCount. Nota: este atributo se ignora si no se especifica el número de invocaciones.

3 - prueba.xml

Puede invocar TestNG de varias maneras diferentes:

- Con un archivo `testng.xml`
- con hormiga
- Desde la línea de comando

Esta sección describe el formato de `testng.xml` (abajo encontrará documentación sobre ant y la línea de comando).

La DTD actual para `testng.xml` se puede encontrar en el sitio web principal: testng-1.0.dtd (para su comodidad, es posible que prefiera buscar la versión HTML).

Aquí hay un ejemplo de archivo `testng.xml` :

```
<!DOCTYPE suite SYSTEM
"https://testng.org/testng-
1.0.dtd" >

<suite name="Suite1" verbose="1" >
  <test name="Nopackage" >
    <classes>
       <class name="NoPackageTest" />
    </classes>
  </test>

  <test name="Regression1">
    <classes>
      <class name="test.sample.ParameterSample"/>
      <class name="test.sample.ParameterTest"/>
    </classes>
  </test>
</suite>
```

Puede especificar nombres de paquetes en lugar de nombres de clases:

```
<!DOCTYPE suite SYSTEM
"https://testng.org/testng-
1.0.dtd" >

<suite name="Suite1" verbose="1" >
  <test name="Regression1"    >
    <packages>
```

```
      <package name="test.sample" />
    </packages>
  </test>
 </suite>
```

En este ejemplo, TestNG observará todas las clases en el paquete `test.sample` y retendrá solo las clases que tengan anotaciones TestNG.

También puede especificar grupos y métodos para incluirlos y excluirlos:

```
 <test name="Regression1">
   <groups>
     <run>
       <exclude name="brokenTests"  />
       <include name="checkinTests"  />
     </run>
   </groups>

   <classes>
     <class name="test.IndividualMethodsTest">
       <methods>
         <include name="testMethod" />
       </methods>
     </class>
   </classes>
 </test>
```

También puede definir nuevos grupos dentro `de testng.xml` y especificar detalles adicionales en los atributos, como si ejecutar las pruebas en paralelo, cuántos subprocesos usar, si está ejecutando pruebas JUnit, etc.

De forma predeterminada, TestNG ejecutará sus pruebas en el orden en que se encuentran en el archivo XML. Si desea que las clases y los métodos enumerados en este archivo se ejecuten en un orden impredecible, establezca el atributo `de orden de conservación en falso`

```
 <test name="Regression1" preserve-
 order="false">
   <classes>

     <class name="test.Test1">
       <methods>
         <include name="m1" />
         <include name="m2" />
       </methods>
     </class>

     <class name="test.Test2" />

   </classes>
```

```
</test>
```

Consulte la DTD para obtener una lista completa de las funciones, o siga leyendo.

4 - Ejecutando TestNG

TestNG se puede invocar de diferentes maneras:

- Línea de comando
- en
- Eclipse
- IDEA de IntelliJ

Esta sección solo explica cómo invocar TestNG desde la línea de comandos. Haga clic en uno de los enlaces anteriores si está interesado en una de las otras formas.

Suponiendo que tiene TestNG en su classpath, la forma más sencilla de invocar TestNG es la siguiente:

```
java
org.testng.TestNG
testng1.xml
[testng2.xml
testng3.xml ...]
```

Debe especificar al menos un archivo XML que describa la suite TestNG que está intentando ejecutar. Además, los siguientes modificadores de línea de comandos están disponibles:

Parámetros de línea de comando		
Opción	**Argumento**	**Documentación**
-configfailurepolicy	`saltar` | `Seguir`	Si TestNG debe `continuar` ejecutando las pruebas restantes en la suite u `omitirlas` si falla un método @Before*. El comportamiento predeterminado es `skip` .
-d	un directorio	El directorio donde se generarán los informes (predeterminado en `test-output`).
-dataproviderthreadcount	El número predeterminado de subprocesos que se usarán para los proveedores de datos al ejecutar pruebas en paralelo.	Esto establece la cantidad máxima predeterminada de subprocesos que se usarán para los proveedores de datos cuando se ejecutan pruebas en paralelo. Solo tendrá efecto si se ha seleccionado el modo paralelo (por ejemplo, con la opción -parallel). Esto se puede anular en la definición de la suite.
-excludegroups	Una lista de grupos separados por comas.	La lista de grupos que desea excluir de esta ejecución.
- groups	Una lista de grupos separados por comas.	La lista de grupos que desea ejecutar (por ejemplo , `"windows,linux,regression"`).
- listener	Una lista separada por comas de las clases de Java que se pueden encontrar en su classpath.	Le permite especificar sus propios oyentes de prueba. Las clases necesitan implementar `org.testng.ITestListener`

Parámetros de línea de comando		
Opción	**Argumento**	**Documentación**
-usedefaultlisteners	`cierto` | `falso`	Ya sea para usar los oyentes predeterminados
-methods	Una lista separada por comas de métodos y nombres de clases completamente calificados. Por ejemplo `com.example.Foo.f1, com.example.Bar.f2` .	Le permite especificar métodos individuales para ejecutar.
-methodselectors	Una lista separada por comas de clases Java y prioridades de métodos que definen los selectores de métodos.	Le permite especificar selectores de métodos en la línea de comandos. Por ejemplo: `com.ejemplo.Selector1:3,com.ejemplo.Selector2:2`
- parallel	métodos|pruebas|clases	Si se especifica, establece el mecanismo predeterminado utilizado para determinar cómo utilizar subprocesos paralelos al ejecutar pruebas. Si no se establece, el mecanismo predeterminado es no usar subprocesos paralelos en absoluto. Esto se puede anular en la definición de la suite.
- reporter	La configuración extendida para un detector de informes personalizado.	Similar a la opción `-listener` , excepto que permite la configuración de propiedades de estilo JavaBeans en la instancia del reportero. Ejemplo: `-reporter com.test.MyReporter:methodFilter=*insert*,enableFiltering=true` Puede tener tantas instancias de esta opción, una para cada informador que deba agregarse.
-sourcedir	Una lista de directorios separados por punto y coma.	Los directorios donde se encuentran las fuentes de prueba anotadas de javadoc. Esta opción solo es necesaria si está utilizando anotaciones de tipo javadoc. (por ejemplo, `"src/test"` o `"src/test/org/testng/eclipse-plugin;src/test/org/testng/testng"`) .
- suitename	El nombre predeterminado que se utilizará para un conjunto de pruebas.	Esto especifica el nombre del conjunto para un conjunto de pruebas definido en la línea de comandos. Esta opción se ignora si el archivo suite.xml o el código fuente especifica un nombre de suite diferente. Es posible crear un nombre de suite con espacios si lo rodea con comillas dobles "como este".
- testclass	Una lista separada por comas de las clases que se pueden encontrar en su classpath.	Una lista de archivos de clase separados por comas (por ejemplo , `"org.foo.Test1,org.foo.test2"`).
-Test	Un archivo jar.	Especifica un archivo jar que contiene clases de prueba. Si se encuentra un archivo `testng.xml` en la raíz de ese archivo jar, se utilizará; de lo contrario, todas las clases de prueba que se encuentren en este archivo jar se considerarán clases de prueba.

Parámetros de línea de comando		
Opción	**Argumento**	**Documentación**
- testname	El nombre predeterminado que se utilizará para una prueba.	Esto especifica el nombre de una prueba definida en la línea de comando. Esta opción se ignora si el archivo suite.xml o el código fuente especifica un nombre de prueba diferente. Es posible crear un nombre de prueba con espacios si lo rodea con comillas dobles "como este".
- testnames	Una lista separada por comas de los nombres de las pruebas.	Solo se ejecutarán las pruebas definidas en una etiqueta <test> que coincida con uno de estos nombres.
- testrunfactory	Una clase de Java que se puede encontrar en su classpath.	Le permite especificar sus propios corredores de prueba. La clase necesita implementar `org.testng.ITestRunnerFactory` .
- threadcount	El número predeterminado de subprocesos que se utilizarán al ejecutar pruebas en paralelo.	Esto establece el número máximo predeterminado de subprocesos que se utilizarán para ejecutar pruebas en paralelo. Solo tendrá efecto si se ha seleccionado el modo paralelo (por ejemplo, con la opción -parallel). Esto se puede anular en la definición de la suite.
-xmlpathinjar	La ruta del archivo XML dentro del archivo jar.	Este atributo debe contener la ruta a un archivo XML válido dentro del contenedor de prueba (por ejemplo , `"resources/testng.xml"`). El valor predeterminado es `"testng.xml"` , lo que significa un archivo llamado " `testng.xml` " en la raíz del archivo jar. Esta opción se ignorará a menos que se especifique `–testjar` .

Esta documentación se puede obtener invocando a TestNG sin argumentos.

También puede poner los interruptores de la línea de comando en un archivo de texto, digamos `c:\command.txt` , y decirle a TestNG que use ese archivo para recuperar sus parámetros:

```
C:> more
c:\command.txt
-d test-
output
testng.xml
C:> java
org.testng.TestNG
@c:\command.txt
```

Además, a TestNG se le pueden pasar propiedades en la línea de comandos de la máquina virtual de Java, por ejemplo

```
java -
Dtestng.test.classpath="c:/build;c:/java/classes;"
org.testng.TestNG testng.xml
```

Aquí están las propiedades que TestNG entiende:

Propiedades del sistema		
Propiedad	**Escribe**	**Documentación**
testng.test.classpath	Una serie de directorios separados por punto y coma que contienen sus clases de prueba.	Si se establece esta propiedad, TestNG la usará para buscar sus clases de prueba en lugar de la ruta de clases. Esto es conveniente si está utilizando la etiqueta del `paquete` en su archivo XML y tiene muchas clases en su classpath, la mayoría de las cuales no son clases de prueba.

Ejemplo:

```
java
org.testng.TestNG
-groups
windows,linux -
testclass
org.test.MyTest
```

La tarea ant y testng.xml le permiten iniciar TestNG con más parámetros (métodos para incluir, especificar parámetros, etc.), por lo que debe considerar usar la línea de comando solo cuando intente aprender sobre TestNG y desee ponte en marcha rápidamente.

Importante : los indicadores de la línea de comandos que especifican qué pruebas deben ejecutarse se ignorarán si también especifica un archivo `testng.xml` , con la excepción de `-includedgroups` y `-excludedgroups` , que anularán todas las inclusiones/exclusiones de grupos que se encuentran en `testng.xml` .

5 - Métodos de prueba, clases de prueba y grupos de prueba

5.1 - Métodos de prueba

Los métodos de prueba se anotan con `@Test` . Se ignorarán los métodos anotados con `@Test` que devuelvan un valor, a menos que establezca `allow-return-values` en `verdadero` en su `testng.xml` :

```
<suite allow-
return-
values="true">

or

<test allow-
return-
values="true">
```

5.2 - Grupos de prueba

TestNG le permite realizar agrupaciones sofisticadas de métodos de prueba. No solo puede declarar que los métodos pertenecen a grupos, sino que también puede especificar grupos que contengan otros grupos. Luego se puede invocar a TestNG y pedirle que incluya un cierto conjunto de grupos (o expresiones regulares) mientras excluye otro conjunto. Esto le brinda la máxima flexibilidad en la forma en que divide sus pruebas y no requiere que vuelva a compilar nada si desea ejecutar dos conjuntos diferentes de pruebas consecutivas.

Los grupos se especifican en su archivo `testng.xml` y se pueden encontrar bajo la etiqueta `<test>` o `<suite>` . Los grupos especificados en la etiqueta `<suite>` se aplican a todas las

etiquetas `<test>` debajo. Tenga en cuenta que los grupos son acumulativos en estas etiquetas: si especifica el grupo "a" en `<suite>` y "b" en `<test>` , se incluirán tanto "a" como "b".

Por ejemplo, es bastante común tener al menos dos categorías de pruebas

- Pruebas de entrada. Estas pruebas deben ejecutarse antes de enviar un código nuevo. Por lo general, deben ser rápidos y solo asegurarse de que no se rompa ninguna funcionalidad básica.
- Pruebas funcionales. Estas pruebas deben cubrir todas las funcionalidades de su software y ejecutarse al menos una vez al día, aunque lo ideal sería ejecutarlas continuamente.

Por lo general, las pruebas de verificación son un subconjunto de las pruebas funcionales. TestNG le permite especificar esto de una manera muy intuitiva con grupos de prueba. Por ejemplo, podría estructurar su prueba diciendo que toda su clase de prueba pertenece al grupo "functest" y, además, que un par de métodos pertenecen al grupo "checkintest":

```
public class Test1
{
  @Test(groups =
{ "functest", "checkintest" })
  public void testMethod1()
{
  }

  @Test(groups =
{"functest", "checkintest"}
)
  public void testMethod2()
{
  }

  @Test(groups
=
{ "functest" })
  public void testMethod3()
{
  }
}
```

Invocando TestNG con

```
<test name="Test1">
  <groups>
    <run>
      <include name="functest"/>
    </run>
  </groups>
  <classes>
    <class name="example1.Test1"/>
  </classes>
</test>
```

ejecutará todos los métodos de prueba en esas clases, mientras que invocarlo con `checkintest` solo ejecutará `testMethod1()` y `testMethod2()` .

Aquí hay otro ejemplo, usando expresiones regulares esta vez. Suponga que algunos de sus métodos de prueba no deben ejecutarse en Linux, su prueba se vería así:

```
@Test
public class Test1
{
  @Test(groups =
{ "windows.checkintest" })
  public void testWindowsOnly()
{
  }

  @Test(groups =
{"linux.checkintest"}
)
  public void testLinuxOnly()
{
  }

  @Test(groups =
{ "windows.functest" )
  public void testWindowsToo()
{
  }
}
```

Puede usar el siguiente testng.xml para iniciar solo los métodos de Windows:

```
<test name="Test1">
  <groups>
    <run>
      <include name="windows.*"/>
    </run>
  </groups>

  <classes>
    <class name="example1.Test1"/>
  </classes>
</test>
```

Nota: TestNG usa expresiones regulares y no comodines . Tenga en cuenta la diferencia (por ejemplo, "cualquier cosa" coincide con "." (punto estrella) y no con "*").*

Grupos de métodos

También puede excluir o incluir métodos individuales:

```
<test name="Test1">
  <classes>
    <class name="example1.Test1">
```

```
      <methods>
        <include name=".*enabledTestMethod.*"/>
        <exclude name=".*brokenTestMethod.*"/>
      </methods>
     </class>
  </classes>
 </test>
```

Esto puede ser útil para desactivar un solo método sin tener que volver a compilar nada, pero no recomiendo usar esta técnica demasiado, ya que hace que su marco de prueba se rompa si comienza a refactorizar su código Java (las expresiones regulares utilizadas en el es posible que las etiquetas ya no coincidan con sus métodos).

5.3 - Grupos de grupos

Los grupos también pueden incluir otros grupos. Estos grupos se denominan "Metagrupos". Por ejemplo, es posible que desee definir un grupo "todos" que incluya "checkintest" y "functest". "functest" en sí mismo contendrá los grupos "windows" y "linux", mientras que "checkintest solo contendrá "windows". Así es como definiría esto en su archivo de propiedades:

```
 <test name="Regression1">
   <groups>
     <define name="functest">
       <include name="windows"/>
       <include name="linux"/>
     </define>

     <define name="all">
       <include name="functest"/>
       <include name="checkintest"/>
     </define>

     <run>
       <include name="all"/>
     </run>
   </groups>

   <classes>
     <class name="test.sample.Test1"/>
   </classes>
 </test>
```

5.4 - Grupos de exclusión

TestNG le permite incluir grupos y excluirlos.

Por ejemplo, es bastante común tener pruebas que fallan temporalmente debido a un cambio reciente y aún no tiene tiempo para arreglar la falla. 4 Sin embargo, desea tener ejecuciones limpias de sus pruebas funcionales, por lo que debe desactivar estas pruebas, pero tenga en cuenta que deberán reactivarse.

Una forma simple de resolver este problema es crear un grupo llamado "roto" y hacer que estos métodos de prueba pertenezcan a él. Por ejemplo, en el ejemplo anterior, sé que testMethod2() ahora está roto, así que quiero desactivarlo:

```
@Test(groups =
{"checkintest", "broken"}
)
public void testMethod2()
{
}
```

Todo lo que necesito hacer ahora es excluir este grupo de la ejecución:

```
<test name="Simple
example">
  <groups>
    <run>
      <include name="checkintest"/>
      <exclude name="broken"/>
    </run>
  </groups>

  <classes>
    <class name="example1.Test1"/>
  </classes>
</test>
```

De esta manera, obtendré una ejecución de prueba limpia mientras mantengo un registro de qué pruebas están rotas y deben corregirse más adelante.

Nota: también puede deshabilitar las pruebas de forma individual utilizando la propiedad "habilitada" disponible en las anotaciones @Test y @Before/After.

5.5 - Grupos parciales

Puede definir grupos a nivel de clase y luego agregar grupos a nivel de método:

```
@Test(groups
=
{ "checkin-
test" })
public class All
{

  @Test(groups
= { "func-
test" )
  public void method1()
{ ... }
```

```
  public void method2()
{ ... }
}
```
En esta clase, method2() forma parte del grupo "checkin-test", que se define a nivel de clase, mientras que method1() pertenece tanto a "checkin-test" como a "func-test".

5.6 - Parámetros

TestNG le permite pasar un número arbitrario de parámetros a cada una de sus pruebas usando la anotación `@Parameters` .

Hay tres formas de configurar estos parámetros.

1. El archivo `testng.xml`
2. Programáticamente
3. Propiedades del sistema Java

5.6.1 - Parámetros de `testng.xml`

Si está utilizando valores simples para sus parámetros, puede especificarlos en su `testng.xml` :
```
@Parameters({ "first-
name" })
@Test
public void testSingleString(String
firstName) {
  System.out.println("Invoked
testString " + firstName);
  assert "Cedric".equals(firstName);
}
```
En este código, especificamos que el parámetro `firstName` de su método Java debe recibir el valor del parámetro XML llamado `first-name` . Este parámetro XML se define en `testng.xml` :
```
<suite name="My
suite">
  <parameter name="first-
name"  value="Cedric"/>
  <test name="Simple
example">
  <-
-
...
-->
```

Se puede usar la misma técnica para las anotaciones `@Before/After` y `@Factory` :

```
@Parameters({ "datasource", "jdbcDriver" })
@BeforeMethod
public void beforeTest(String
ds, String driver) {
```

```
  m_dataSource =
...;                                  //
look up the value of datasource
  m_jdbcDriver
= driver;
}
```

Esta vez, los dos parámetros de Java *ds* y *driver* recibirán el valor dado a las propiedades `datasource` y `jdbc-driver` respectivamente.

Los parámetros se pueden declarar opcionales con la anotación `Optional` :

```
@Parameters("db")
@Test
public void testNonExistentParameter(@Optional("mysql")
String db) { ... }
```

Si no se encuentra ningún parámetro llamado "db" en su archivo `testng.xml` , su método de prueba recibirá el valor predeterminado especificado dentro de la anotación `@Optional` : "mysql".

La anotación `@Parameters` se puede colocar en las siguientes ubicaciones:

- En cualquier método que ya tenga una `anotación @Test` , `@Before/After` o `@Factory` .
- En como máximo un constructor de su clase de prueba. En este caso, TestNG invocará este constructor en particular con los parámetros inicializados a los valores especificados en `testng.xml` siempre que necesite instanciar su clase de prueba. Esta característica se puede utilizar para inicializar campos dentro de sus clases a valores que luego serán utilizados por sus métodos de prueba.

Notas:

- *Los parámetros XML se asignan a los parámetros de Java en el mismo orden en que se encuentran en la anotación, y TestNG generará un error si los números no coinciden.*
- *Los parámetros tienen alcance. En `testng.xml` , puede declararlos bajo una etiqueta `<suite>` o bajo `<test>` . Si dos parámetros tienen el mismo nombre, es el definido en `<test>` el que tiene prioridad. Esto es conveniente si necesita especificar un parámetro aplicable a todas sus pruebas y anular su valor solo para ciertas pruebas.*

5.6.2 - Parámetros con DataProviders

Especificar parámetros en `testng.xml` puede no ser suficiente si necesita pasar parámetros complejos o parámetros que deben crearse desde Java (objetos complejos, objetos leídos desde un archivo de propiedades o una base de datos, etc.). En este caso, puede usar un proveedor de datos para proporcionar los valores que necesita probar. Un proveedor de datos es un método en su clase que devuelve una matriz de una matriz de objetos. Este método está anotado con `@DataProvider` :

```
//This
method
will
provide
data to
any test
```

```
method
that
declares
that its
Data
Provider
//is
named
"test1"
@DataProvider(name
= "test1")
public Object[][]
createData1() {
 return new Object[][]
{
   { "Cedric", new Integer(36)
},
   { "Anne", new Integer(37)},
 };
}

// Este método de ensayo declara que sus datos deben ser
suministrados por el proveedor de datos

//named
"test1"
@Test(dataProvider
= "test1")
public void verifyData1(String
n1, Integer n2) {
 System.out.println(n1
+ " " + n2);
}
```

imprimirá

```
Cedric 36

Anne 37
```

Un método `@Test` especifica su proveedor de datos con el atributo `dataProvider` . Este nombre debe corresponder a un método en la misma clase anotado con `@DataProvider(name="...")` con un nombre coincidente.

De forma predeterminada, el proveedor de datos se buscará en la clase de prueba actual o en una de sus clases base. Si desea colocar su proveedor de datos en una clase diferente, debe ser un método estático o una clase con un constructor sin argumentos, y debe especificar la clase donde se puede encontrar en el atributo `dataProviderClass` :

```
public class StaticProvider
{
  @DataProvider(name
= "create")
  public static Object[][]
createData() {
    return new Object[][]
{
      new Object[]
{ new Integer(42)
}
    };
  }
}

public class MyTest
{
  @Test(dataProvider
= "create",
dataProviderClass =
StaticProvider.class)
  public void test(Integer
n) {
    //
...
  }
}
```

El proveedor de datos también admite la inyección. TestNG utilizará el contexto de prueba para la inyección. El método Proveedor de datos puede devolver uno de los siguientes tipos:

- Una matriz de matriz de objetos (`Object[][]`) donde el tamaño de la primera dimensión es la cantidad de veces que se invocará el método de prueba y el tamaño de la segunda dimensión contiene una matriz de objetos que deben ser compatibles con los tipos de parámetros de la prueba método. Este es el caso ilustrado por el ejemplo anterior.
- Un `Iterador<Objeto[]>` . La única diferencia con `Object[][]` es que `Iterator` le permite crear sus datos de prueba de forma perezosa. TestNG invocará el iterador y luego el método de prueba con los parámetros devueltos por este iterador uno por uno. Esto es particularmente útil si tiene muchos conjuntos de parámetros para pasar al método y no desea crearlos todos por adelantado.
    - Una matriz de objetos (`Object[]`). Esto es similar a `Iterator<Object[]>` pero hace que el método de prueba se invoque una vez para cada elemento de la matriz de origen.
    - Un `Iterador<Objeto>>` . Alternativa perezosa de `Object[]` . Hace que el método de prueba se invoque una vez para cada elemento del iterador.

Debe decirse que el tipo de devolución no se limita solo a `Objeto , por lo que también son posibles MyCustomData[][]` o `Iterator<Supplier>` . La única limitación es que, en caso de iterador, su tipo de parámetro no se puede parametrizar explícitamente. Aquí hay un ejemplo de esta característica:

```
@DataProvider(name
= "test1")
public Iterator<Object[]>
createData() {
  return new MyIterator(DATA);
}
```

Uso `de MyCustomData[]` como tipo de retorno

```
@DataProvider(name
= "test1")
public MyCustomData[]
createData() {
  return new MyCustomData[]{ new MyCustomData()
};
}
```

O su opción perezosa con `Iterator<MyCustomData>`

```
@DataProvider(name
= "test1")
public Iterator<MyCustomData>
createData() {
  return Arrays.asList(new MyCustomData()).iterator();
}
```

El tipo de parámetro (`Stream`) de `Iterator` no se puede parametrizar explícitamente

```
@DataProvider(name
= "test1")
public Iterator<Stream>
createData() {
  return Arrays.asList(Stream.of("a", "b", "c")).iterator();
}
```

Si declara que su `@DataProvider` toma `java.lang.reflect.Method` como primer parámetro, TestNG pasará el método de prueba actual para este primer parámetro. Esto es particularmente útil cuando varios métodos de prueba usan el mismo `@DataProvider` y desea que arroje diferentes valores según el método de prueba para el que proporciona datos.

Por ejemplo, el siguiente código imprime el nombre del método de prueba dentro de su `@DataProvider` :

```
@DataProvider(name
= "dp")
public Object[][]
createData(Method
m) {
  System.out.println(m.getName());  //
print test method name
  return new Object[][]
{ new Object[]
{ "Cedric" }};
}
```

```
@Test(dataProvider
= "dp")
public void test1(String
s) {
}

@Test(dataProvider
= "dp")
public void test2(String
s) {
}
```

y por lo tanto mostrará:

```
test1
test2
```

Los proveedores de datos pueden ejecutarse en paralelo con el atributo `paralelo` :

```
@DataProvider(parallel
= true)
//
...
```

Los proveedores de datos paralelos que se ejecutan desde un archivo XML comparten el mismo conjunto de subprocesos, que tiene un tamaño de 10 de forma predeterminada. Puede modificar este valor en la etiqueta `<suite>` de su archivo XML:

```
<suite name="Suite1" data-
provider-thread-
count="20" >
...
```

Si desea ejecutar algunos proveedores de datos específicos en un grupo de subprocesos diferente, debe ejecutarlos desde un archivo XML diferente.

5.6.3 - Parámetros de Propiedades del Sistema

A TestNG se le pueden pasar parámetros en la línea de comando de la máquina virtual de Java usando las propiedades del sistema (-D). No es necesario que los parámetros pasados de esta manera estén predefinidos en `testng.xml` , pero anularán cualquier parámetro definido allí.

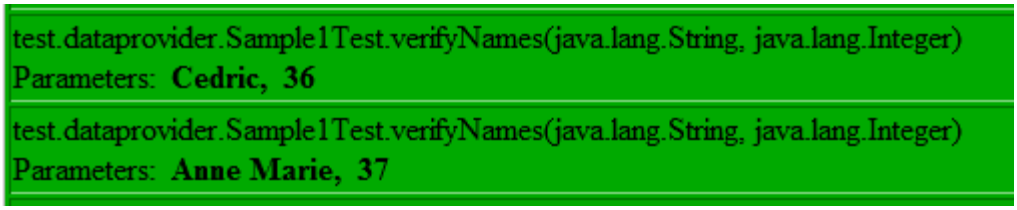
```
java -Dfirst-
name=Cedrick -
Dlast-name="von
Braun"
org.testng.TestNG
testng.xml
```

La variable de propiedad del sistema Java es una cadena sin espacios que representa el nombre de la propiedad. La variable de valor es una cadena que representa el valor de la propiedad. Si el valor es una cadena con espacios, escríbalo entre comillas.

Nota: En TestNG 6.x , las propiedades del sistema no podían sobrescribir los parámetros definidos en `testng.xml`

5.6.4 - Parámetros en informes

Los parámetros utilizados para invocar sus métodos de prueba se muestran en los informes HTML generados por TestNG. Aquí hay un ejemplo:



5.7 - Dependencias

A veces, necesita invocar sus métodos de prueba en un orden determinado. Aquí están algunos ejemplos:

- Para asegurarse de que una cierta cantidad de métodos de prueba se hayan completado y hayan tenido éxito antes de ejecutar más métodos de prueba.
- Para inicializar sus pruebas mientras desea que estos métodos de inicialización también sean métodos de prueba (los métodos etiquetados con `@Antes/Después` no serán parte del informe final).

TestNG le permite especificar dependencias con anotaciones o en XML.

5.7.1 - Dependencias con anotaciones

Puede usar los atributosdependOnMethods `odependOnGroups , que` se encuentran en la `anotación` @Test .

Hay dos tipos de dependencias:

- **Dependencias duras** . Todos los métodos de los que depende deben haberse ejecutado y tener éxito para que pueda ejecutarlos. Si ocurrió al menos una falla en sus dependencias, no será invocado y marcado como SALTO en el informe.
- **Dependencias blandas** . Siempre se ejecutará después de los métodos de los que depende, incluso si algunos de ellos han fallado. Esto es útil cuando solo desea asegurarse de que sus métodos de prueba se ejecutan en un orden determinado, pero su éxito no depende realmente del éxito de los demás. Se obtiene una dependencia blanda agregando `"alwaysRun=true"` en su anotación `@Test` .

Aquí hay un ejemplo de una dependencia dura:

```
@Test
public void serverStartedOk()
{}

@Test(dependsOnMethods
=
{ "serverStartedOk" })
public void method1()
{}
```

En este ejemplo, `method1()` se declara como dependiente del método serverStartedOk(), lo que garantiza que serverStartedOk() siempre se invocará primero.

También puede tener métodos que dependen de grupos completos:

```
@Test(groups
=
{ "init" })
public void serverStartedOk()
{}

@Test(groups
=
{ "init" })
public void initEnvironment()
{}

@Test(dependsOnGroups
= { "init.*" })
public void method1()
{}
```

En este ejemplo, method1() se declara como dependiente de cualquier grupo que coincida con la expresión regular "init.*", lo que garantiza que los métodos `serverStartedOk()` e `initEnvironment()` siempre se invocarán antes que `method1()` .

Nota: como se indicó anteriormente, no se garantiza que el orden de invocación de los métodos que pertenecen al mismo grupo sea el mismo en todas las ejecuciones de prueba.

Si un método del que dependía falla y tiene una fuerte dependencia de él (`alwaysRun=false` , que es el valor predeterminado), los métodos que dependen de él **no** se marcan como `FAIL` sino como `SKIP` . Los métodos omitidos se informarán como tales en el informe final (en un color que no es ni rojo ni verde en HTML), lo cual es importante ya que los métodos omitidos no son necesariamente fallas.

Tanto `DependOnGroups` como `DependOnMethods` aceptan expresiones regulares como parámetros. Para `dependOnMethods` , si depende de un método que tiene varias versiones sobrecargadas, se invocarán todos los métodos sobrecargados. Si solo desea invocar uno de los métodos sobrecargados, debe usar `dependOnGroups` .

Para ver un ejemplo más avanzado de métodos dependientes, consulte este artículo , que utiliza la herencia para proporcionar una solución elegante al problema de las dependencias múltiples.

De forma predeterminada, los métodos dependientes se agrupan por clase. Por ejemplo, si el método `b()` depende del método `a()` y tiene varias instancias de la clase que contiene estos métodos (debido a una fábrica de un proveedor de datos), entonces el orden de invocación será el siguiente:

```
a(1)
a(2)
b(2)
b(2)
```

TestNG no ejecutará `b()` hasta que todas las instancias hayan invocado su método `a()` .

Este comportamiento puede no ser deseable en ciertos escenarios, como por ejemplo probar un inicio y cierre de sesión de un navegador web para varios países. En tal caso, le gustaría el siguiente orden:

```
signIn("us")
signOut("us")
signIn("uk")
signOut("uk")
```

Para esta ordenación, puede utilizar el atributo XML `group-by-instances` . Este atributo es válido en <suite> o <test>:

```
  <suite name="Factory" group-
by-instances="true">
or
  <test name="Factory" group-
by-instances="true">
```

5.7.2 - Dependencias en XML

Como alternativa, puede especificar las dependencias de su grupo en el archivo `testng.xml` . Utiliza la etiqueta `<dependencies>` para lograr esto:

```
<test name="My
suite">
  <groups>
    <dependencies>
      <group name="c" depends-
on="a  b" />
      <group name="z" depends-
on="c" />
    </dependencies>
  </groups>
</test>
```

El atributo `<depends-on>` contiene una lista de grupos separados por espacios.

5.8 – Fábricas

Las fábricas le permiten crear pruebas dinámicamente. Por ejemplo, imagine que desea crear un método de prueba que accederá a una página en un sitio web varias veces y desea invocarlo con diferentes valores:

```
public class TestWebServer
{
  @Test(parameters
= { "number-of-
times" })
  public void accessPage(int numberOfTimes)
{
    while (numberOfTimes-
- > 0) {
     //
access
```

```
the web
page
    }
  }
}
<test
name="T1">
  <parameter
name="number-of-
times" value="10"/>
  <classes>
    <class name= "TestWebServer" />
  </classes>
</test>

<test
name="T2">
  <parameter
name="number-of-
times" value="20"/>
  <classes>
    <class name= "TestWebServer"/>
  </classes>
</test>

<test
name="T3">
  <parameter
name="number-of-
times" value="30"/>
  <classes>
    <class name= "TestWebServer"/>
  </classes>
</test>
```

Esto puede volverse rápidamente imposible de administrar, por lo que en su lugar, debe usar una fábrica:

```
public class WebTestFactory
{
  @Factory
  public Object[]
createInstances()
{
   Object[] result
= new Object[10];
   for (int i
= 0; i < 10;
i++) {
```

```
      result[i]
= new WebTest(i
* 10);
    }
    return result;
  }
}
```

y la nueva clase de prueba es ahora:

```
public class WebTest
{
  private int m_numberOfTimes;
  public WebTest(int numberOfTimes)
{
    m_numberOfTimes
= numberOfTimes;
  }

  @Test
  public void testServer()
{
   for (int i
= 0; i <
m_numberOfTimes;
i++) {
     //
access
the web
page
    }
  }
}
```

Su `testng.xml` solo necesita hacer referencia a la clase que contiene el método de fábrica, ya que las instancias de prueba se crearán en tiempo de ejecución:

```
<class name="WebTestFactory" />
```

O, si crea una instancia de un conjunto de pruebas programáticamente, puede agregar la fábrica de la misma manera que para las pruebas:

```
TestNG testNG
= new TestNG();
testNG.setTestClasses(WebTestFactory.class);
testNG.run();
```

El método de fábrica puede recibir parámetros como `@Test` y `@Before/After` y debe devolver `Object[]` . Los objetos devueltos pueden ser de cualquier clase (no necesariamente la misma

clase que la clase de fábrica) y ni siquiera necesitan contener anotaciones de TestNG (en cuyo caso TestNG las ignorará).

Las fábricas también se pueden usar con proveedores de datos, y puede aprovechar esta funcionalidad colocando la anotación `@Factory` en un método regular o en un constructor. Aquí hay un ejemplo de una fábrica de constructores:

```
@Factory(dataProvider
= "dp")
public FactoryDataProviderSampleTest(int n)
{
  super(n);
}

@DataProvider
static public Object[][]
dp() {
  return new Object[][]
{
    new Object[]
{ 41 },
    new Object[]
{ 42 },
  };
}
```

El ejemplo hará que TestNG cree dos clases de prueba, una con el constructor invocado con el valor 41 y la otra con 42.

5.9 - Anotaciones de nivel de clase

La anotación `@Test` se puede poner en una clase en lugar de un método de prueba:

```
@Test
public class Test1
{
  public void test1()
{
  }

  public void test2()
{
  }
}
```

`El efecto de una anotación @Test` a nivel de clase es hacer que todos los métodos públicos de esta clase se conviertan en métodos de prueba incluso si no están anotados. Todavía puede repetir la anotación `@Test` en un método si desea agregar ciertos atributos.

Por ejemplo:

```
@Test
```

```
public class Test1
{
  public void test1()
{
  }

  @Test(groups
= "g1")
  public void test2()
{
  }
}
```

hará que los métodos de prueba `test1()` y `test2() pero además de eso, test2()` ahora pertenece al grupo "g1".

5.10 - Ignorar pruebas

TestNG le permite ignorar todos los métodos `@Test` :

- En una clase (o)
- En un paquete particular (o)
- En un paquete y todos sus paquetes secundarios

usando la nueva anotación `@Ignore` .

Cuando se usa a nivel de método, la anotación `@Ignore` es funcionalmente equivalente a `@Test(enabled=false)` . Aquí hay una muestra que muestra cómo ignorar todas las pruebas dentro de una clase.

```
import org.testng.annotations.Ignore;
import org.testng.annotations.Test;

@Ignore
public class TestcaseSample
{

    @Test
    public void testMethod1()
{
    }

    @Test
    public void testMethod2()
{
    }
}
```

La anotación `@Ignore` tiene una prioridad más alta que las anotaciones individuales del método `@Test` . Cuando `@Ignore` se coloca en una clase, todas las pruebas en esa clase se desactivarán.

Para ignorar todas las pruebas en un paquete en particular, solo necesita crear `package-info.java` y agregarle la anotación `@Ignore` . Aquí hay una muestra:

```
@Ignore
package com.testng.master;

import org.testng.annotations.Ignore;
```

Esto hace que se ignoren todos los métodos `@Test` en el paquete `com.testng.master` y todos sus subpaquetes.

5.11 - Paralelismo y tiempos muertos

Puede indicarle a TestNG que ejecute sus pruebas en subprocesos separados de varias maneras.

5.11.1 - Suites paralelas

Esto es útil si está ejecutando varios archivos de conjunto (por ejemplo, " `java org.testng.TestNG testng1.xml testng2.xml` ") y desea que cada uno de estos conjuntos se ejecute en un subproceso separado. Puede usar el siguiente indicador de línea de comando para especificar el tamaño de un grupo de subprocesos:

```
java
org.testng.TestNG -
suitethreadpoolsize
3 testng1.xml
testng2.xml
testng3.xml
```

El nombre de la tarea ant correspondiente es `suitethreadpoolsize` .

5.11.2 - Pruebas paralelas, clases y métodos

El atributo *paralelo* en la etiqueta <suite> puede tomar uno de los siguientes valores:

```
<suite name="My
suite" parallel="methods" thread-
count="5">
<suite name="My
suite" parallel="tests" thread-
count="5">
<suite name="My
suite" parallel="classes" thread-
count="5">
<suite name="My
suite" parallel="instances" thread-
count="5">
```

- **`paralelo="métodos"`** : TestNG ejecutará todos sus métodos de prueba en subprocesos separados. Los métodos dependientes también se ejecutarán en subprocesos separados, pero respetarán el orden que especificó.

- **`paralelo="pruebas"`** : TestNG ejecutará todos los métodos en la misma etiqueta <test> en el mismo hilo, pero cada etiqueta <test> estará en un hilo separado. Esto le permite agrupar todas sus clases que no son seguras para subprocesos en el mismo <test> y garantizar que todas se ejecutarán en el mismo subproceso mientras aprovecha TestNG utilizando tantos subprocesos como sea posible para ejecutar sus pruebas.

- **`paralelo="clases"`** : TestNG ejecutará todos los métodos en la misma clase en el mismo hilo, pero cada clase se ejecutará en un hilo separado.

- **`paralelo="instancias"`** : TestNG ejecutará todos los métodos en la misma instancia en el mismo subproceso, pero dos métodos en dos instancias diferentes se ejecutarán en subprocesos diferentes.

Además, el atributo *thread-count* le permite especificar cuántos hilos se deben asignar para esta ejecución.

Nota: el atributo `@Test timeOut` funciona tanto en modo paralelo como no paralelo.

También puede especificar que se invoque un método `@Test` desde diferentes subprocesos. Puede usar el atributo `threadPoolSize` para lograr este resultado:

```
@Test(threadPoolSize
= 3, invocationCount
= 10,  timeOut
= 10000)
public void testServer()
{
```

En este ejemplo, la función `testServer` se invocará diez veces desde tres subprocesos diferentes. Además, un tiempo de espera de diez segundos garantiza que ninguno de los subprocesos se bloqueará en este subproceso para siempre.

5.12 - Reejecutar pruebas fallidas

Cada vez que fallan las pruebas en una suite, TestNG crea un archivo llamado `testng-failed.xml` en el directorio de salida. Este archivo XML contiene la información necesaria para volver a ejecutar solo estos métodos que fallaron, lo que le permite reproducir rápidamente las fallas sin tener que ejecutar la totalidad de sus pruebas. Por lo tanto, una sesión típica se vería así:

```
java -classpath
testng.jar;%CLASSPATH%
org.testng.TestNG -d
test-outputs
testng.xml
java -classpath
testng.jar;%CLASSPATH%
org.testng.TestNG -d
test-outputs test-
```

```
outputs\testng-
failed.xml
```

Tenga en cuenta que `testng-failed.xml` contendrá todos los métodos dependientes necesarios para que tenga la garantía de ejecutar los métodos que fallaron sin fallas de SKIP.

A veces, es posible que desee que TestNG vuelva a intentar automáticamente una prueba cada vez que falla. En esas situaciones, puede utilizar un analizador de reintentos. Cuando vincula un analizador de reintentos a una prueba, TestNG invoca automáticamente el analizador de reintentos para determinar si TestNG puede volver a intentar un caso de prueba nuevamente en un intento de ver si la prueba que acaba de fallar ahora pasa. Así es como se utiliza un analizador de reintentos:

1. Cree una implementación de la interfaz `org.testng.IRetryAnalyzer`
2. Vincule esta implementación a la anotación `@Test` , por ejemplo, `@Test(retryAnalyzer = LocalRetry.class)`

A continuación se muestra una implementación de muestra del analizador de reintentos que vuelve a intentar una prueba un máximo de tres veces.

```
import org.testng.IRetryAnalyzer;
import org.testng.ITestResult;

public class MyRetry implements IRetryAnalyzer
{

  private int retryCount
= 0;
  private static final int maxRetryCount
= 3;

  @Override
  public boolean retry(ITestResult
result) {
    if (retryCount
< maxRetryCount) {
      retryCount++;
      return true;
    }
    return false;
  }
}
import org.testng.Assert;
import org.testng.annotations.Test;

public class TestclassSample
{

  @Test(retryAnalyzer
= MyRetry.class)
```

```
  public void test2()
{
    Assert.fail();
  }
}
```

5.13 - Pruebas JUnit

TestNG puede ejecutar pruebas JUnit 3 y JUnit 4. Todo lo que necesita hacer es colocar el archivo jar JUnit en el classpath, especificar sus clases de prueba JUnit en la propiedad `testng.classNames` y establecer la propiedad `testng.junit` en verdadero:

```
<test name="Test1" junit="true">
  <classes>
    <!-
- ... -
->
```

El comportamiento de TestNG en este caso es similar a JUnit según la versión de JUnit que se encuentre en la ruta de clase:

- JUnidad 3:
  - Se ejecutarán todos los métodos que comiencen con test* en sus clases
  - Si hay un método setUp () en su clase de prueba, se invocará antes de cada método de prueba
  - Si hay un método tearDown() en su clase de prueba, se invocará antes después de cada método de prueba
  - Si su clase de prueba contiene un conjunto de métodos (), se invocarán todas las pruebas devueltas por este método
- Unidad 4:
  - TestNG utilizará el corredor `org.junit.runner.JUnitCore` para ejecutar sus pruebas

5.14 - Ejecutar TestNG programáticamente

Puede invocar TestNG desde sus propios programas muy fácilmente:

```
TestListenerAdapter tla
= new TestListenerAdapter();
TestNG testng
= new TestNG();
testng.setTestClasses(new Class[]
{ Run2.class });
testng.addListener(tla);
testng.run();
```

Este ejemplo crea un objeto `TestNG` y ejecuta la clase de prueba `Run2` . También agrega un `TestListener` . Puede usar la clase de adaptador `org.testng.TestListenerAdapter` o implementar `org.testng.ITestListener` usted mismo. Esta interfaz contiene varios métodos de devolución de llamada que le permiten realizar un seguimiento de cuándo comienza una prueba, tiene éxito, falla, etc.

Del mismo modo, puede invocar TestNG en un archivo `testng.xml` o puede crear un archivo `testng.xml` virtual usted mismo. Para hacer esto, puede usar las clases que se encuentran en el

paquete `org.testng.xml` : `XmlClass` , `XmlTest` , etc... Cada una de estas clases corresponde a su contraparte de etiqueta XML.

Por ejemplo, suponga que desea crear el siguiente archivo virtual:

```
<suite
name="TmpSuite" >
  <test
name="TmpTest" >
    <classes>
      <class name="test.failures.Child"  />
    <classes>
    </test>
</suite>
```

Usarías el siguiente código:

```
XmlSuite suite
= new XmlSuite();
suite.setName("TmpSuite");

XmlTest test
= new XmlTest(suite);
test.setName("TmpTest");
List<XmlClass> classes
= new ArrayList<XmlClass>();
classes.add(new XmlClass("test.failures.Child"));
test.setXmlClasses(classes)
;
```

Y luego puedes pasar este `XmlSuite` a TestNG:

```
List<XmlSuite> suites
= new ArrayList<XmlSuite>();
suites.add(suite);
TestNG tng
= new TestNG();
tng.setXmlSuites(suites);
tng.run();
```

Consulte los JavaDocs para conocer toda la API.

5.15 - BeanShell y selección avanzada de grupos

Si las etiquetas `<include>` y `<exclude>` en `testng.xml` no son suficientes para sus necesidades, puede usar una expresión BeanShell para decidir si un determinado método de prueba debe incluirse o no en una ejecución de prueba. Especifique esta expresión justo debajo de la etiqueta `<test>` :

```
<test name="BeanShell
test">
```

```
   <method-
selectors>
     <method-
selector>
       <script language="beanshell"><![CDATA[
         groups.containsKey("test1")
       ]]></script>
     </method-
selector>
   </method-
selectors>
  <!-
- ...
-->
```

Cuando se encuentra una etiqueta `<script> en testng.xml` , TestNG ignorará los subsiguientes `<include>` y `<exclude>` de grupos y métodos en la etiqueta `<test>` actual : su expresión BeanShell será la única forma de decidir si un método de prueba está incluido o no.

Aquí hay información adicional sobre el script BeanShell:

- Debe devolver un valor booleano. Excepto por esta restricción, se permite cualquier código BeanShell válido (por ejemplo, es posible que desee devolver `verdadero` durante los días de semana y falso durante los fines de semana, lo que le permitiría ejecutar pruebas de manera diferente según la fecha).

- TestNG define las siguientes variables para su comodidad:
  `java.lang.reflect.Method method` : el método de prueba actual.
  org.testng.ITestNGMethod testngMethod : la descripción del método de prueba actual.
  `java.util.Map<String, String> grupos` : un mapa de los grupos a los que pertenece el método de prueba actual.

- Es posible que desee rodear su expresión con una declaración `CDATA` (como se muestra arriba) para evitar las tediosas citas de caracteres XML reservados).

Nota:

```
A partir de la versión 7.5 , TestNG no genera ninguna dependencia de las
implementaciones de BeanShell de forma predeterminada.
Entonces, para aprovechar los selectores de métodos basados en BeanShell, recuerde
agregar una dependencia explícita en BeanShell.
Por ejemplo org.apache-extras.beanshell
```

5.16 - Transformadores de anotación

TestNG le permite modificar el contenido de todas las anotaciones en tiempo de ejecución. Esto es especialmente útil si las anotaciones en el código fuente son correctas la mayor parte del tiempo, pero hay algunas situaciones en las que le gustaría anular su valor.

Para lograr esto, necesita usar un transformador de anotación.

Un Annotation Transformer es una clase que implementa la siguiente interfaz:

```
public interface IAnnotationTransformer
{

  /**
   * This method will be invoked by TestNG to give you a chance

   * to modify a TestNG annotation read from your test classes.

   * You can change the values you need by calling any of the

   * setters on the ITest interface.

   *

   * Note that only one of the three parameters testClass,

   * testConstructor and testMethod will be non-null.

   *

   * @param annotation The annotation that was read from your


    * test class.

   * parameter represents this class (null otherwise).

    * @param testConstructor If the annotation was found on a constructor,

   * @param testClass If the annotation was found on a class, this

   * this parameter represents this constructor (null otherwise).

   * @param testMethod If the annotation was found on a method,

   * this parameter represents this method (null otherwise).

   */


public void transform(ITest
annotation, Class
testClass,
      Constructor
testConstructor,
```

```
Method
testMethod);
}
```

Al igual que todos los demás oyentes de TestNG, puede especificar esta clase en la línea de comando o con ant:

```
java
org.testng.TestNG
-listener
MyTransformer
testng.xml
```

o programáticamente:

```
TestNG tng
= new TestNG();
tng.setAnnotationTransformer(new MyTransformer());
//
...
```

Cuando se invoca el método `transform()` , puede llamar a cualquiera de los configuradores en el parámetro de `prueba ITest` para modificar su valor antes de que TestNG continúe.

Por ejemplo, así es como anularía el atributo `invocationCount` pero solo en el método de prueba `invocar()` de una de sus clases de prueba:

```
public class MyTransformer implements IAnnotationTransformer
{
  public void transform(ITest
annotation, Class testClass,
      Constructor
testConstructor,
Method
testMethod)
  {
    if ("invoke".equals(testMethod.getName()))
{
      annotation.setInvocationCount(5);
    }
  }
}
```

`IAnnotationTransformer` solo le permite modificar una anotación `@Test` . Si necesita modificar otra anotación de TestNG (una anotación de configuración, `@Factory` o `@DataProvider`), use un `IAnnotationTransformer2` .

5.17 - Interceptores de métodos

Una vez que TestNG ha calculado en qué orden se invocarán los métodos de prueba, estos métodos se dividen en dos grupos:

- *Los métodos se ejecutan secuencialmente* . Estos son todos los métodos de prueba que tienen dependencias o dependientes. Estos métodos se ejecutarán en un orden específico.

- *Los métodos se ejecutan sin ningún orden en particular* . Estos son todos los métodos que no pertenecen a la primera categoría. El orden en que se ejecutan estos métodos de prueba es aleatorio y puede variar de una ejecución a la siguiente (aunque de manera predeterminada, TestNG intentará agrupar los métodos de prueba por clase).

Para darle más control sobre los métodos que pertenecen a la segunda categoría, TestNG define la siguiente interfaz:

```
public interface IMethodInterceptor
{

  List<IMethodInstance>
intercept(List<IMethodInstance>
methods, ITestContext context);

}
```

La lista de métodos pasados en parámetros son todos los métodos que se pueden ejecutar en cualquier orden. Se espera que su método de `intercepción devuelva una lista similar de IMethodInstance` , que puede ser cualquiera de las siguientes:

- La misma lista que recibió en el parámetro pero en un orden diferente.
- Una lista más pequeña de objetos `IMethodInstance` .
- Una lista más grande de objetos `IMethodInstance` .

Una vez que haya definido su interceptor, lo pasa a TestNG como oyente. Por ejemplo:

```
java -classpath "testng-jdk15.jar:test/build"
org.testng.TestNG -listener
test.methodinterceptors.NullMethodInterceptor
   -testclass
test.methodinterceptors.FooTest
```

Para conocer la sintaxis de `ant` equivalente , consulte el atributo `listeners en la` documentación de ant .

Por ejemplo, aquí hay un interceptor de métodos que reordenará los métodos para que los métodos de prueba que pertenecen al grupo "rápido" siempre se ejecuten primero:

```
public List<IMethodInstance>
intercept(List<IMethodInstance>
methods, ITestContext context)
{
  List<IMethodInstance> result
= new ArrayList<IMethodInstance>();
  for (IMethodInstance
m : methods) {
    Test test =
m.getMethod().getConstructorOrMethod().getAnnotation(Test.class);
    Set<String> groups
= new HashSet<String>();
    for (String
group :
```

```
test.groups())
{
      groups.add(group);
    }
    if (groups.contains("fast"))
{
      result.add(0,
m);
    }
    else {
      result.add(m);
    }
  }
  return result;
}
```

5.18 - Listeners TestNG

Hay varias interfaces que le permiten modificar el comportamiento de TestNG. Estas interfaces se denominan ampliamente "Oyentes TestNG". Aquí hay algunos oyentes:

- `IAnnotationTransformer` (doc , javadoc)
- `IAnnotationTransformer2` (doc , javadoc)
- `IHookable` (doc , javadoc)
- `IInvokedMethodListener` (doc, javadoc)
- `IMethodInterceptor` (doc , javadoc)
- `Reportero` (doc , javadoc)
- `ISuiteListener` (doc, javadoc)
- `ITestListener` (doc , javadoc)

Cuando implementa una de estas interfaces, puede informar a TestNG de cualquiera de las siguientes maneras:

- Usando -listener en la línea de comando.
- Uso de <oyentes> con ant.
- Usando <oyentes> en su archivo `testng.xml` .
- Usando la anotación `@Listeners` en cualquiera de sus clases de prueba.
- Uso `de ServiceLoader` .

5.18.1 - Especificación de oyentes con `testng.xml` o en Java

Así es como puede definir oyentes en su archivo `testng.xml` :

```
<suite>

  <listeners>
    <listener class-
name="com.example.MyListener" />
    <listener class-
name="com.example.MyMethodInterceptor" />
```

```
  </listeners>

 ...
```

O si prefiere definir estos oyentes en Java:

```
@Listeners({
com.example.MyListener.class,
com.example.MyMethodInterceptor.class })
public class MyTest
{
  //
...
}
```

La anotación `@Listeners` puede contener cualquier clase que amplíe `org.testng.ITestNGListener` **excepto** `IAnnotationTransformer` e `IAnnotationTransformer2` . La razón es que estos oyentes deben conocerse muy temprano en el proceso para que TestNG pueda usarlos para reescribir sus anotaciones, por lo tanto, debe especificar estos oyentes en su archivo `testng.xml` .

Tenga en cuenta que la anotación `@Listeners` se aplicará a todo su archivo de suite, como si lo hubiera especificado en un archivo `testng.xml` . Si desea restringir su alcance (por ejemplo, solo se ejecuta en la clase actual), el código en su oyente primero podría verificar el método de prueba que está a punto de ejecutarse y decidir qué hacer entonces. Así es como se puede hacer.

1. Primero defina una nueva anotación personalizada que se pueda usar para especificar esta restricción:

```
@Retention(RetentionPolicy.RUNTIME)
@Target ({ElementType.TYPE})
public @interface DisableListener
{}
```

4. Agregue una verificación de edición como se muestra a continuación dentro de sus oyentes habituales:

```
public void beforeInvocation(IInvokedMethod
iInvokedMethod, ITestResult iTestResult) {
  ConstructorOrMethod consOrMethod
=iInvokedMethod.getTestMethod().getConstructorOrMethod();
  DisableListener disable =
consOrMethod.getMethod().getDeclaringClass().getAnnotation(D
isableListener.class);
  if (disable
!= null) {
    return;
  }
  // else
resume
your
```

```
normal
operations
}
```

12. Anote las clases de prueba en las que no se va a invocar al oyente:

```
@DisableListener
@Listeners({
com.example.MyListener.class,
com.example.MyMethodInterceptor.class })
public class MyTest
{
  //
...
}
```

5.18.2 - Especificación de listeners con `ServiceLoader`

Finalmente, el JDK ofrece un mecanismo muy elegante para especificar implementaciones de interfaces en el classpath a través de la clase `ServiceLoader` .

Con ServiceLoader, todo lo que necesita hacer es crear un archivo jar que contenga su (s) oyente (s) y algunos archivos de configuración, coloque ese archivo jar en el classpath cuando ejecute TestNG y TestNG los encontrará automáticamente.

Aquí hay un ejemplo concreto de cómo funciona.

Comencemos por crear un oyente (cualquier oyente de TestNG debería funcionar):

```
package test.tmp;

public class TmpSuiteListener implements ISuiteListener
{
  @Override
  public void onFinish(ISuite
suite) {
    System.out.println("Finishing");
  }

  @Override
  public void onStart(ISuite
suite) {
    System.out.println("Starting");
  }
}
```

Compile este archivo, luego cree un archivo en la ubicación `META-INF/services/org.testng.ITestNGListener` , que nombrará las implementaciones que desea para esta interfaz.

Debería terminar con la siguiente estructura de directorios, con solo dos archivos:

```
$
tree
|____META-
INF
|
|____services
| |
|____org.testng.ITestNGListener
|____test
|
|____tmp
| |
|____TmpSuiteListener.class

$ cat META-
INF/services/org.testng.ITestNGListener
test.tmp.TmpSuiteListener
```

Crea un jar de este directorio:

```
$ jar cvf
../sl.jar
.
added
manifest
ignoring
entry
META-
INF/
adding: META-
INF/services/(in
= 0) (out=
0)(stored 0%)
adding: META-
INF/services/org.testng.ITestNGListener(in
= 26) (out= 28)(deflated -7%)
adding:
test/(in
= 0)
(out=
0)(stored
0%)
adding:
test/tmp/(in
= 0) (out=
```

```
0)(stored
0%)
adding:
test/tmp/TmpSuiteListener.class(in
= 849) (out= 470)(deflated 44%)
```

A continuación, coloque este archivo jar en su classpath cuando invoque TestNG:

```
$ java -classpath
sl.jar:testng.jar
org.testng.TestNG
testng-
single.yaml
Starting
f2
11
2
PASSED:
f2("2")
Finishing
```

Este mecanismo le permite aplicar el mismo conjunto de oyentes a toda una organización simplemente agregando un archivo jar a la ruta de clase, en lugar de pedirle a cada desarrollador que recuerde especificar estos oyentes en su archivo testng.xml.

5.19 - Inyección de dependencia

TestNG admite dos tipos diferentes de inyección de dependencia: nativa (realizada por el propio TestNG) y externa (realizada por un marco de inyección de dependencia como Guice).

5.19.1 - Inyección de dependencia nativa

TestNG le permite declarar parámetros adicionales en sus métodos. Cuando esto sucede, TestNG llenará automáticamente estos parámetros con el valor correcto. La inyección de dependencia se puede utilizar en los siguientes lugares:

- Cualquier método @Before o método @Test puede declarar un parámetro de tipo `ITestContext` .
- Cualquier método @AfterMethod puede declarar un parámetro de tipo `ITestResult` , que reflejará el resultado del método de prueba que se acaba de ejecutar.
- Cualquier método @Before y @After (excepto @BeforeSuite y @AfterSuite) puede declarar un parámetro de tipo `XmlTest` , que contiene la etiqueta `<test> actual.`
- Cualquier @BeforeMethod (y @AfterMethod) puede declarar un parámetro de tipo `java.lang.reflect.Method` . Este parámetro recibirá el método de prueba que se llamará una vez que finalice este @BeforeMethod (o después de que el método se ejecute para @AfterMethod).
- Cualquier @BeforeMethod puede declarar un parámetro de tipo `Object[]` . Este parámetro recibirá la lista de parámetros que están a punto de enviarse al próximo método de prueba, que TestNG podría inyectar, como `java.lang.reflect.Method` , o provenir de un `@DataProvider` .
- Cualquier @DataProvider puede declarar un parámetro de tipo `ITestContext` o `java.lang.reflect.Method` . Este último parámetro recibirá el método de prueba que está a punto de ser invocado.

Puede desactivar la inyección con la anotación `@NoInjection` :

```
public class NoInjectionTest
{

  @DataProvider(name
= "provider")
  public Object[][]
provide() throws Exception
{
      return new Object[][]
{ { CC.class.getMethod("f")
} };
  }

  @Test(dataProvider
= "provider")
  public void withoutInjection(@NoInjection Method
m) {
      Assert.assertEquals(m.getName(), "f");
  }

  @Test(dataProvider
= "provider")
  public void withInjection(Method
m) {
      Assert.assertEquals(m.getName(), "withInjection");
  }
}
```

La siguiente tabla resume los tipos de parámetros que se pueden inyectar de forma nativa para las diversas anotaciones de TestNG:

Anotación	Contexto de prueba de TI	XmlPrueba	Método	Objeto[]	Resultado de la prueba de TI
BeforeSuite	Sí	No	No	No	No
BeforeTest	Sí	Sí	No	No	No
BeforeGroups	Sí	Sí	No	No	No
BeforeClass	Sí	Sí	No	No	No
BeforeMethod	Sí	Sí	Sí	Sí	Sí
Test	Sí	No	No	No	No
DataProvider	Sí	No	Sí	No	No
AfterMethod	Sí	Sí	Sí	Sí	Sí
AfterClass	Sí	Sí	No	No	No
AfterGroups	Sí	Sí	No	No	No

AfterTest	Sí	Sí	No	No	No
AfterSuite	Sí	No	No	No	No

5.19.2 - Inyección de dependencia de Guice

Si usa Guice, TestNG le brinda una manera fácil de inyectar sus objetos de prueba con un módulo Guice:

```
@Guice(modules =
GuiceExampleModule.class)
public class GuiceTest extends SimpleBaseTest
{

  @Inject
  ISingleton
m_singleton;

  @Test
  public void singletonShouldWork()
{
    m_singleton.doSomething();
  }

}
```

En este ejemplo, se espera que `GuiceExampleModule` vincule la interfaz `ISingleton` a alguna clase concreta:

```
public class GuiceExampleModule implements Module
{

  @Override
  public void configure(Binder
binder) {
    binder.bind(ISingleton.class).to(ExampleSingleton.class).in(Si
ngleton.class);
  }

}
```

Si necesita más flexibilidad para especificar qué módulos deben usarse para instanciar sus clases de prueba, puede especificar una fábrica de módulos:

```
@Guice(moduleFactory
=
ModuleFactory.class)
public class GuiceModuleFactoryTest
{

  @Inject
  ISingleton
m_singleton;
```

```
  @Test
  public void singletonShouldWork()
{
    m_singleton.doSomething();
  }
}
```

La fábrica de módulos necesita implementar la interfaz IModuleFactory :

```
public interface IModuleFactory
{
 /**
   * @param context The current test context

   * @param testClass The test class

   *
   * @return The Guice module that should be used to get an instance of
this

   * test class.

   */

Module createModule(ITestContext context, Class<?> testClass);

}
```

A su fábrica se le pasará una instancia del contexto de prueba y la clase de prueba que TestNG necesita instanciar. Su método `createModule` debería devolver un Módulo Guice que sabrá cómo instanciar esta clase de prueba. Puede usar el contexto de prueba para obtener más información sobre su entorno, como los parámetros especificados en `testng.xml` , etc. Obtendrá aún más flexibilidad y potencia de Guice con los parámetros de la suite `del módulo` principal y `de la etapa de guice . guice-stage` le permite elegir la `etapa` utilizada para crear el inyector principal. El predeterminado es `DESARROLLO` . Otros valores permitidos son `PRODUCCIÓN` y `HERRAMIENTA`. Así es como puede definir el módulo principal en su archivo test.xml:

```
<suite parent-
module="com.example.SuiteParenModule" guice-
stage="PRODUCTION">
</suite>
```

TestNG creará este módulo solo una vez para la suite dada. También usará este módulo para obtener instancias de módulos Guice específicos de prueba y fábricas de módulos, luego creará un inyector secundario para cada clase de prueba. Con tal enfoque, puede declarar todos los enlaces comunes en el módulo principal y también puede inyectar enlaces declarados en el módulo principal en el módulo y la fábrica de módulos. Aquí hay un ejemplo de esta funcionalidad:

```
package com.example;

public class ParentModule extends AbstractModule
{
  @Override
```

```
  protected void conigure()
{
    bind(MyService.class).toProvider(MyServiceProvider.class);
    bind(MyContext.class).to(MyContextImpl.class).in(Singleton.cla
ss);
  }
}
package com.example;

public class TestModule extends AbstractModule
{
  private final MyContext
myContext;

  @Inject
  TestModule(MyContext
myContext) {
    this.myContext
= myContext
  }

  @Override
  protected void configure()
{
    bind(MySession.class).toInstance(myContext.getSession());
  }
}
<suite parent-
module="com.example.ParentModule">
</suite>
package com.example;

@Test
@Guice(modules =
TestModule.class)
public class TestClass
{
  @Inject
  MyService
myService;
  @Inject
  MySession
mySession;

  public void testServiceWithSession()
{
```

```
    myService.serve(mySession);
  }
}
```

Como puede ver, ParentModule declara el enlace para las clases MyService y MyContext. Luego, MyContext se inyecta mediante la inyección del constructor en la clase TestModule, que también declara el enlace para MySession. Luego, el módulo principal en el archivo XML de prueba se establece en la clase ParentModule, esto permite la inyección en TestModule. Más tarde, en TestClass, verá dos inyecciones: * MyService: enlace tomado de ParentModule * MySession: enlace tomado de TestModule Esta configuración le garantiza que todas las pruebas en este conjunto se ejecutarán con la misma instancia de sesión, el objeto MyContextImpl solo se crea una vez por conjunto, esto le brinda la posibilidad de configurar el estado del entorno común para todas las pruebas en la suite.

5.20 - Escuchar invocaciones de métodos

El oyente `IInvokedMethodListener` le permite recibir una notificación cada vez que TestNG está a punto de invocar un método de prueba (anotado con `@Test`) o configuración (anotado con cualquiera de las anotaciones `@Before` o `@After`). Necesitas implementar la siguiente interfaz:

```
public interface IInvokedMethodListener extends ITestNGListener
{
  void beforeInvocation(IInvokedMethod
method, ITestResult testResult);
  void afterInvocation(IInvokedMethod
method, ITestResult testResult);
}
```

y declararlo como oyente, como se explica en la sección sobre oyentes de TestNG .

5.21 - Métodos de prueba anulados

TestNG le permite anular y posiblemente omitir la invocación de métodos de prueba. Un ejemplo de dónde esto es útil es si necesita probar sus métodos con un administrador de seguridad específico. Esto se logra proporcionando un oyente que implementa `IHookable` .

Aquí hay un ejemplo con JAAS:

```
public class MyHook implements IHookable
{
  public void run(final IHookCallBack
icb, ITestResult testResult) {
    //
Preferably
initialized in
a
@Configuration
method
    mySubject =
authenticateWithJAAs();

    Subject.doAs(mySubject, new PrivilegedExceptionAction()
{
```

```
      public Object
run() {
        icb.callback(testResult);
      }
    };
  }
}
```

5.22 - Alteración de suites (o) pruebas

A veces, es posible que solo desee modificar una suite (o) una etiqueta de prueba en una suite xml en tiempo de ejecución sin tener que cambiar el contenido de un archivo de suite.

Un ejemplo clásico de esto sería intentar aprovechar su archivo de suite existente e intentar usarlo para simular una prueba de carga en su "Aplicación bajo prueba". Como mínimo, terminaría duplicando el contenido de su etiqueta <test> varias veces y crearía un nuevo archivo xml de suite y trabajaría con él. Pero esto no parece escalar mucho.

TestNG le permite modificar una suite (o) una etiqueta de prueba en el archivo xml de su suite en tiempo de ejecución a través de oyentes. Esto se logra proporcionando un oyente que implementa `IAlterSuiteListener` . Consulte la sección Oyentes para obtener información sobre los oyentes.

Aquí hay un ejemplo que muestra cómo se modifica el nombre de la suite en tiempo de ejecución:

```
public class AlterSuiteNameListener implements IAlterSuiteListener
{

    @Override
    public void alter(List<XmlSuite>
suites) {
        XmlSuite
suite =
suites.get(0);
        suite.setName(getClass().getSimpleName());
    }
}
```

Este oyente solo se puede agregar con cualquiera de las siguientes formas:

- A través de la etiqueta `<oyentes>` en el archivo xml de la suite.
- A través de un cargador de servicios

Este oyente no se puede agregar a la ejecución mediante la anotación `@Listeners` .

6 - Resultados de la prueba

6.1 - Éxito, fracaso y afirmación

Una prueba se considera exitosa si se completó sin generar ninguna excepción o si generó una excepción que se esperaba (consulte la documentación del atributo de excepciones esperadas que se encuentra en la `anotación @Test`).

Por lo general, sus métodos de prueba estarán compuestos por llamadas que pueden generar una excepción o varias afirmaciones (usando la palabra clave "assert" de Java). Un error de "afirmación" activará una AssertionErrorException, que a su vez marcará el método como fallido (recuerde usar -ea en la JVM si no ve los errores de afirmación).

Aquí hay un método de prueba de ejemplo:

```
@Test
public void verifyLastName()
{
  assert "Beust".equals(m_lastName)
: "Expected name Beust, for" +
m_lastName;
}
```

TestNG también incluye la clase Assert de JUnit, que le permite realizar afirmaciones en objetos complejos:

```
import static org.testng.AssertJUnit.*;
//...
@Test
public void verify()
{
  assertEquals("Beust",
m_lastName);
}
```

Tenga en cuenta que el código anterior usa una importación estática para poder usar el método `assertEquals` sin tener que prefijarlo por su clase.

6.2 - Registro y resultados

Los resultados de la ejecución de la prueba se crean en un archivo llamado `index.html` en el directorio especificado al iniciar SuiteRunner. Este archivo apunta a varios otros archivos HTML y de texto que contienen el resultado de toda la ejecución de la prueba.

Es muy fácil generar sus propios informes con TestNG with Listeners and Reporters:

- **Los oyentes** implementan la interfaz `org.testng.ITestListener` y son notificados en tiempo real cuando una prueba comienza, pasa, falla, etc...
- **Los reporteros** implementan la interfaz `org.testng.IReporter` y reciben una notificación cuando TestNG ha ejecutado todas las suites. La instancia de IReporter recibe una lista de objetos que describen toda la ejecución de la prueba.

Por ejemplo, si desea generar un informe en PDF de su ejecución de prueba, no necesita recibir una notificación en tiempo real de la ejecución de prueba, por lo que probablemente debería usar un `IReporter` . Si desea escribir un informe en tiempo real de sus pruebas, como una GUI con una barra de progreso o un reportero de texto que muestra puntos (".") a medida que se invoca cada prueba (como se explica a continuación), `ITestListener` es su mejor elección.

6.2.1 - Registro de listeners

Aquí hay un oyente que muestra un "." por cada prueba aprobada, una "F" por cada falla y una "S" por cada omisión:

```
public class DotTestListener extends TestListenerAdapter
{
  private int m_count
= 0;

  @Override
  public void onTestFailure(ITestResult
tr) {
    log("F");
  }

  @Override
  public void onTestSkipped(ITestResult
tr) {
    log("S");
  }

  @Override
  public void onTestSuccess(ITestResult
tr) {
    log(".");
  }

  private void log(String
string) {
    System.out.print(string);
    if (++m_count
% 40 == 0) {
      System.out.println("");
    }
  }
}
```

En este ejemplo, elegí extender `TestListenerAdapter` , que implementa `ITestListener` con métodos vacíos, por lo que no tengo que anular otros métodos de la interfaz que no me interesan. Puede implementar la interfaz directamente si lo prefiere.

Así es como invoco a TestNG para usar este nuevo oyente:

```
java -classpath
testng.jar;%CLASSPATH%
org.testng.TestNG -listener
org.testng.reporters.DotTestListener
test\testng.xml
```

y la salida:

```
.....................................
.....................................
.....................................
```

```
.....................................
.....................................
.....................
================================================
TestNG
JDK
1.5
Total
tests
run: 226,
Failures:
0, Skips:
0
================================================
```

Tenga en cuenta que cuando usa `-listener` , TestNG determinará automáticamente el tipo de oyente que desea usar.

6.2.2 - Registro de reporteros

La interfaz `org.testng.IReporter` solo tiene un método:

```
public void generateReport(List<ISuite>
suites, String outputDirectory)
```

TestNG invocará este método cuando se hayan ejecutado todas las suites y usted pueda inspeccionar sus parámetros para acceder a toda la información sobre la ejecución que se acaba de completar.

6.2.3 - Informes JUnit

TestNG contiene un oyente que toma los resultados de TestNG y genera un archivo XML que luego se puede enviar a JUnitReport. Aquí hay un ejemplo y la tarea ant para crear este informe:

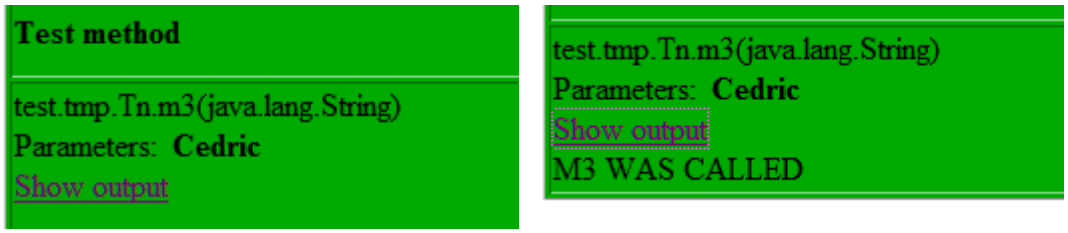
```
<target name="reports">
  <junitreport todir="test-
report">
    <fileset dir="test-
output">
      <include name="*/*.xml"/>
    </fileset>

    <report format="noframes"  todir="test-
report"/>
  </junitreport>
</target>
```

Nota: una incompatibilidad actual entre JDK 1.5 y JUnitReports impide que funcione la versión del marco, por lo que debe especificar "noframes" para que esto funcione por ahora.

6.2.4 - API del informador

Si necesita registrar mensajes que deberían aparecer en los informes HTML generados, puede usar la clase `org.testng.Reporter` :

```
Reporter.log ( "M3 FUE LLAMADO" ) ;
```



6.2.5 - Informes XML

TestNG ofrece un reportero XML que captura información específica de TestNG que no está disponible en los informes JUnit. Esto es particularmente útil cuando el entorno de prueba del usuario necesita consumir resultados XML con datos específicos de TestNG que el formato JUnit no puede proporcionar. Este reportero se puede inyectar en TestNG a través de la línea de comandos con `-reporter` .

Este es un ejemplo de uso: `-reporter org.testng.reporters.XMLReporter:generateTestResultAttributes=true,generateGroupsAttribute=true` .

El conjunto completo de opciones que se pueden pasar se detalla en la siguiente tabla. Asegúrate de usar:

- `:` - para separar el nombre del reportero de sus propiedades
- `=` - para separar pares clave/valor para propiedades
- `,` - para separar varios pares clave/valor

A continuación se muestra una muestra de la salida de dicho reportero:

```
<testng-
results>
  <suite name="Suite1">
    <groups>
      <group name="group1">
        <method signature="com.test.TestOne.test2()" name="test2"
class="com.test.TestOne"/>
        <method signature="com.test.TestOne.test1()" name="test1"
class="com.test.TestOne"/>
      </group>
      <group name="group2">
        <method signature="com.test.TestOne.test2()" name="test2"
class="com.test.TestOne"/>
      </group>
    </groups>
    <test name="test1">
      <class name="com.test.TestOne">
```

```
        <test-
method status="FAIL" signature="test1()" name="test1" duration-
ms="0"
              started-at="2007-05-
28T12:14:37Z" description="someDescription2"
              finished-
at="2007-05-
28T12:14:37Z">
          <exception class="java.lang.AssertionError">
            <short-
stacktrace>
              <![CDATA[
                java.lang.AssertionError
                ...
Removed 22 stack
frames
              ]]>
            </short-stacktrace>

          </exception>
        </test-method>


        <test-
method status="PASS" signature="test2()" name="test2" duration-
ms="0"
              started-at="2007-05-
28T12:14:37Z" description="someDescription1"
              finished-
at="2007-05-
28T12:14:37Z">
        </test-
method>
        <test-
method status="PASS" signature="setUp()" name="setUp" is-
config="true" duration-ms="15"
              started-
at="2007-05-
28T12:14:37Z" finished-
at="2007-05-
28T12:14:37Z">
        </test-
method>
      </class>
    </test>
  </suite>
```

```
</testng-
results>
```

Este reportero se inyecta junto con los otros oyentes predeterminados para que pueda obtener este tipo de salida de forma predeterminada. El oyente proporciona algunas propiedades que pueden modificar el reportero para que se ajuste a sus necesidades. La siguiente tabla contiene una lista de estas propiedades con una breve explicación:

Propiedad	Comentario	Valor por defecto
outputDirectory	Una `cadena` que indica el directorio donde se deben generar los archivos XML.	El directorio de salida TestNG
timestampFormat	Especifica el formato de los campos de fecha que genera este reportero	aaaa-MM-dd'T'HH:mm:ss'Z'

fileFragmentationLevel	Un número entero con los valores 1, 2 o 3, que indica la forma en que se generan los archivos XML:
	`1 - generará todos los resultados en un archivo.`
	`2 - cada suite se genera en un archivo XML separado que está vinculado al archivo principal.`
	`3: igual que 2 más archivos separados para casos de prueba a los que se hace referencia desde los archivos de la suite.`
		1

splitClassAndPackageNames	Este booleano especifica la forma en que se generan los nombres de clase para el elemento `<class>` . Por ejemplo, obtendrá `<class class="com.test.MyTest">` para falso y `<class class="MyTest" package="com.test">` para verdadero.	falso
generateGroupsAttribute	Un valor booleano que indica si se debe generar un atributo de `grupos para el elemento <test-method>` . Esta función tiene como objetivo proporcionar un método directo para recuperar los grupos que incluyen un método de prueba sin tener que navegar por los elementos `<group>` .	falso
generateTestResultAttributes	Un valor booleano que indica si se debe generar una etiqueta `<attributes> para cada elemento <test-method>` , que contiene los atributos del resultado de la prueba (consulte `ITestResult.setAttribute()` sobre la configuración de los atributos del resultado de la prueba). Cada representación de atributo `toString()` se escribirá en una etiqueta `<attribute name="[attribute name]">` .	falso
stackTraceOutputMethod	Especifica el tipo de seguimiento de pila que se generará para las excepciones y tiene los siguientes valores: `0: sin seguimiento de pila (solo clase de excepción y mensaje).`	2

	1: una versión corta del seguimiento de la pila que mantiene solo unas pocas líneas desde la parte superior 2 - el stacktrace completo con todas las excepciones internas 3 - stacktrace tanto corto como largo	
generateDependsOnMethods	Utilice este atributo para habilitar o deshabilitar la generación de un atributo que `depende de los métodos` para el elemento `<test-method>`.	verdadero
generateDependsOnGroups	Habilita/deshabilita la generación de un atributo `depende de grupos para el elemento <test-method>`.	verdadero

Para configurar este reportero puede usar la opción `-reporter` en la línea de comando o la tarea Ant con el elemento anidado `<reporter>`. Para cada uno de estos debe especificar la clase `org.testng.reporters.XMLReporter`. Tenga en cuenta que no puede configurar el reportero incorporado porque este solo usará la configuración predeterminada. Si solo necesita el informe XML con configuraciones personalizadas, deberá agregarlo manualmente con uno de los dos métodos y deshabilitar los oyentes predeterminados.

6.2.6 - Códigos de salida de TestNG

Cuando TestNG completa la ejecución, sale con un código de retorno.
Este código de retorno se puede inspeccionar para tener una idea de la naturaleza de las fallas (si las hubo).
La siguiente tabla resume los diferentes códigos de salida que utiliza actualmente TestNG.

FailedWithinSuccess	omitido	Ha fallado	Código de estado	Observaciones
No	No	No	0	Pruebas aprobadas
No	No	Sí	1	Pruebas fallidas
No	Sí	No	2	Pruebas salteadas
No	Sí	Sí	3	Pruebas saltadas/fallidas
Sí	No	No	4	Pruebas FailedWithinSuccess
Sí	No	Sí	5	FailedWithinSuccess/pruebas fallidas
Sí	Sí	No	6	FailedWithinSuccess/Pruebas omitidas
Sí	Sí	Sí	7	FailedWithinSuccess/Skipped/Failed tests

7 - YAML

TestNG admite YAML como una forma alternativa de especificar su archivo de suite. Por ejemplo, el siguiente archivo XML:

```
<suite name="SingleSuite" verbose="2" thread-
count="4">

  <parameter name="n" value="42" />

  <test name="Regression2">
```

```
    <groups>
      <run>
        <exclude name="broken" />
      </run>
    </groups>

    <classes>
      <class name="test.listeners.ResultEndMillisTest" />
    </classes>
  </test>
</suite>
```

y aquí está su versión YAML:

```
name: SingleSuite

threadCount: 4

parameters: { n: 42 }


tests:
  - name: Regression2

    parameters: { count: 10 }

    excludedGroups: [ broken ]


    classes:

      -
test.listeners.ResultEndMillisTest
```

Aquí está el archivo de la suite de TestNG y su contraparte de YAML .

Es posible que encuentre el formato de archivo YAML más fácil de leer y mantener. Los archivos YAML también son reconocidos por el complemento TestNG Eclipse. Puede encontrar más información sobre YAML y TestNG en esta publicación de blog .

Nota:
TestNG de forma predeterminada no incluye la biblioteca relacionada con YAML en su classpath. Entonces, dependiendo de su sistema de compilación (Gradle/Maven), debe agregar una referencia explícita a la biblioteca YAML en su archivo de compilación.
Por ejemplo, si estuviera usando Maven, necesitaría agregar una dependencia como se muestra a continuación en su archivo `pom.xml` :

```
<dependency>
  <groupid>org.yaml</groupid>
```

```
    <artifactid>snakeyaml</artifactid>
    <version>1.23</version>
  </dependency>
```

O si estuviera usando Gradle, agregaría una dependencia como se muestra a continuación en su archivo `build.gradle` :

```
compile
group:
'org.yaml',
name:
'snakeyaml',
version:
'1.23'
```

8 - Prueba en seco para tus pruebas

Cuando se inicia en modo de ejecución en seco, TestNG mostrará una lista de los métodos de prueba que se invocarían pero sin llamarlos realmente.

Puede habilitar el modo de ejecución en seco para TestNG pasando el argumento JVM – `Dtestng.mode.dryrun=true`

9 - Argumentos JVM en TestNG

Argumento JVM	Comentario	Valor por defecto
`testng.thread.affinity`	Un `valor booleano` que indica si TestNG debe recurrir a la ejecución de métodos dependientes en el mismo subproceso que los métodos ascendentes.	`falso`
`testng.modo.dryrun`	Un `valor booleano` que indica si TestNG debe simular una ejecución real. En este modo, los métodos de prueba no se ejecutan realmente.	`falso`
`testng.test.classpath`	Una `cadena` que representa una lista de archivos zip o jar que deben agregarse a la ruta de clases de TestNG para que recupere las clases de prueba para su ejecución.	""
`skip.caller.clsLoader`	Un `valor booleano` que indica si TestNG debe omitir el uso del ClassLoader actual para cargar clases.	`falso`
`testng.dtd.http`	Un `valor booleano` que indica si TestNG debe cargar DTD desde puntos finales `http` .	`falso`
`testng.show.stack.frames`	Un `valor booleano` que indica si TestNG debe mostrar seguimientos de pila detallados en los informes.	`falso`

Argumento JVM	Comentario	Valor por defecto
`testng.memoria.amigable`	Un `valor booleano` que indica si TestNG debe reconocer la memoria y utilizar representaciones de métodos de prueba ligeros.	`falso`
`testng.strict.parallel`	Un `booleano` que indica que TestNG debe intentar iniciar todos los métodos de prueba simultáneamente cuando hay más de una etiqueta de `prueba` y se ha establecido el paralelismo en los `métodos` .	`falso`
`emailable.report2.name`	Una `cadena` que indica el nombre del archivo en el que se escribirán los informes que se pueden enviar por correo electrónico.	`emailable-informe.html`
`oldTestngEmailableReporter`	Un valor `booleano` que indica si TestNG debe usar el antiguo detector de informes que se puede enviar por correo electrónico para crear un informe html simple que se puede enviar por correo electrónico.	`falso`
`noEmailableReporter`	Un valor `booleano` que indica si TestNG debe usar el NUEVO detector de informes que se puede enviar por correo electrónico para crear un informe html simple que se puede enviar por correo electrónico.	`verdadero`
`testng.report.xml.name`	Una `cadena` que indica el nombre del archivo en el que se escribirán los informes xml.	`testng-resultados.xml`
`fileStringBuffer`	Un `booleano` que indica si TestNG debe generar registros detallados cuando se trabaja con datos de texto muy grandes.	`falso`
`stacktrace.success.output.level`	Una `cadena` que indica los niveles de registro que se incluirán en los informes XML (los valores válidos incluyen: `NINGUNO` (sin stacktraces), `SHORT` (stacktrace corto), `FULL` (stacktrace completo), `AMBOS` (stacktrace corto y completo).	`LLENO`

Volver a mi página de inicio .

O echa un vistazo a algunos de mis otros proyectos:

- EJBGen : un generador de etiquetas EJB.
- TestNG : un marco de prueba que utiliza anotaciones, grupos de prueba y parámetros de método.

- Doclipse : un complemento de Eclipse de etiqueta JavaDoc.
- J15 : un complemento de Eclipse para ayudarlo a migrar su código a las nuevas construcciones JDK 1.5.
- SGen : un reemplazo para XDoclet con una arquitectura de complemento fácil.
- Canvas : un generador de plantillas basado en el lenguaje Groovy.

10 - Integración del marco de registro en TestNG

A partir de la versión **7.5** de TestNG, TestNG utiliza la fachada de registro proporcionada por Slf4j.
TestNG por defecto no trae ninguna implementación de fachada Slf4j explícita.
Para controlar los registros que emiten los internos de TestNG, agregue una dependencia en cualquier implementación Slf4j adecuada (implementación *nativa o envuelta*) desde aquí