Why drink it?

"We decided during the main conference that we should use JUnit 4 and Mockito because we think they are the future of TDD and mocking in Java"

Dan North, the originator of BDD

Mocking

- Un objeto mock es un sustituto de un objeto real del programa, es “un fake”
- Se simulan partes del sistema con las que el objeto a testar tiene alguna dependencia
- Un mock usa la interfaz de un objeto real y la implementa definiendo su comportamiento
- Se sustituye la instancia del objeto que en realidad se esperaba por el objeto falseado
- La clase testeable no sabe si interacciona con un objeto de identidad real
- Se pretende comprobar que el código funciona independientemente de sus dependencias

Mockito

- Basado en EasyMock
- Mejora la Api a nivel sintáctico
- Permite crear mocks de interfaces y de clases concretas
- Verificación en las invocaciones a los métodos
- Se centra en las interacciones de las clases a probar

Para trabajar con mockito:

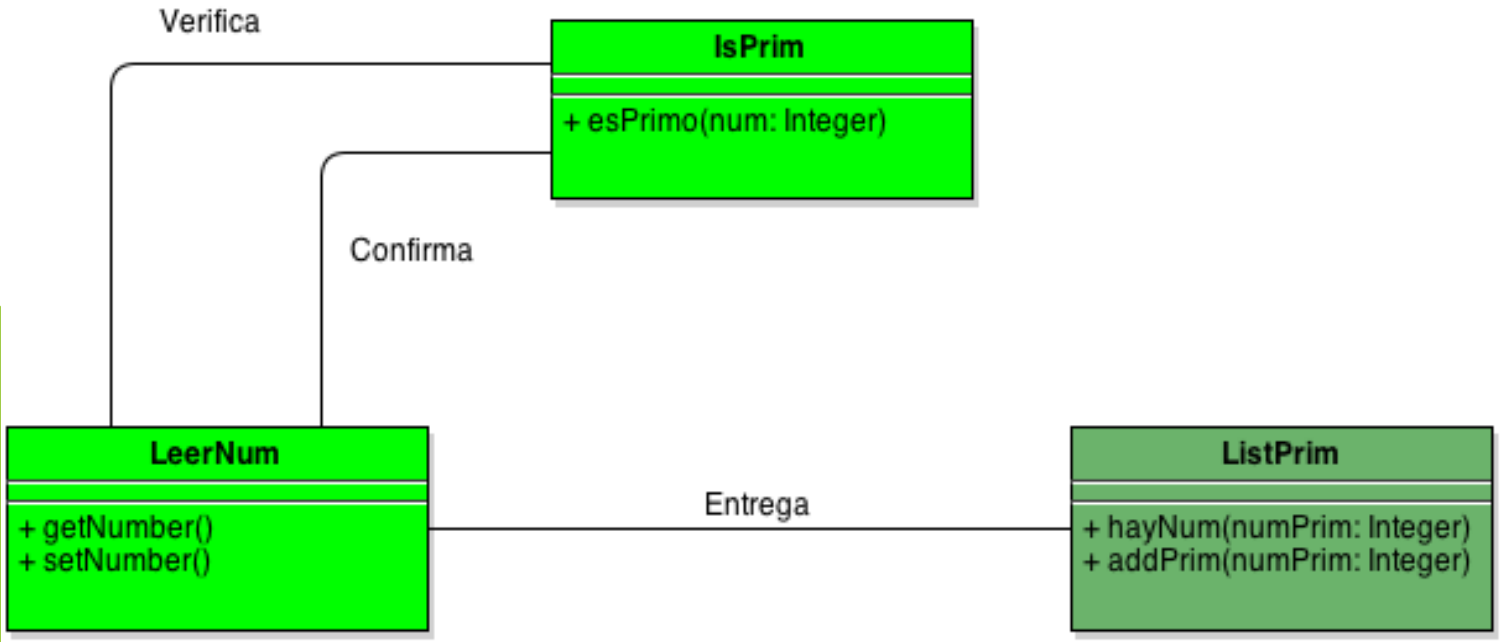
http://code.google.com/p/mockito/downloads/detail?name=mockito-all-1.9.0.jar

Etapas


```
public class MockitoTest{

	@Test
	public void probarAlgo() {
		Entidad entidad = mock(Entidad.class);
		AlgunaClase objetoAProbar= new AlgunaClase(entidad);

		objetoAProbar.ejecutaMetodo ();

		verify(entidad).hacerAlgoConcreto();
	}
}
```

List<Prim>

- Queremos memorizar números primos
- El usuario introduce un número y se verifica si es primo
- Si se confirma que es un número primo se guarda en una lista
- Si el número ya existe en la lista, se ignora



Test

- Tenemos acceso únicamente a la interfaz de la clase **ListPrimos:**
  - No nos importa
  - Pero tenemos que probar nuestra clase que la usará

```
public interface ListPrimos {

   void addPrim(int numPrim);

   boolean hayNum(int numPrim);

}
```

Mockito in java

- Comprobamos que se ha añadido correctamente en la lista un número primo proporcionado

```
public class Test{

   @Test
   public void setPrimTest() {
   // Inicializar
   ListPrimos listprim = mock(ListPrimos.class);
   when (listprim.hayNum(83)).thenReturn(false);

   // Ejecutar
   LeerNum leerNum = new LeerNum();
   int num = LeerNum.getNumber();
   leerNum.addPrim(num);

   // Comprobar
   System.out.print(verify(ListPrimos).addPrim(83));
}
```

Clase a probar

```
public class LeerNum {

    Private ArrayList<Integer> listaPrimos;

    public LeerNum(){
        listaPrimos = new ArrayList<Integer>();
    }

    public int getNumber() throws IOException{
      int Num = 0;
      InputStreamReader entrada = new InputStreamReader(System.in);
      BufferedReader R = new BufferedReader(entrada);
      System.out.print("Introduzca N: ");
      try {
            Num = Integer.parseInt(R.readLine());
      } catch (NumberFormatException e) {
         throw new IOException("Entrada Incorrecta.");
      }

       return Num;
    }

    public void setNumber(int numero){
        listaPrimos.add(numero);
    }
}
```

Pruebas unitarias con mockito

- Podemos simular un objeto

```
LeerNum mockedPrim = mock(LeerNum.class);
```

- Tenemos las mismas posibilidades que un objeto real (llamada a métodos)
- Desventaja: En la llamada al método, éste no devuelve ningun valor en caso de tener retorno a no ser que se lo indiquemos con when, solo testea lo que hay dentro del método.
- Por eso decidimos testear un objeto real con mock, para ello usamos spy()
- Inicializamos mock

```
LeerNum mockedPrim = spy(new LeerNum());
```

- Llamamos al método y devolvemos valor

```
int num = mockedPrim.getNumber();
```

- Verifica la función

```
verify(mockedPrim).getNumber();
```

- Si hubiera algún error mockito nos lo muestra en la consola
- Que ocurre si comentamos la linea mockedPrim.getNumber() mockito nos muestra la verificación:

```
int num = 0; // mockedPrim.getNumber();
```

```
Exception in thread "main" Wanted but not invoked:
LeerNum.getNumber();
 -> at Prim.main(Prim.java:12)
 Actually, there were zero interactions with this mock
```

Más mockito

- Añadir un número primo a la lista y verificarlo:
  - mockedPrim.setNumber(83);
  - **verify**(mockedPrim).setNumber(83);
- Modificar el resultado de retorno de una función:
  - **when**(mockedPrim.getNumber()).thenReturn(null);
  - **verify**(mockedPrim).getNumber();
- Reacciones ante acciones
  - **stub**(mockedPrim.getNumber()).toReturn(5).toThrow(new IOException());

    Imprime en pantalla 5 al llamar al método:
  - System.out.println(mockedPrim.getNumber());

    Genera excepción en la segunda llamada:
  - mockedPrim.getNumber();
- De otra forma podemos realizar la misma acción
  - **doThrow**(new IOException()).when(mockedPrim).getNumber();
- Podemos verificar el número de llamadas a los métodos y desde donde:
  - verifyNoMoreInteractions(mockedPrim);

- Verificación en orden de objetos:
  - LeerNum primerMock = **spy**(new LeerNum());
  - LeerNum segundoMock = **spy**(new LeerNum());
  - primerMock.setNumber(13);
  - SegundoMock.setNumber(17);
  - InOrder inOrder = inOrder(primerMock, segundoMock);
  - inOrder.**verify**(primerMock).setNumber(17);
  - inOrder.**verify**(segundoMock).setNumber(13);

```
Exception in thread "main" Argument(s) are different! Wanted:
leerNum.setNumber(17);
-> at Prim.main(Prim.java:37)
Actual invocation has different arguments:
leerNum.setNumber(13);
-> at Prim.main(Prim.java:34)
at Prim.main(Prim.java:37)
```

- Verificación de iteracciones:
  - **verify**(mockedPrim, times(1)).setNumber(19);
  - **verify**(mockedPrim, never()).setNumber(23);
  - **verify**(mockedPrim, atLeastOnce()).setNumber(23);

```
Exception in thread "main" Wanted but not invoked:
leerNum.setNumber(23);
-> at Prim.main(Prim.java:42)
```

  - **verifyZeroInteractions**(mockPrim);

Metodos como parámetros

```
verify(mockedPrim).addPrim(anyInt(), anyString(), …);
```

anyBoolean, anyByte, anyChar, anyCollection, anyDouble, anyFloat, anyInt, anyList, anyLong, anyMap, anyObject, anyShort, anyString, argThat, booleanThat, byteThat, charThat, contains, doubleThat, endsWith, eq, floatThat, intThat, isA, isNotNull, isNull, longThat, matches, notNull, refEq, same, shortThat, startsWith

http://mockito.googlecode.com/svn/branches/1.5/javadoc/org/mockito/Mockito.html

Api Mockito

static VerificationMode	**atLeastOnce**() Allows at-least-once verification.
static Stubber	**doAnswer**(Answer answer) Use doAnswer() when you want to stub a void method with generic Answer.
static Stubber	**doNothing**() Use doNothing() for setting void methods to do nothing.
static Stubber	**doReturn**(java.lang.Object toBeReturned) Use doReturn() in those rare occasions when you cannot use stub(Object).
static Stubber	**doThrow**(java.lang.Throwable toBeThrown) Use doThrow() when you want to stub the void method with an exception.
static InOrder	**inOrder**(java.lang.Object... mocks) Creates InOrder object that allows verifying mocks in order.
static <T> T	**mock**(java.lang.Class<T> classToMock) Creates mock object of given class or interface.
static <T> T	**mock**(java.lang.Class<T> classToMock, java.lang.String name) Creates mock with a name.
static VerificationMode	**never**() Alias to times(0), see times(int)
static <T> T	**spy**(T object) Creates a spy of the real object.
static <T> OngoingStubbing<T>	**stub**(T methodCall) Stubs with return value or exception.
static <T> VoidMethodStubbable<T>	**stubVoid**(T mock) **Deprecated.** *Use doThrow(Throwable) method for stubbing voids*
static VerificationMode	**times**(int wantedNumberOfInvocations) Allows verifying exact number of invocations.
static <T> T	**verify**(T mock) Verifies certain behavior **happened once**
static <T> T	**verify**(T mock, VerificationMode mode) Verifies certain behavior happened at least once / exact number of times / never.
static void	**verifyNoMoreInteractions**(java.lang.Object... mocks) Throws an AssertionError if any of given mocks has any unverified interaction.
static void	**verifyZeroInteractions**(java.lang.Object... mocks) Verifies that no interactions happened on given mocks.

Conclusiones

- El uso de mock objects nos facilita mucho hacer test unitarios, permitiendo la detección de errores y asegurándonos el buen funcionamiento durante el futuro.

- ¿Por qué perder el tiempo usando mock objects cuando puedo realizar las pruebas apoyandome en otras clases que ya han sido probadas, y funcionan bien?

-Nunca se pierde tiempo en generar test ni en usar mock objects, puesto que ese código nos automatizará las tareas de pruebas nos sólo durante el desarrollo, si no tambien durante las fases de mantenimiento de la aplicación.

- Pensar que el código que hoy no falla no puede fallar mañana es erróneo también.

Autores: Javier Partido Rufo y Abdelmounaim Kastit