Geeks México

BLOG DE PROGRAMACIÓN EN ESPAÑOL SOBRE JAVA, FRAMEWORKS, BASES DE DATOS, CÓMPUTO EN LA NUBE, ETC. EN ESPAÑOL Y EN INGLÉS.

Pruebas Unitarias Parte 1: ¿Cómo escribir

código “Testeable”?

HACE 1 MES DEJA UN COMENTARIO

Rate This

Esta es la 1era parte de una serie de posts sobre tests unitarios. Hablaremos sobre que es una prueba unitaria, como escribir código que sea fácil de probar y como codificar estas pruebas usando JUnit y Mockito.

Una prueba unitaria es un tipo de prueba de software que tiene como objetivo verificar el funcionamiento correcto de un componente individual de una aplicación aislándolo del resto de su entorno. Para realmente probar un componente de manera individual, es necesario tener pleno control sobre su entorno (entradas y dependencias), ya que de no hacerlo, estaremos probando el componente más sus dependencias, lo que hará más difícil identificar la causa raíz de algún defecto ya que no sabremos si el problema se encuentra en el componente que estamos probando o en alguna de sus dependencias. Hacer esto puede parecer complicado debido a que la mayoría de los componentes que tenemos en nuestras aplicaciones dependen de uno o más componentes para su funcionamiento (por ejemplo, un servicio puede depender de un repositorio para obtener información de una base de datos). Sin embargo, si desarrollamos nuestros componentes con la idea de que deben ser “testeables” desde un inicio, esto será muy sencillo.

Un componente puede considerarse “testeable” si:

1. Sigue el Principio de Responsabilidad Única
2. Está correctamente encapsulado y tiene una interfaz pública bien definida
3. Se puede aislar fácilmente del resto del sistema
4. Sus salidas son observables, es decir, se puede observar fácilmente los resultados de una ejecución
5. Tiene un comportamiento determinístico (ejecutar un mismo método varias veces con las mismas entradas siempre dará el mismo resultado)

El siguiente código es un ejemplo de una clase **NO** “testeable”.

```
1  public class PayrollServiceImpl {
2
3      private EmployeeRepositoryImpl employe
4      private NotificationServiceImpl notifi
5
6      public PayrollServiceImpl() {
7          this.employeeRepository = new Empl
8          this.notificationService = new Not
9      }
10
11     public void increaseSalary(int employe
12         Employee employee = this.employeeR
13
14         // only employees with more than 5
15         if (employee.getYearsOfExperience(
16             employee.setSalary(employee.ge
17             this.employeeRepository.save(e
18
19             // notify employee about the r
20             this.notificationService.notif
21         }
22     }
23
24 }
```

Esta clase tiene varios problemas. En primer lugar, no se puede aislar fácilmente del resto del sistema ya que su constructor crea las dependencias explícitamente. En segundo lugar, las dependencias son clases concretas (no interfaces), lo que aumenta el acomplamiento de

nuestra clase y hace más difícil controlar el comportamiento de las dependencias. Finalmente, no hay forma de observar cuál fue el resultado de llamar al método increaseSalary.

Una más “testeable” del código se muestra a continuación:

```
1  public class PayrollServiceImpl {
2
3      private EmployeeRepository employeeRep
4      private NotificationService notificati
5
6      private static final int NOT_ENOUGH_EX
7      private static final int SALARY_INCREA
8
9      public PayrollServiceImpl(EmployeeRepo
10         this.employeeRepository = employee
11         this.notificationService = notifi
12     }
13
14     public int increaseSalary(int employee
15         Employee employee = this.employeeR
16         // only employees with more than 5
17         if (employee.getYearsOfExperience(
18             employee.setSalary(employee.ge
19             this.employeeRepository.save(e
20
21             // notify employee about the r
22             this.notificationService.notif
23
24             return SALARY_INCREASED;
25         } else {
26             return NOT_ENOUGH_EXPERIENCE;
27         }
28     }
29
30 }
```

Los problemas de la versión anterior de la clase PayrollServiceImpl se corrigieron de la siguiente manera:

1. El constructor de la clase ahora recibe sus dependencias como parámetros
2. Las dependencias son ahora interfaces en lugar de implementaciones concretas de las clases. Lo que facilita la creación de “mocks”.

3. El método increaseSalary ahora devuelve códigos de estado para reflejar cuál fue el resultado de la ejecución

Los cambios realizados en la clase PayrollServiceImpl tienen varios beneficios: ahora podemos controlar cuáles son las dependencias de esta clase en nuestras pruebas unitarias, lo que nos permitirá usar “mocks” para controlar el funcionamiento de estas dependencias durante la prueba. Por ejemplo, podríamos pasar un “mock” de la interfaz NotificationService al constructor de clase de PayrollServiceImpl al inicializar nuestra prueba unitaria para así verificar si el método notifySalaryIncreased fue llamado o no. De manera similar, podríamos hacer que el “mock” del EmployeeRepository arroje una excepción al ser invocado para simular el caso en el que se pasa un employeeId no existente como parámetro.

Como podemos observar, escribir código “testeable” requiere algo de disciplina y un esfuerzo extra a la hora del diseño y desarrollo de nuestros componentes. Sin embargo, como recompensa, tenemos un código más fácil de probar, limpio, fácil de mantener y débilmente acoplado. Escribiendo pruebas unitarias adecuadas para nuestros componentes podremos incrementar la confianza que tenemos en el correcto funcionamiento de nuestro sistema.

En el siguiente post explicaremos como escribir dichas pruebas unitarias usando JUnit y mockito.