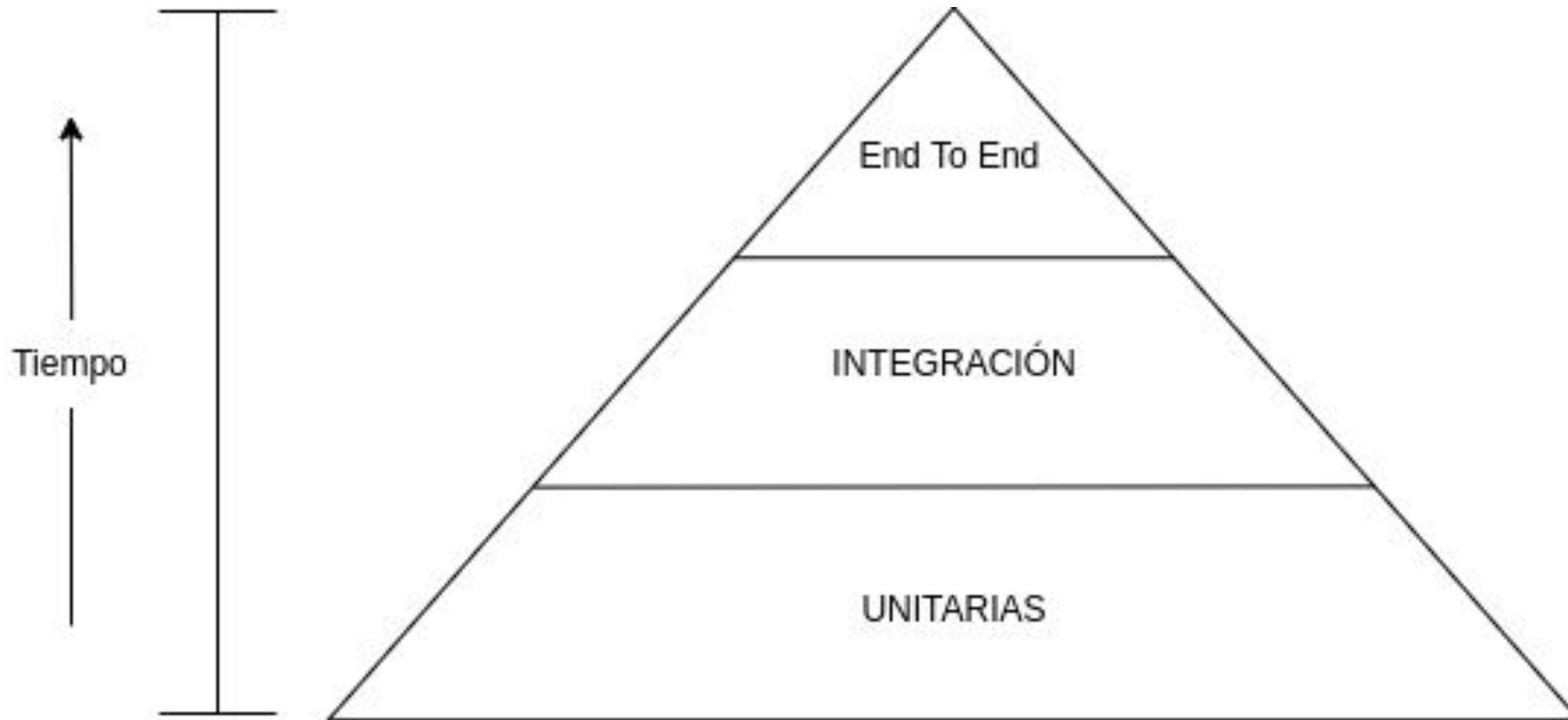


# Pruebas unitarias en Java

Esta presentación proporciona una visión general de las pruebas unitarias y su importancia en el desarrollo de software.



**Pirámide de Pruebas - Tipos de pruebas en el desarrollo**

# Automatización de pruebas unitarias

## ¿Qué es una prueba, para que las escribimos?

La prueba unitaria es una forma de prueba de caja blanca en la que los casos de prueba se basan en la estructura interna. El propósito de las pruebas unitarias es examinar los componentes individuales o piezas de métodos/clases para verificar la funcionalidad, asegurando que el comportamiento sea el esperado.

## ¿Qué es JUnit?

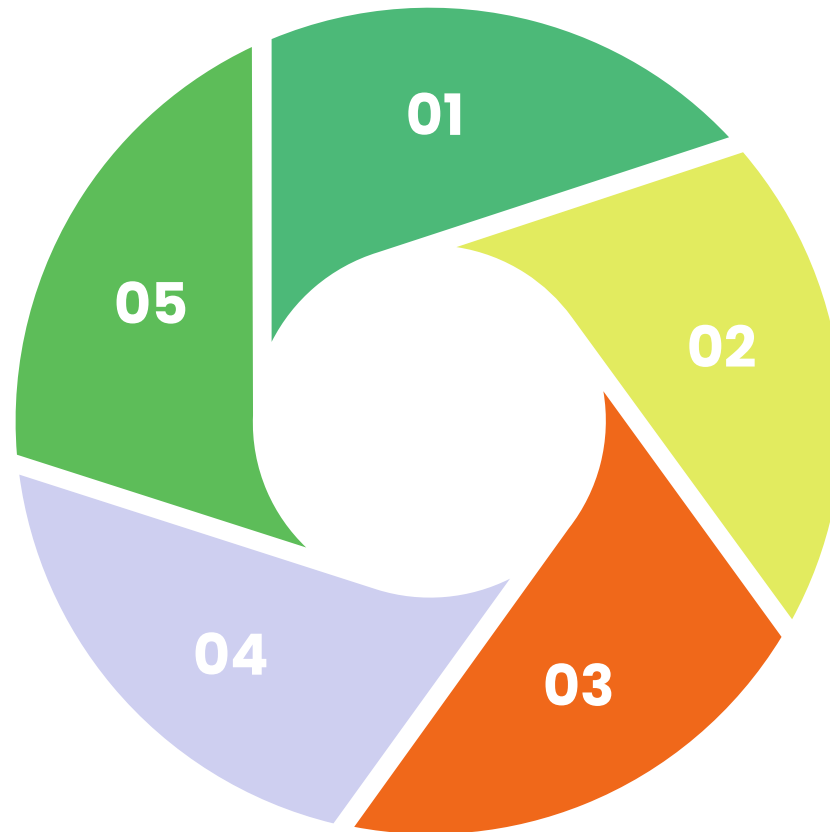
JUnit es el marco de pruebas unitarias de Java más popular. Un marco de código abierto, se utiliza para escribir y ejecutar pruebas automatizadas repetibles.



# Ventajas de las pruebas unitarias

Proporciona confianza en la refactorización y la realización de cambios en el código.

Sirve como documentación sobre cómo se espera que funcione el código.

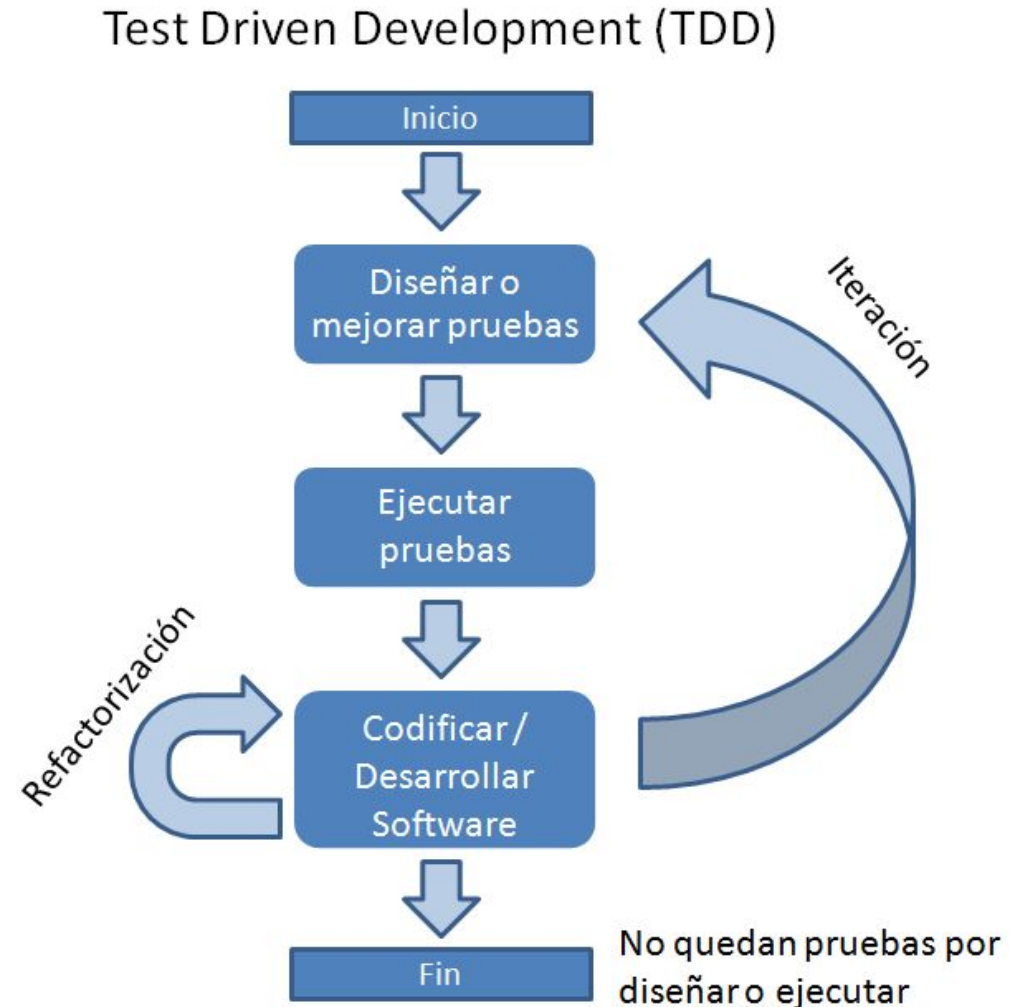


Detección temprana de errores y problemas en el código.

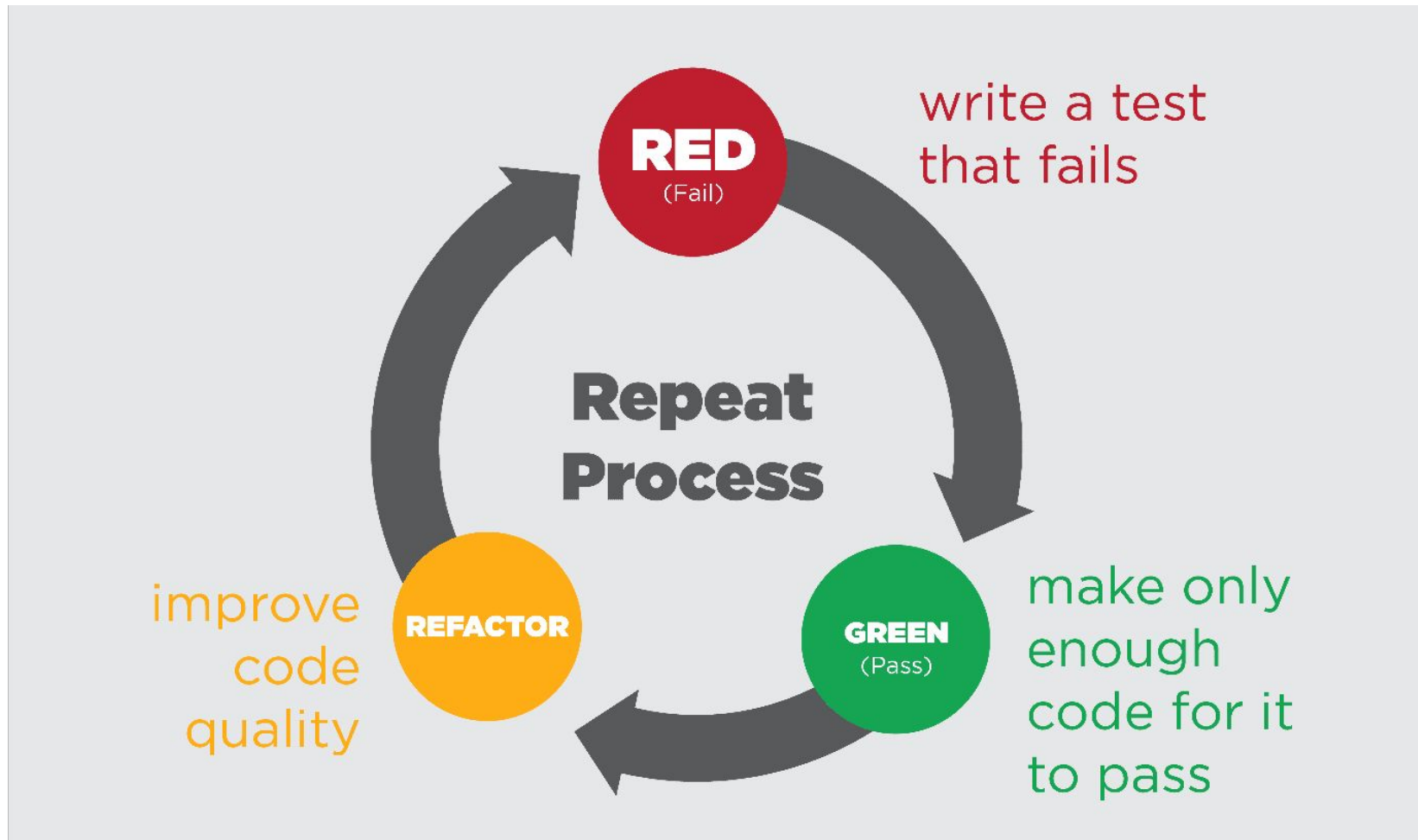
Ayuda a mantener la calidad y legibilidad del código.

Promueve las buenas prácticas de desarrollo de software y la modularidad.

# Buenas prácticas de desarrollo: Modelo TDD



## Estados del desarrollo de pruebas



# Prueba automática: Documentación

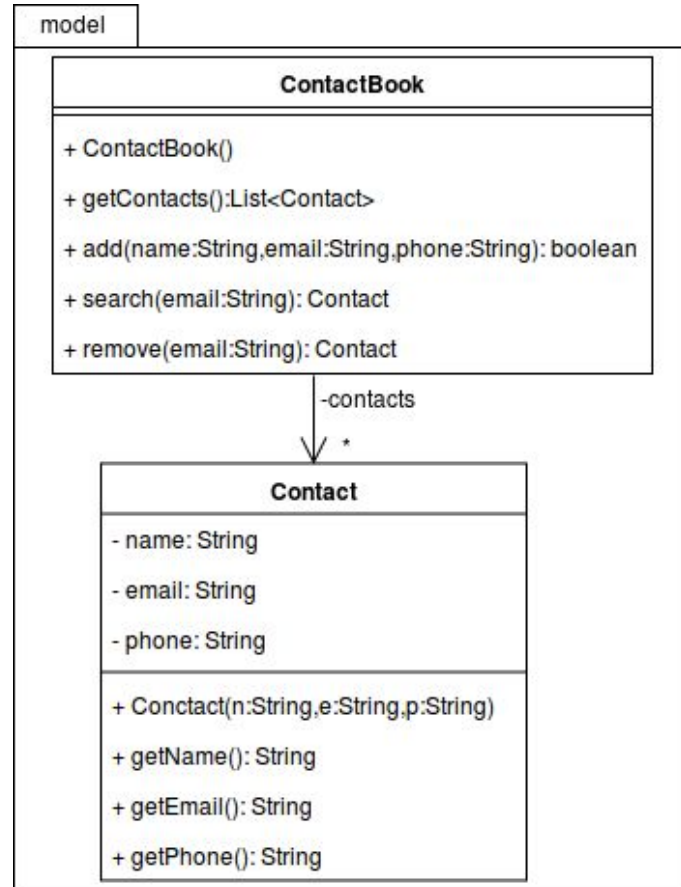
| Nombre   | Clase     | Escenario |
|----------|-----------|-----------|
| setUp1() | ClaseTest |           |
| setUp2() | ClaseTest |           |

| Prueba N° 1 | Objetivo: |           |            |           |
|-------------|-----------|-----------|------------|-----------|
| Clase       | Método    | Escenario | Entrada(s) | Salida(s) |
| ModelClass  | method()  | setUp1()  |            |           |
| Prueba N° n | Objetivo: |           |            |           |
| Clase       | Método    | Escenario | Entrada(s) | Salida(s) |
| ModelClass  | method()  |           |            |           |

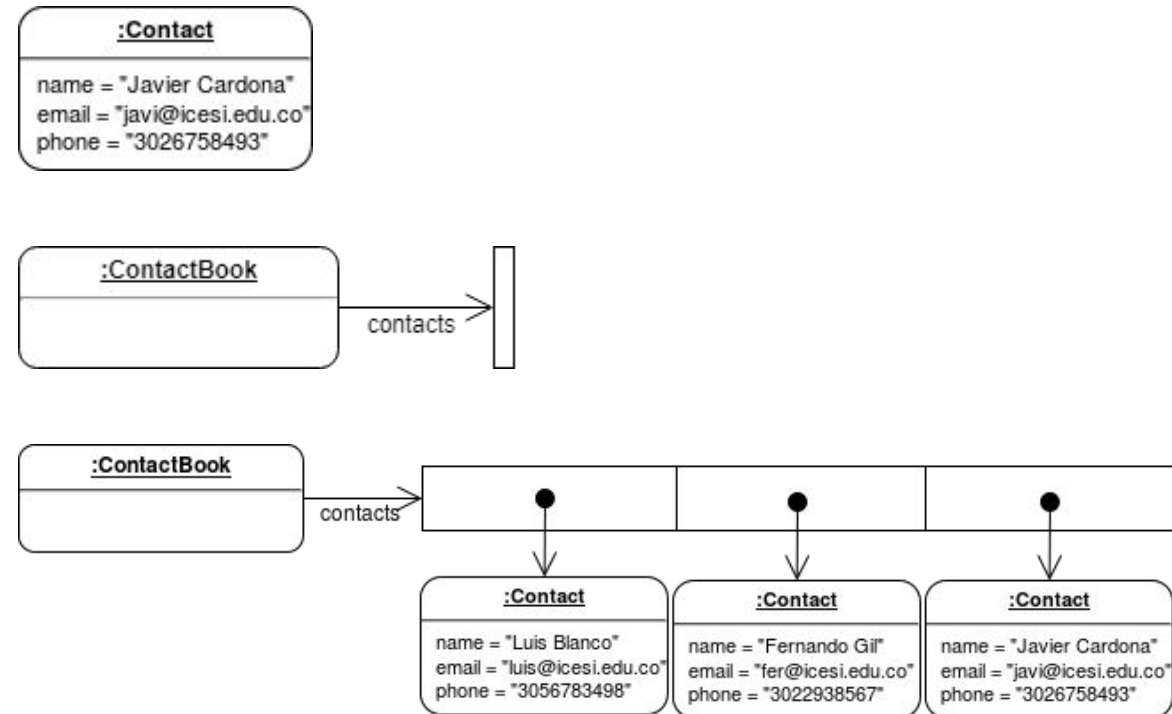
Formato escenarios – casos de pruebas

# Prueba automática: Documentación

Clase para probar



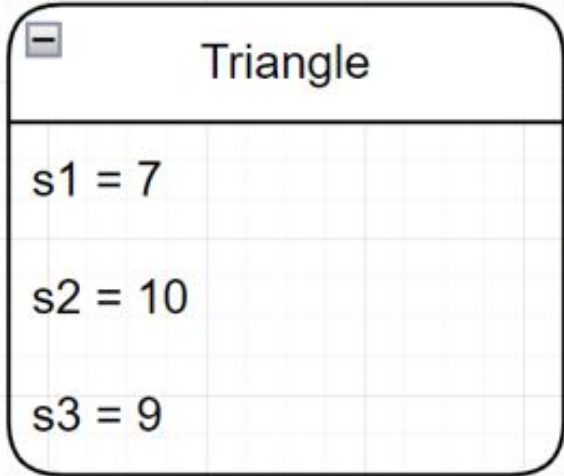
Algunos escenarios de prueba





## Ejemplo: Documentación de prueba - Escenarios

### Configuración de los Escenarios

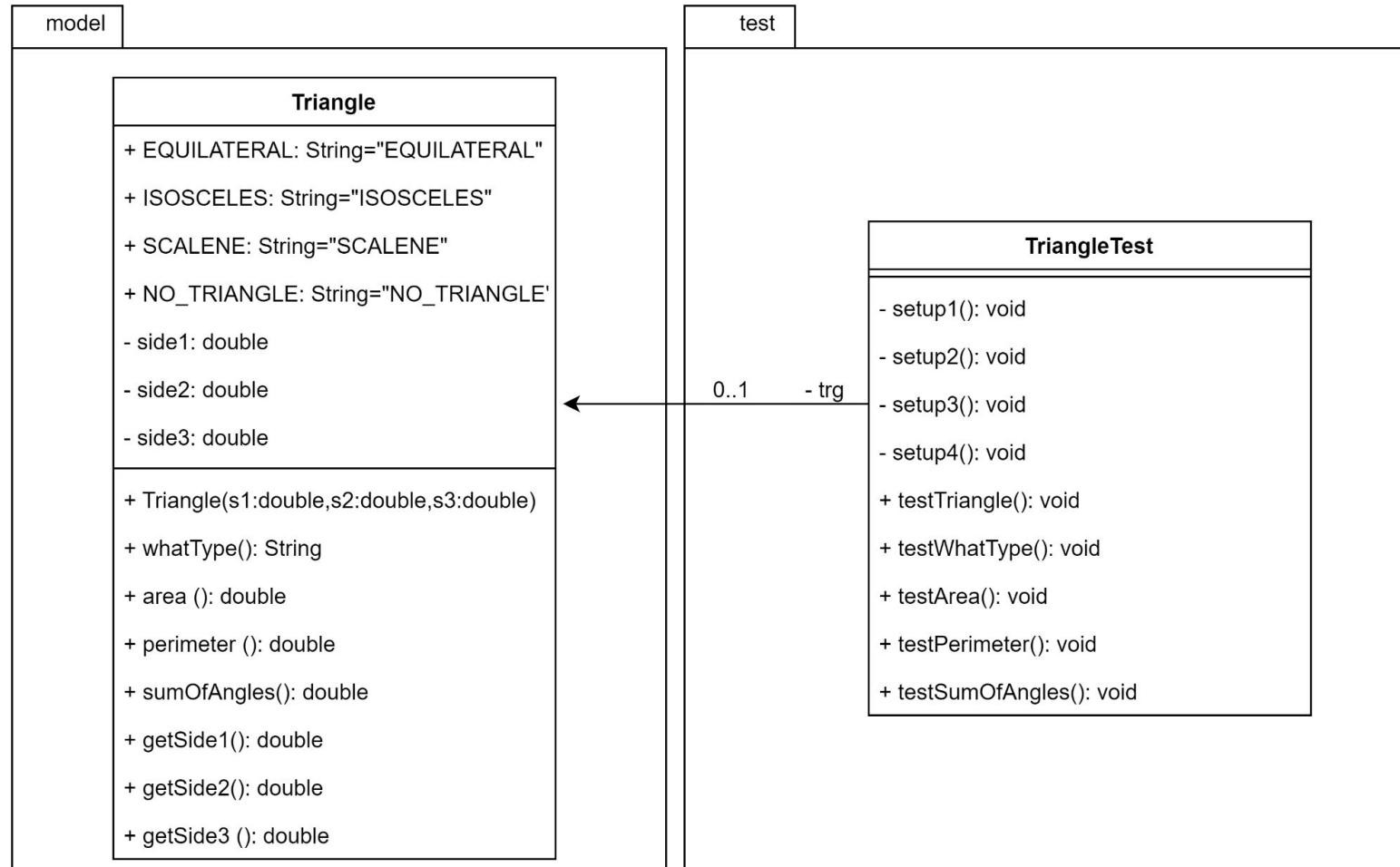
| Nombre   | Clase        | Escenario  |
|----------|--------------|--|
| setUp1() | TriangleTest |  <p>The diagram shows a rounded rectangle representing an object named 'Triangle'. It has a small square icon in the top-left corner. Below the name, there are three lines of text: 's1 = 7', 's2 = 10', and 's3 = 9'.</p> |

## Ejemplo: Documentación de prueba - Casos de prueba

### Diseño de Casos de Prueba

| Objetivo de la Prueba: Identificar el tipo de triangulo, según unos valores de lado. |                            |           |  |                                 |
|--|----------------------------|-----------|--|---------------------------------|
| Clase  | Método                     | Escenario | Valores de Entrada   | Resultado                       |
| Triangle   | <a href="#">whatType()</a> | setUp1()  | <ul style="list-style-type: none"><li>- S1: Un lado de tamaño 7</li><li>- S2: Un lado de tamaño 10</li><li>- S3: Un lado de tamaño 9</li></ul> | "El triángulo es un": Escaleno. |

## Ejemplo: Diagrama de clases (incluyendo las pruebas)



# Automatización de pruebas unitarias

Debido a la forma modular de JUnit 5 se usa MAVEN para importar todos los módulos y dependencias de JUnit. Si solo se necesitan módulos específicos, se pueden especificar grupos o artefactos individuales en su lugar.

Para agregar JUnit 5 a Maven, agregue lo siguiente a pom.xml:

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->  
  
<dependency>  
  
    <groupId>org.junit.jupiter</groupId>  
  
    <artifactId>junit-jupiter-api</artifactId>  
  
    <version>5.11.4</version>  
  
    <scope>test</scope>  
  
</dependency>
```

# Anatomía de una prueba automática

```
package examples.nbank;

public class Conversion {

    public double tempConversion (double temperature, String unit) {
        if (unit.equals("F"))
            return (temperature - 32) * (5.0/9.0);
        else
            return (temperature * (9.0/5.0)) + 32;
    }
}
```

# Anatomía de una prueba automática

```
package examples.nbank;

1import static org.junit.Assert.assertEquals;

2import org.junit.*;

3public class ConversionTest {

4    @Test
5    public void testTempConversion() throws Throwable {
        // Given
6        Conversion underTest = new Conversion();

        // When
7        double temperature = 80.0d;
        String unit = "";
8        double result = underTest.tempConversion(temperature, unit);

        // Then - assertions for result of method tempConversion(double, String)
9        assertEquals(176.0d, result, 0.0);
    }
}
```

# Anatomía de una prueba automática

**Piezas 1 y 2** Estas son importaciones para las clases y paquetes JUnit utilizados por la prueba unitaria.

**Parte 3** Esto define el inicio de nuestra clase de prueba. Lo importante a tener en cuenta aquí es la convención de nomenclatura utilizada para la clase, que es *Prueba de nombre de clase*.

**Parte 4** Anotaciones de JUnit, hay muchas otras anotaciones, pero algunas de las más comunes son las siguientes.

- @Before identifica un método que debe ejecutarse antes de cada método de prueba en la clase. Por lo general, se usa para actualizar o restablecer el estado necesario para que los métodos de prueba se ejecuten correctamente.
- @After identifica un método que debe ejecutarse después de cada método de prueba en la clase. Se puede utilizar para restablecer variables, eliminar archivos temporales, etc.
- @Ignore especifica que no se debe ejecutar un método de prueba.
- @BeforeClass identifica un método que debe ejecutarse una vez antes de ejecutar cualquier método de prueba.
- @AfterClass identifica un método que debe ejecutarse una vez después de ejecutar todos los métodos de prueba.

**Parte 5** Una vez más, tenga en cuenta la convención de nomenclatura *nombre del método de prueba*, Donde *nombreMétodo* es el nombre del método que se está probando en la clase bajo prueba.

**Parte 6** En el *Given* construimos una nueva instancia de la clase bajo prueba y la inicializamos según corresponda. Esto es necesario ya que el método de prueba necesita llamar al método bajo prueba para probarlo. En nuestro ejemplo, no se necesita ninguna otra inicialización más allá de instanciar la clase, pero en muchos casos, es posible que se deba realizar una configuración adicional, como inicializar objetos para pasar al constructor o llamar a métodos que configuran el estado de la clase bajo prueba.

# Anatomía de una prueba automática

**Parte 7** En el *When* La sección de la prueba incluye la inicialización de variables que deben pasarse al llamar al método que se está probando y luego llamar al método de prueba (parte 8). Las variables deben recibir valores significativos que hagan que la prueba ejerza las partes del método de prueba que nos interesan. Tenga en cuenta que si una variable es un objeto, se debe crear una instancia.

**Parte 8** Si el método bajo prueba devuelve un valor, debe capturarse en una variable para que se pueda afirmar su valor.

**Parte 9** En el *Then* las pruebas unitarias sólo son valiosas si incluyen afirmaciones que validan que el método que se está probando devuelve el valor correcto y/o ajusta el estado de otros objetos como se esperaba. Sin aserciones, no tiene verificación, y su prueba es, en el mejor de los casos, una prueba de humo que brinda retroalimentación sólo cuando se lanza una excepción.

Los métodos de aserción JUnit, que se incluyen en la clase `org.junit.jupiter.api.Assertions` en JUnit 5 se usan comúnmente para determinar el estado de aprobación/rechazo de los casos de prueba. El marco JUnit solo informa las afirmaciones fallidas. Al igual que con las anotaciones, hay muchas opciones de afirmación.

En nuestro ejemplo JUnit anterior, usamos el método `assertEquals` (esperado, real, delta). El primer argumento es:

- El **gastos esperados**, que define el autor de la prueba.
- El **salida real**, que es el valor de retorno del método que se llama
- El **delta**, que permite una desviación aceptable entre los valores esperados y reales. Este delta es específico del hecho de que estamos validando el valor de un tipo doble.



## Escritura de pruebas unitarias efectivas

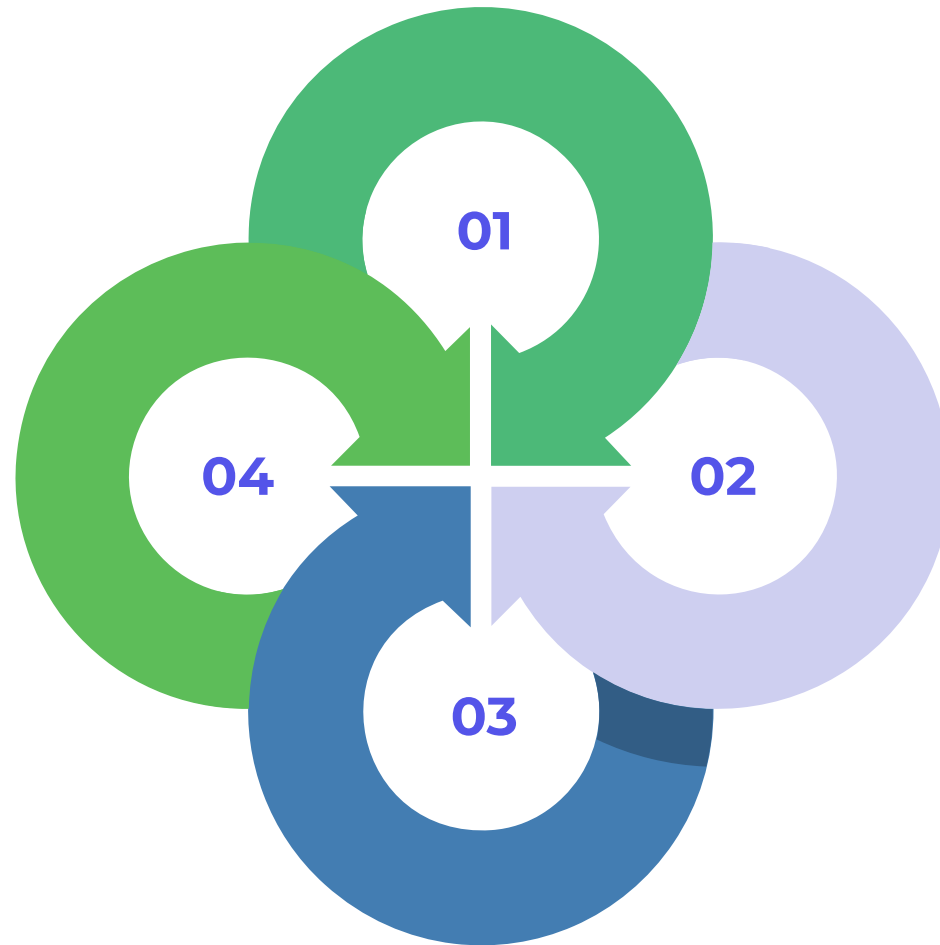
- Pruebe una unidad (función o componente) a la vez.
- Cubre tanto los escenarios positivos como los negativos.
- Utilice nombres descriptivos para los métodos y variables de prueba.
- Incluya aserciones para comprobar el comportamiento esperado.
- Mantenga las pruebas unitarias independientes y aisladas de otros componentes.



# Buenas prácticas

Aproveche los informes de las pruebas para identificar áreas con pruebas insuficientes.

Usa herramientas de compilación como Maven o Gradle para ejecutar pruebas automáticamente.



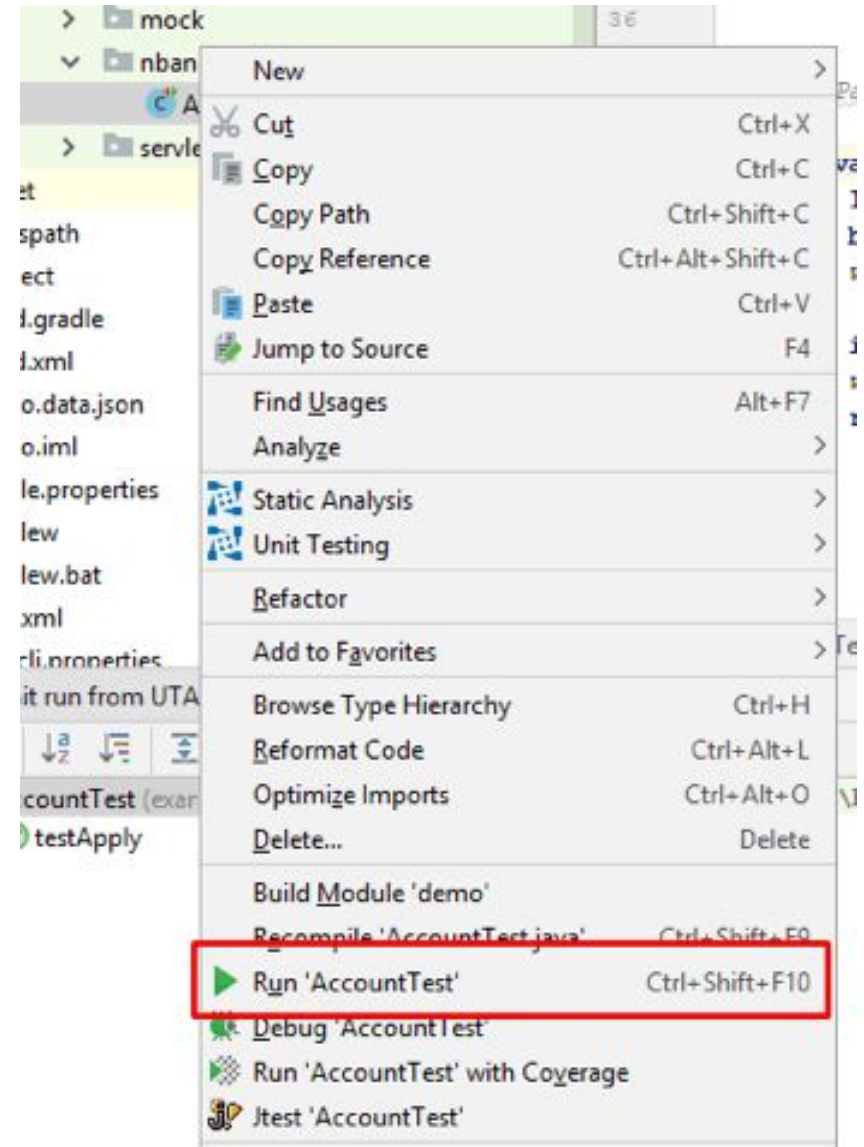
Las pruebas automatizadas se pueden ejecutar como parte de una canalización para garantizar la calidad del código antes de la implementación.

Integre las pruebas unitarias en el proceso de desarrollo, como la integración continua.

# Anexo: Ejecución de una prueba automática

## IntelliJ

Ejecutar una prueba en IntelliJ: En la ventana Proyecto, localice su prueba, haga clic con el botón derecho y seleccione Ejecutar 'testName'. Al igual que Eclipse, se abrirá una ventana JUnit con los resultados de la prueba.



**Hoja de trabajo**

**Figuras Geométricas**

**Gracias.** 😊

