DigitalHouse>

Certified Tech Developer
The Ultimate Degree

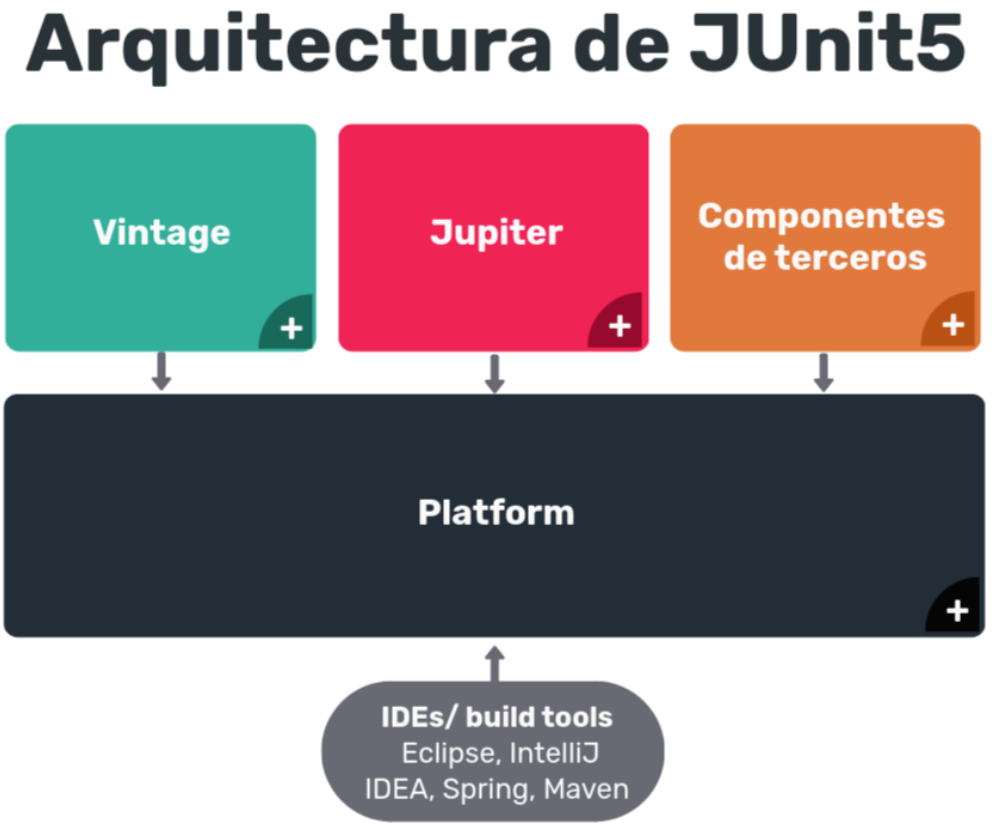
Back End I

JUnit

1. *¿QUÉ ES JUnit?*

JUnit *(http://junit.org) es el framework open source de testing para Java más utilizado. Nos permite escribir y ejecutar tests automatizados. Es soportado por todas las* ***IDEs (Eclipse, IntelliJ IDEA)****, build tools* ***(Maven, Gradle)*** *y por frameworks como* ***Spring****. Conozcamos su arquitectura.*

2. *ARQUITECTURA DE JUnit 5*

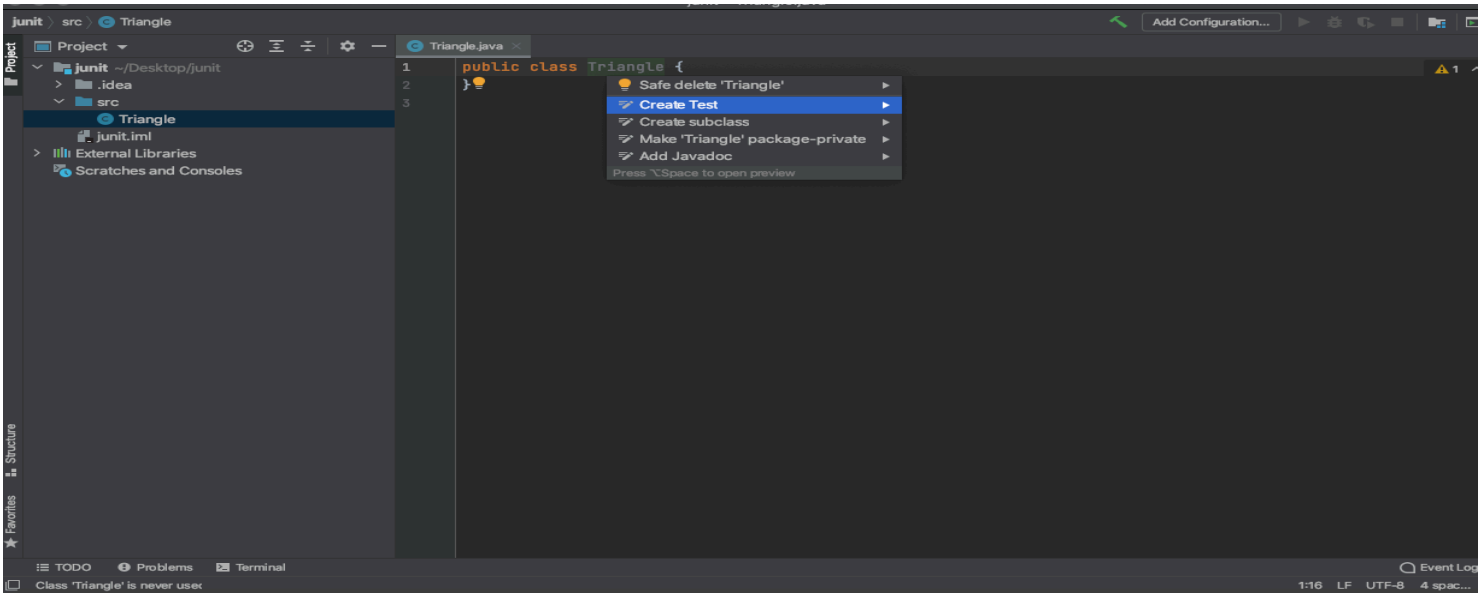


PRINCIPIO	*DESCRIPCIÓN*
VINTAGE	*Contiene el motor de JUnit 3 y 4 para correr tests escritos en estas versiones (Old test).*
JUPITER	*Es el componente que implementa el nuevo modelo de programación y extensión: API para escribir tests y motor para ejecutarlos (New tests).*
COMPONENTES DE TERCEROS	*La idea es que otros frameworks puedan ejecutar otros casos de prueba realizando una extensión de la plataforma.*
PLATFORM	*Es un componente que actúa de ejecutor genérico de los test. La platform-launcher es usada por las IDEs y los build tools.*

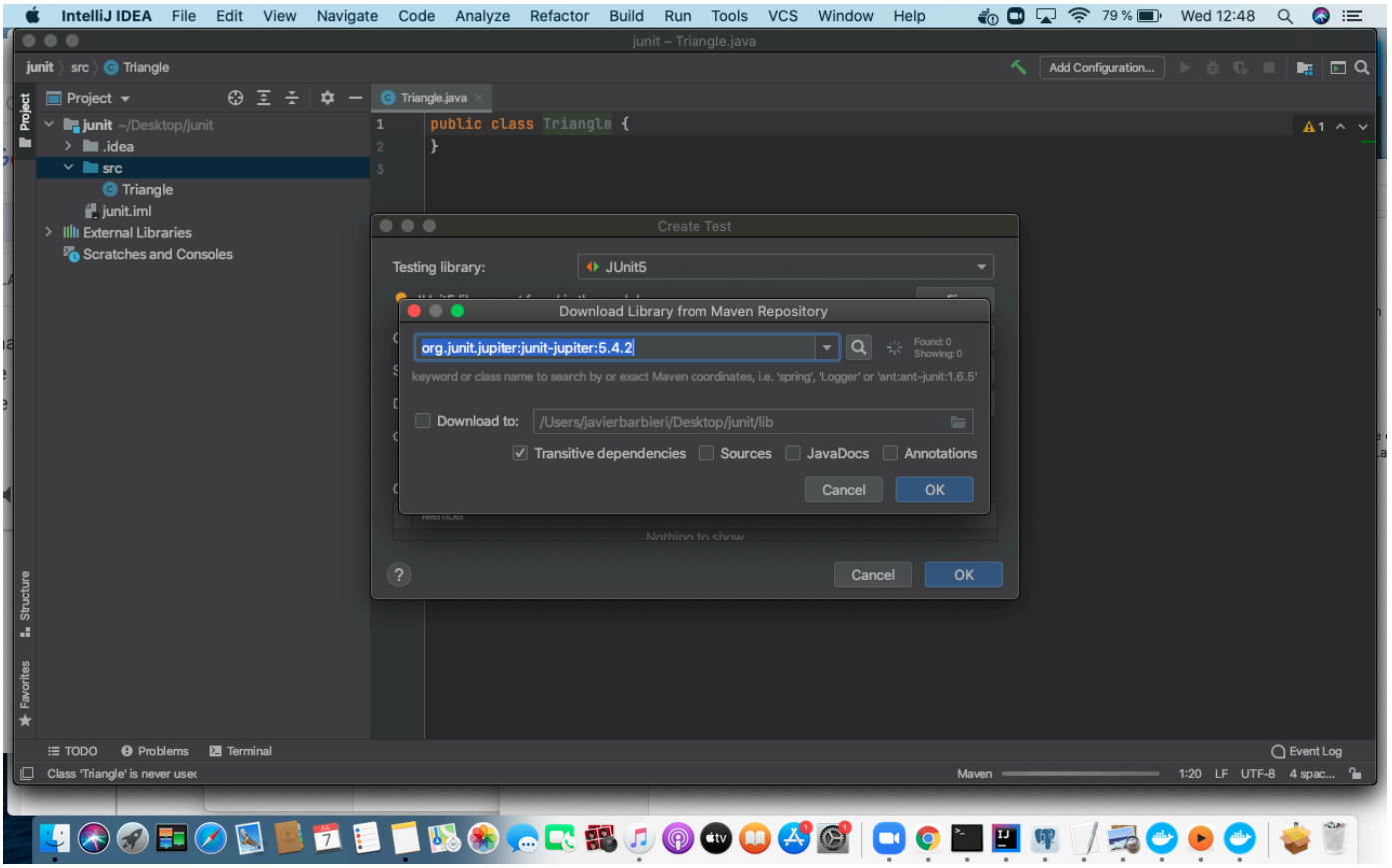
3. *¿CÓMO CONFIGURAMOS LA LIBRERÍA?*

Para configurar la librería de JUnit en nuestro entorno de desarrollo (IntelliJ IDEA) debemos seguir los siguientes pasos:

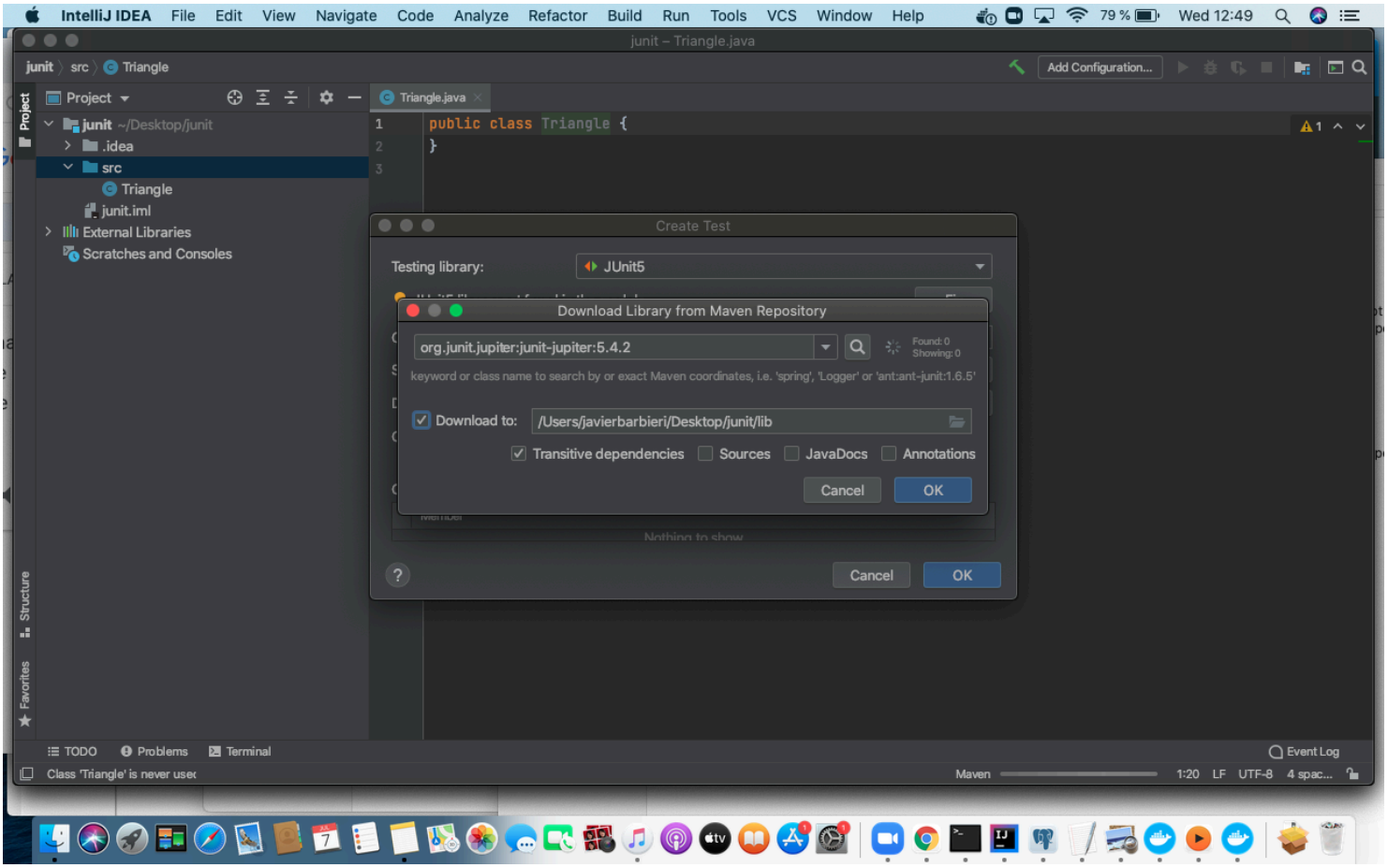
1) *Nos paramos sobre la clase que queremos testear, hacemos alt + Enter y hacemos click en Create Test.*



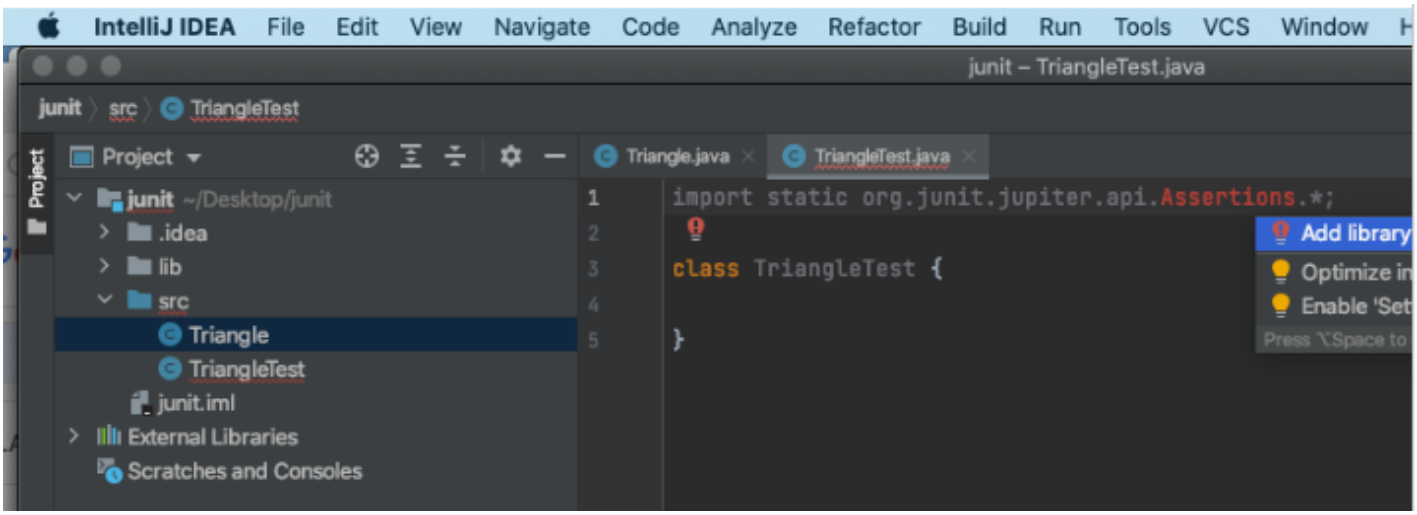
2) *Luego el IDE nos guiará automáticamente para agregar la librería de JUnit de la siguiente manera:*



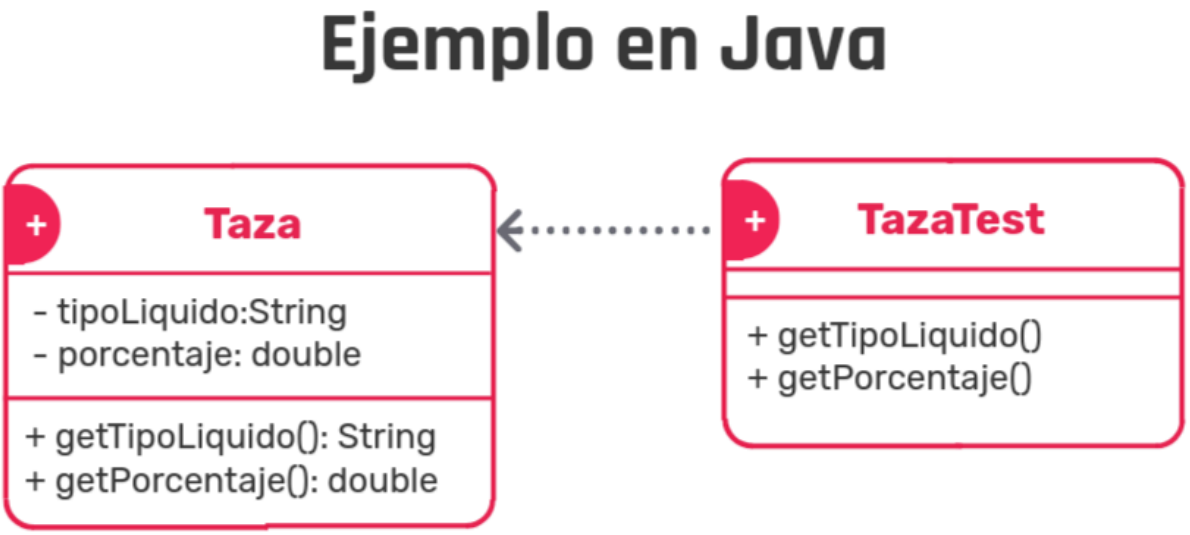
3) *A continuación, simplemente hacemos click sobre el botón OK y se descargará la librería en la ruta recomendada.*



4) *Como último paso debemos agregarla a lo que se denomina classpath. La manera más fácil es agregar el paquete, como vemos en la imagen, y en la recomendación del IDE aparece cómo agregarla.*



4. *EJEMPLO EN JAVA*

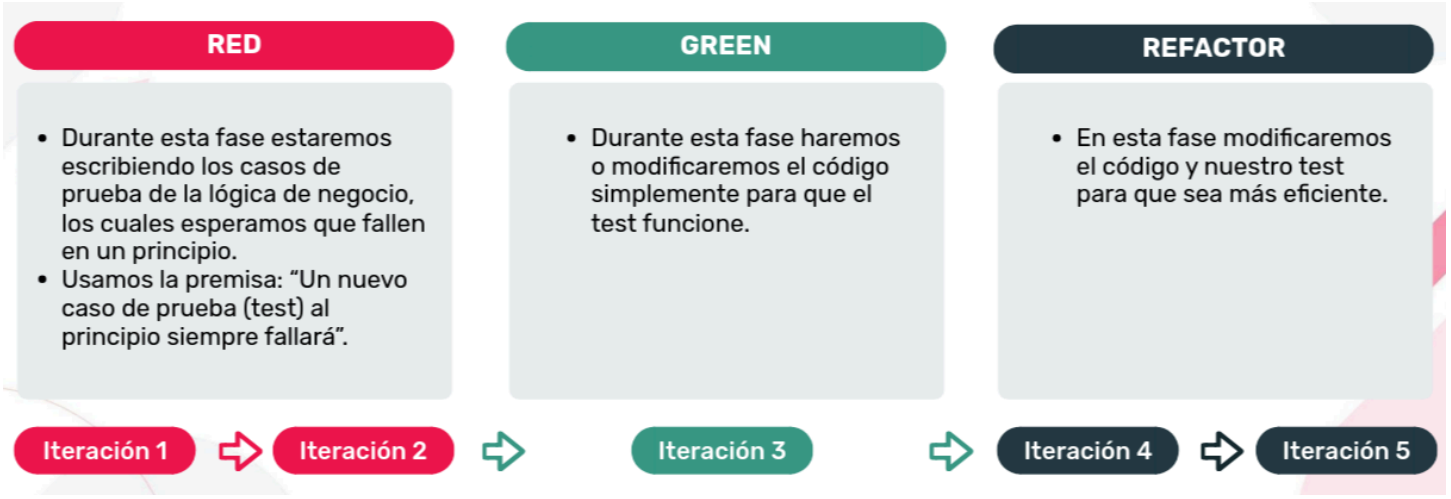



```java
public class Taza {
   private String tipoLiquido;
   private double porcentaje;
   public Taza(String tipoLiquido, double porcentaje) {
       this.tipoLiquido = tipoLiquido;
       this.porcentaje = porcentaje;
   }
   public String getTipoLiquido() {
       return tipoLiquido;
   }
   public void setTipoLiquido(String tipoLiquido)
       this.tipoLiquido = tipoLiquido;
   }
   public double getPorcentaje() {
       return porcentaje;
   }
   public void setPorcentaje(double porcentaje) {
       this.porcentaje = porcentaje;
   }
}
```

```java
import static org.junit.jupiter.api.Assertions.*;
class TazaTest {
   @Test
   void getTipoLiquido() {
      Taza taza = new Taza("Jugo de Naranja", 70.5);
      assertEquals("Jugo de Naranja", taza.getTipoLiquido());
   }
   @Test
   void getPorcentaje() {
      Taza taza = new Taza("Jugo de Naranja", 70.5);
      assertEquals(70.5, taza.getPorcentaje());
   }
}
```

5. *TDD - TEST DEVELOPMENT DRIVEN*

TDD *significa que usamos las pruebas (tests) para orientar o dirigir la forma en la que escribimos nuestro código. Normalmente, el flujo de trabajo se describe como* ***Red, Green, Refactor.***



Iteración 1

Se comienza con el testeo, al no existir todavía el método, falla por error de compilación.

Código del método a testear

No existe

Test

```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int resultado = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado == 1);
}
```

FALLA

Iteración 2

Se busca lograr que el caso al menos compile, para lo cual se programa el método que se desea probar con el objetivo de que simplemente pase la compilación, no necesariamente de que resuelva bien la necesidad de negocio.

Código del método a testear

```
public class Validador {
  public static int esPar(int numero) {
       return 0;
  }
}
```

Test

```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int resultado = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado == 1);
}
```

FALLA

Iteración 3

Hacemos lo estrictamente necesario para que el caso pase correctamente el test. En este caso, como siempre devuelve 1, el test pasará correctamente.

Código del método a testear

```
public class Validador {
  public static int esPar(int numero) {
       return 1;
  }
}
```

Test

```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int resultado = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado == 1);
}
```

CORRECTO

Iteración 4

En nuestro ejemplo, nos damos cuenta de que usando un tipo de dato booleano podemos ser más eficientes, por lo que hacemos las modificaciones necesarias para implementar esta mejora.

Código del método a testear

```
public class Validador {
  public static boolean esPar(int numero) {
       return true;
  }
}
```

Test

```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int boolean = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado);
}
```

CORRECTO

Iteración 5

Hacemos que el algoritmo para determinar la paridad sea cada vez más eficiente. Para este ejemplo, dado que todo número par al dividirlo por 2 siempre su resto da 0 (cero), podemos utilizar el operador % que nos devuelve el resto de una división.

Código del método a testear

```
public class Validador {
  public static boolean esPar(int numero) {
        return numero%2 == 0;
  }
}
```

Test

```
@Test
public void testEsPar(){
   //Dado
   int numero = 4;
   //Cuando
   int boolean = Validador.esPar(numero);
   //Entonces
   Assert.assertTrue(resultado);
}
```

CORRECTO